Application System/400™

SC41-8226-02

AS/400 Machine Interface Functional Reference

Application System/400™

SC41-8226-02

AS/400 Machine Interface Functional Reference

Third Edition (November 1993)

The functions described in this publication apply to the IBM AS/400 machine interface.

Order publications through your IBM representative or the IBM branch serving your locality. Publications are not stocked at the address given below.

A Customer Satisfaction Feedback form for readers' comments is provided at the back of this publication. If the form has been removed, you may address your comments to:

Attn Department 245
IBM Corporation
3605 Highway 52 N
Rochester, MN 55901-7899 USA

or you can fax your comments to:

United States and Canada: 800+937-3430
Other countries: (+1)+507+253-5192

When you send information to IBM, you grant IBM a non-exclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you or restricting your use of it.

© Copyright International Business Machines Corporation 1991, 1993. All rights reserved.
Note to U.S. Government Users — Documentation related to restricted rights — Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corp.

Special Notices

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates.

Any reference to an IBM licensed program or other IBM product in this publication is not intended to state or imply that only IBM's program or other product may be used.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Commercial Relations, IBM Corporation, Purchase, NY 10577.

The following terms, denoted by an asterisk (*) in this publication, are trademarks of the IBM Corporation in the United States and/or other countries:

Application System/400	AS/400	IBM
400		

This publication could contain technical inaccuracies or typographical errors.

The information herein is subject to change.

Contents

Special Notices iii

About This Manual v
Who Should Use This Manual v
What You Should Know v
How This Manual Is Organized v

Overview

Chapter 1. Introduction 1-1
Instruction Format Conventions Used In This Manual 1-2
Definition of the operand syntax 1-3

Basic Function Instructions

Chapter 2. Computation and Branching Instructions 2-1
Add Logical Character (ADDLC) 2-3
Add Numeric (ADDN) 2-6
And (AND) 2-10
Branch (B) 2-13
Clear Bit in String (CLRBTS) 2-15
Compare Bytes Left-Adjusted (CMPBLA) 2-17
Compare Bytes Left-Adjusted with Pad (CMPBLAP) 2-19
Compare Bytes Right-Adjusted (CMPBRA) 2-21
Compare Bytes Right-Adjusted with Pad (CMPBRAP) 2-23
Compare Numeric Value (CMPNV) 2-25
Compress Data (CPRDATA) 2-28
Compute Array Index (CAI) 2-30
Compute Math Function Using One Input Value (CMF1) 2-32
Compute Math Function Using Two Input Values (CMF2) 2-41
Concatenate (CAT) 2-46
Convert BSC to Character (CVTBC) 2-48
Convert Character to BSC (CVTCB) 2-52
Convert Character to Hex (CVTCH) 2-55
Convert Character to MRJE (CVTCM) 2-57
Convert Character to Numeric (CVTCN) 2-62
Convert Character to SNA (CVTCS) 2-65
Convert Decimal Form to Floating-Point (CVTDFFP) 2-74
Convert External Form to Numeric Value (CVTEFN) 2-76
Convert Floating-Point to Decimal Form (CVTFPDF) 2-79
Convert Hex to Character (CVTHC) 2-82
Convert MRJE to Character (CVTMC) 2-84
Convert Numeric to Character (CVTNC) 2-88
Convert SNA to Character (CVTSC) 2-90
Copy Bits Arithmetic (CPYBTA) 2-100
Copy Bits Logical (CPYBTL) 2-102
Copy Bits with Left Logical Shift (CPYBTLLS) 2-104
Copy Bits with Right Arithmetic Shift (CPYBTRAS) 2-106
Copy Bits with Right Logical Shift (CPYBTRLS) 2-108
Copy Bytes Left-Adjusted (CPYBLA) 2-110

© Copyright IBM Corp. 1991, 1993

Copy Bytes Left-Adjusted with Pad (CPYBLAP) . . . 2-112
Copy Bytes Overlap Left-Adjusted (CPYBOLA) . . . 2-114
Copy Bytes Overlap Left-Adjusted with Pad (CPYBOLAP) . . . 2-116
Copy Bytes Repeatedly (CPYBREP) . . . 2-118
Copy Bytes Right-Adjusted (CPYBRA) . . . 2-120
Copy Bytes Right-Adjusted with Pad (CPYBRAP) . . . 2-122
Copy Bytes to Bits Arithmetic (CPYBBTA) . . . 2-124
Copy Bytes to Bits Logical (CPYBBTL) . . . 2-126
Copy Extended Characters Left-Adjusted With Pad (CPYECLAP) . . . 2-128
Copy Hex Digit Numeric to Numeric (CPYHEXNN) . . . 2-132
Copy Hex Digit Numeric to Zone (CPYHEXNZ) . . . 2-134
Copy Hex Digit Zone To Numeric (CPYHEXZN) . . . 2-136
Copy Hex Digit Zone To Zone (CPYHEXZZ) . . . 2-138
Copy Numeric Value (CPYNV) . . . 2-140
Decompress Data (DCPDATA) . . . 2-143
Divide (DIV) . . . 2-146
Divide with Remainder (DIVREM) . . . 2-150
Edit (EDIT) . . . 2-154
Exchange Bytes (EXCHBY) . . . 2-162
Exclusive Or (XOR) . . . 2-164
Extended Character Scan (ECSCAN) . . . 2-167
Extract Exponent (EXTREXP) . . . 2-171
Extract Magnitude (EXTRMAG) . . . 2-174
Multiply (MULT) . . . 2-177
Negate (NEG) . . . 2-181
Not (NOT) . . . 2-184
Or (OR) . . . 2-187
Remainder (REM) . . . 2-190
Scale (SCALE) . . . 2-194
Scan (SCAN) . . . 2-198
Scan with Control (SCANWC) . . . 2-201
Search (SEARCH) . . . 2-209
Set Bit in String (SETBTS) . . . 2-212
Set Instruction Pointer (SETIP) . . . 2-214
Store and Set Computational Attributes (SSCA) . . . 2-216
Subtract Logical Character (SUBLC) . . . 2-220
Subtract Numeric (SUBN) . . . 2-223
Test and Replace Characters (TSTRPLC) . . . 2-227
Test Bit in String (TSTBTS) . . . 2-229
Test Bits Under Mask (TSTBUM) . . . 2-231
Translate (XLATE) . . . 2-233
Translate with Table (XLATEWT) . . . 2-235
| Translate with Table and DBCS Skip (XLATWTDS) . . . 2-237
Trim Length (TRIML) . . . 2-240
Verify (VERIFY) . . . 2-242

Chapter 3. Date/Time/Timestamp Instructions . . . 3-1
External Data Formats . . . 3-3
Date, Time, and Timestamp Concepts . . . 3-3
Compute Date Duration (CDD) . . . 3-13
Compute Time Duration (CTD) . . . 3-16
Compute Timestamp Duration (CTSD) . . . 3-19
Convert Date (CVTD) . . . 3-22
Convert Time (CVTT) . . . 3-25
Convert Timestamp (CVTTS) . . . 3-28

Decrement Date (DECD) . . . 3-31
Decrement Time (DECT) . . . 3-35
Decrement Timestamp (DECTS) . . . 3-38
Increment Date (INCD) . . . 3-42
Increment Time (INCT) . . . 3-46
Increment Timestamp (INCTS) . . . 3-49

Chapter 4. Pointer/Name Resolution Addressing Instructions . . . 4-1
Compare Pointer for Object Addressability (CMPPTRA) . . . 4-3
Compare Pointer for Space Addressability (CMPPSPAD) . . . 4-5
| Compare Pointers for Equality (CMPPTRE) . . . 4-7
Compare Pointer Type (CMPPTRT) . . . 4-9
Copy Bytes with Pointers (CPYBWP) . . . 4-12
Resolve Data Pointer (RSLVDP) . . . 4-14
Resolve System Pointer (RSLVSP) . . . 4-17
Set Space Pointer from Pointer (SETSPPFP) . . . 4-22
Set System Pointer from Pointer (SETSPFP) . . . 4-24

Chapter 5. Space Addressing Instructions . . . 5-1
Add Space Pointer (ADDSPP) . . . 5-3
Compare Space Addressability (CMPSPAD) . . . 5-5
Set Data Pointer (SETDP) . . . 5-7
Set Data Pointer Addressability (SETDPADR) . . . 5-9
Set Data Pointer Attributes (SETDPAT) . . . 5-11
Set Space Pointer (SETSPP) . . . 5-14
Set Space Pointer with Displacement (SETSPPD) . . . 5-16
Set Space Pointer Offset (SETSPPO) . . . 5-18
Store Space Pointer Offset (STSPPO) . . . 5-20
Subtract Space Pointer Offset (SUBSPP) . . . 5-22
Subtract Space Pointers For Offset (SUBSPPFO) . . . 5-24

Chapter 6. Space Management Instructions . . . 6-1
Create Space (CRTS) . . . 6-3
Materialize Space Attributes (MATS) . . . 6-11
Modify Space Attributes (MODS) . . . 6-15

| **Chapter 7. Heap Management Instructions** . . . 7-1
| Allocate Heap Space Storage (ALCHSS) . . . 7-3
| Create Heap Space (CRTHS) . . . 7-6
| Destroy Heap Space (DESHS) . . . 7-11
| Free Heap Space Storage (FREHSS) . . . 7-13
| Free Heap Space Storage From Mark (FREHSSMK) . . . 7-15
| Materialize Heap Space Attributes (MATHSAT) . . . 7-17
| Reallocate Heap Space Storage (REALCHSS) . . . 7-22
| Set Heap Space Storage Mark (SETHSSMK) . . . 7-25

Chapter 8. Program Management Instructions . . . 8-1
| Materialize Bound Program (MATBPGM) . . . 8-3
Materialize Program (MATPG) . . . 8-24

Chapter 9. Program Execution Instructions . . . 9-1
Activate Program (ACTPG) . . . 9-3
Call External (CALLX) . . . 9-5
Call Internal (CALLI) . . . 9-9
Clear Invocation Exit (CLRIEXIT) . . . 9-11

Deactivate Program (DEACTPG) 9-12
End (END) 9-14
| Materialize Activation Attributes (MATACTAT) 9-15
| Materialize Activation Group Attributes (MATAGPAT) 9-19
Modify Automatic Storage Allocation (MODASA) 9-23
Return External (RTX) 9-25
Set Argument List Length (SETALLEN) 9-27
Set Invocation Exit (SETIEXIT) 9-29
Store Parameter List Length (STPLLEN) 9-31
Transfer Control (XCTL) 9-33

Chapter 10. Program Creation Control Instructions 10-1
No Operation (NOOP) 10-3
No Operation and Skip (NOOPS) 10-4
Override Program Attributes (OVRPGATR) 10-5

Chapter 11. Independent Index Instructions 11-1
Create Independent Index (CRTINX) 11-3
Destroy Independent Index (DESINX) 11-10
Find Independent Index Entry (FNDINXEN) 11-12
Insert Independent Index Entry (INSINXEN) 11-16
Materialize Independent Index Attributes (MATINXAT) 11-19
Modify Independent Index (MODINX) 11-23
Remove Independent Index Entry (RMVINXEN) 11-26

Chapter 12. Queue Management Instructions 12-1
Dequeue (DEQ) 12-3
Enqueue (ENQ) 12-9
Materialize Queue Attributes (MATQAT) 12-12
Materialize Queue Messages (MATQMSG) 12-16

Chapter 13. Object Lock Management Instructions 13-1
Lock Object (LOCK) 13-3
Lock Space Location (LOCKSL) 13-8
Materialize Data Space Record Locks (MATDRECL) 13-13
Materialize Process Locks (MATPRLK) 13-17
Materialize Process Record Locks (MATPRECL) 13-20
Materialize Selected Locks (MATSELLK) 13-24
Transfer Object Lock (XFRLOCK) 13-27
Unlock Object (UNLOCK) 13-30
Unlock Space Location (UNLOCKSL) 13-33

Chapter 14. Exception Management Instructions 14-1
Materialize Exception Description (MATEXCPD) 14-3
Modify Exception Description (MODEXCPD) 14-6
Retrieve Exception Data (RETEXCPD) 14-9
Return From Exception (RTNEXCP) 14-12
Sense Exception Description (SNSEXCPD) 14-15
Signal Exception (SIGEXCP) 14-19
Test Exception (TESTEXCP) 14-24

| **Chapter 15. Queue Space Management Instructions** 15-1
| Materialize Process Message (MATPRMSG) 15-3

Extended Function Instructions

Chapter 16. Context Management Instructions . . . 16-1
Materialize Context (MATCTX) . . . 16-3

Chapter 17. Authorization Management Instructions . . . 17-1
Materialize Authority (MATAU) . . . 17-3
Materialize Authority List (MATAL) . . . 17-7
Materialize Authorized Objects (MATAUOBJ) . . . 17-12
Materialize Authorized Users (MATAUU) . . . 17-20
Materialize User Profile (MATUP) . . . 17-25
Test Authority (TESTAU) . . . 17-29
Test Extended Authorities (TESTEAU) . . . 17-34

Chapter 18. Process Management Instructions . . . 18-1
| Materialize Process Activation Groups (MATPRAGP) . . . 18-3
Materialize Process Attributes (MATPRATR) . . . 18-5
Wait On Time (WAITTIME) . . . 18-18

Chapter 19. Resource Management Instructions . . . 19-1
Ensure Object (ENSOBJ) . . . 19-3
Materialize Access Group Attributes (MATAGAT) . . . 19-5
Materialize Resource Management Data (MATRMD) . . . 19-9
Set Access State (SETACST) . . . 19-31

Chapter 20. Dump Space Management Instructions . . . 20-1
Materialize Dump Space (MATDMPS) . . . 20-3

Chapter 21. Machine Observation Instructions . . . 21-1
| Find Relative Invocation Number (FNDRINVN) . . . 21-3
Materialize Instruction Attributes (MATINAT) . . . 21-8
Materialize Invocation (MATINV) . . . 21-14
| Materialize Invocation Attributes (MATINVAT) . . . 21-18
Materialize Invocation Entry (MATINVE) . . . 21-28
Materialize Invocation Stack (MATINVS) . . . 21-32
Materialize Pointer (MATPTR) . . . 21-37
Materialize Pointer Locations (MATPTRL) . . . 21-46
Materialize System Object (MATSOBJ) . . . 21-48

Chapter 22. Machine Interface Support Functions Instructions . . . 22-1
Materialize Machine Attributes (MATMATR) . . . 22-4
Materialize Machine Data (MATMDATA) . . . 22-30

Instruction support interfaces

Chapter 23. Exception Specifications . . . 23-1
Machine Interface Exception Data . . . 23-2
Exception List . . . 23-3

Appendix A. Instruction Summary . . . A-1
Instruction Stream Syntax . . . A-3

Index . . . X-1

About This Manual

The information contained in *AS/400 Machine Interface Functional Reference* has not been submitted to any formal IBM test and is distributed on an 'as is' basis without any warranty either expressed or implied. This manual is written for release 3 of AS/400 Vertical Licensed Integrated Code (VLIC) and may not discuss all the functions available on your AS/400 system.

The *AS/400 Machine Interface Functional Reference* defines the AS/400 Machine Interface to instructions and exceptions.

This manual may refer to products that are announced but are not yet available.

Who Should Use This Manual

This manual is intended for knowledgeable system programmers having substantial experience on AS/400 computer systems.

What You Should Know

The reader should know one more high level languages, assembly languages of other computers, and understand instruction set architectures. The reader would do well to study capability-based computer architectures.

The reader should be familiar with AS/400 objects and their intended use.

How This Manual Is Organized

The *AS/400 Machine Interface Functional Reference* is organized into three parts:

1. Basic Function Instructions

   These instructions provide a basic set of functions commonly needed by most programs executing on the machine. Because of the basic nature of these instructions, they tend to experience less change in their operation in different machine implementations than the extended function instructions.

2. Extended Function Instructions

   These instructions provide an extended set of functions which can be used to control and monitor the operation of the machine. Because of the more complicated nature of these instructions, they are more exposed to changes in their operation in different machine implementations than the basic function instructions.

3. Instruction Support Interfaces

   This part of the document defines those portions of the Machine Interface which provide support for functions or data used pervasively on all instructions. It discusses the exceptions and program objects which can be operated on by instructions.

© Copyright IBM Corp. 1991, 1993

Overview

© Copyright IBM Corp. 1991, 1993

Chapter 1. Introduction

This chapter contains the following:

- Detailed descriptions of the AS/400 machine interface instruction fields and the formats of these fields
- A description of the format used in describing each instruction
- A list of the terms in the syntax that define the characteristics of the operands

You should read this chapter in its entirety before attempting to write instructions.

Instruction Operands

Each instruction requires from zero to four operands. Each operand may consist of one or more fields that contain either a null operand specification, an immediate data value, or a reference to an ODT object. The size of the operand field depends on the version of the program template. If the version number is 0, the size of the operand field is 2 bytes. If the version number is 1, the size of the operand field is 3 bytes.

Null Operands: Certain instructions allow certain operands to be null. In general, a null operand means that some optional function of the instruction is not to be performed or that a default action is to be performed by the instruction.

Immediate Operands: The value of this type of operand is encoded in the instruction operand. Immediate operands may have the following values:

- Signed binary—representing a binary value of negative 4096 to positive 4095.
- Unsigned binary—representing a binary value of 0 to 8191.
- Byte string—representing a single byte value from hex 00 to hex FF.
- Absolute instruction number—representing an instruction number in the range of 1 to 8191.
- Relative instruction number—representing a displacement of an instruction relative to the instruction in which the operand occurs. This operand value may identify an instruction displacement of negative 4096 to positive 4095.

ODT Object References: This type of operand contains a reference (possibly qualified) to an object in the ODT. Operands that are ODT object references may be simple operands or compound operands.

Simple Operands: The value encoded in the operand refers to a specific object defined in the ODT. Simple operands consist of a single 2-byte operand entry.

Compound Operands: A compound operand consists of a primary (2-byte) operand and a series of one to three secondary (2-byte) operands. The primary operand is an ODT reference to a base object while the secondary operands serve as qualifiers to the base object.

A compound operand may have the following uses:

- Subscript references

  An individual element of a data object array, a pointer array, or an instruction definition list may be referenced with a subscript compound operand. The operand consists of a primary reference to the array and a secondary operand to specify the index value to an element of the array.

© Copyright IBM Corp. 1991, 1993

- Substring references

  A portion of a character scalar data object may be referenced as an instruction operand through a substring compound operand. The operand consists of a primary operand to reference the base string object and secondary references to specify the value of an index (position) and a value for the length of the substring.

  The length secondary operand field can specify whether to allow or not allow for a null substring reference (a length value of zero).

- Explicit base references

  An instruction operand may specify an explicit override for the base pointer for a based data object or a based addressing object. The operand consists of a primary operand reference to the based object and a secondary operand reference to the pointer on which to base the object for this operand. The override is in effect for the single operand. The displacement implicit in the ODT definition of the primary operand and the addressability contained in the explicit pointer are combined to provide an address for the operand.

The explicit base may be combined with either the subscript or the substring compound operands to provide a based subscript compound operand or a based substring compound operand.

Instruction Format Conventions Used In This Manual

The user of this manual must be aware that not every instruction uses every field described in this section. Only the information pertaining to the fields that are used by an instruction is provided for each instruction.

In this manual, each instruction is formatted with the instruction name followed by its base mnemonic. Following this is the operation code (op code) in hexadecimal and the number of operands with their general meaning.

Example:

ADD NUMERIC (ADDN)

Op Code (Hex)	**Operand 1**	**Operand 2**	**Operand 3**
1043	Sum	Addend 1	Addend 2

This information is followed by the operands and their syntax. See "Definition of the Operand Syntax" later in this chapter for a detailed discussion of the syntax of instruction operands.

Example:

Operand 1: Numeric variable scalar.

Operand 2: Numeric scalar.

Operand 3: Numeric scalar.

ILE access

```
A description of the parameters for Integrated Language Environment
access to the instruction is given. See the corresponding ILE
language reference manual for details as to how this information
should be interpreted for a given language.
```

Description: A detailed description and a functional definition of the instruction is given.

The terms are defined or fields in a template are described, they are highlighted as follows: **term definition**. When a term is referenced or a value of a field is refered to, it is highlighted as follows: *term reference*.

Fields in a template are generally described in the same order as they are defined in the template. However, some fields are more appropriately described with other related field, so they may not appear in exact order.

Authorization Required: A list of the object authorization required for each of the operands in the instruction or for any objects subsequently referenced by the instruction is given.

Lock Enforcement: Describes the specification of the lock states that are to be enforced during execution of the instruction.

The following states of enforcement can be specified for an instruction:

- Enforcement for materialization

  Access to a system object is allowed if no other process is holding a locked exclusive no read (LENR) lock on the object. In general, this rule applies to instructions that access an object for materialization and retrieval.

- Enforcement for modification

  Access to a system object is allowed if no other process is holding a locked exclusive no read (LENR) or locked exclusive allow read (LEAR) lock. In general, this rule applies to instructions that modify or alter the contents of a system object.

- Enforcement of object control

  Access is prohibited if another process is holding any lock on the system object. In general, this rule applies to instructions that destroy or rename a system object.

Limitations: These are the limits that apply to the functions performed by the instruction.

Resultant Conditions: These are the conditions that can be set at the end of the standard operation in order to perform a conditional branch or set a conditional indicator.

Exceptions: The "exceptions" sections contain a list of exceptions that can be caused by the instruction. Exceptions related to specific operands are indicated for each exception by the exception under the heading operand. An entry under the word, other, indicates that the exception applies to the instruction but not to a particular operand.

A detailed description of exceptions is in Chapter 23, "Exception Specifications" on page 23-1.

Definition of the operand syntax

Syntax consists of the allowable choices for each instruction operand. The following are the common terms used in the syntax and the meanings of those terms:

- *Numeric*: Numeric attribute of binary, packed decimal, zoned-decimal, or floating-point
- *Character*: character attribute
- *Scalar*:
  - Scalar data object that is not an array (see note 1)
  - Constant scalar object

  - Immediate operand (signed or unsigned)
  - Element of an array of scalars (see notes 1 and 2)
  - Substring of a character scalar or a character scalar constant data object (see notes 1 and 3)
- *Data Pointer Defined Scalar*:
  - A scalar defined by a data pointer
  - Substring of a character scalar defined by a data pointer (see notes 1 and 3)
- *Pointer*:
  - Pointer data object that is not an array (see note 1)
  - Element of an array of pointers (see notes 1 and 2)
  - Space pointer machine object
- *Array*: An array of scalars or an array of pointers (see note 1)
- *Variable Scalar*: Same as scalar except constant scalar objects and immediate operand values are excluded.
- *Data Pointer*: A pointer data object that is to be used as a data pointer.
  - If the operand is a source operand, the pointer storage form must contain a data pointer when the instruction is executed.
  - If the operand is a receiver operand, a data pointer is constructed by the instruction in the specified area regardless of its current contents (see note 4).
- *Space Pointer*: A space pointer data object or a space pointer machine object.
- *Space Pointer Data Object*: A pointer data object that is to be used as a space pointer.
  - If the operand is a source operand, the pointer storage form must contain a space pointer when the instruction is executed.
  - If the operand is a receiver operand, a space pointer is constructed by the instruction in the specified area regardless of its current contents (see note 4).
- *System Pointer*: a pointer data object that is to be used as a system pointer.
  - If the operand is a source operand, the specified area must contain a system pointer when the instruction is executed.
  - If the operand is a receiver operand, a system pointer is constructed by the instruction in the specified area regardless of its current contents (see note 4).
- *Relative Instruction Number*: Signed immediate operand.
- *Instruction Number*: Unsigned immediate operand.
- *Instruction Pointer*: A pointer data object that is to be used as an instruction pointer.
  - If the operand is a source operand, the specified area must contain an instruction pointer when the instruction is executed.
  - If the operand is a receiver operand, an instruction pointer is constructed by the instruction in the specified area regardless of its current contents (see notes 4 and 5).
- *Instruction Definition List Element*: An entry in an instruction definition list that can be used as a branch target. A compound subscript operand form must always be used (see note 5).

Notes:

1. An instruction operand in which the primary operand is a scalar or a pointer may also have an operand form in which an explicit base pointer is specified.

   See "ODT Object References" earlier in this chapter for more information on compound operands.

2. A compound subscript operand may be used to select a specific element from an array of scalars or from an array of pointers.

   See "ODT Object references" earlier in this chapter for more information on compound operands.

3. A compound substring operand may be used to define a substring of a character scalar, or a character constant scalar object.

   A compound substring operand that disallows a null substring reference (a length value of zero) may, unless precluded by the particular instruction, be specified for any operand syntactically defined as allowing a character scalar. A compound substring operand that allows a null substring reference may be specified for an operand syntactically defined as allowing a character scalar only if the instruction specifies that it is allowed. Whether a compound substring operand does or does not allow a null substring reference is controlled through the specification of the length secondary operand field.

   See "ODT Object References" earlier in this chapter for more information on compound operands.

4. A compound subscript operand form may be used to select an element from an array of pointers to act as the operand for an instruction. See "ODT Object References" earlier in this chapter for more information on compound operands.

5. Compound subscript forms are not allowed on branch target operands that are used for conditional branching. Selection of elements of instruction pointer arrays and elements of instruction definition lists may, however, be referenced for branch operands by the branch instruction.

Alternate choices of operand types and the allowable variations within each choice are indicated in the syntax descriptions as shown in the following example.

Operand 1: Numeric variable scalar.

Operand 2: Numeric scalar.

Operand 3: Instruction number, branch point or instruction pointer..

Operand 1 must be variable scalar. Operands 1 and 2 must be numeric. Operand 3 can be an instruction number, branch point or instruction pointer.

When a length is specified in the syntax for the operand, character scalar operands must be at least the size specified. Any excess beyond that required by the instruction is ignored.

Scalar operands that are operated on by instructions requiring 1-byte operands, such as pad values or indicator operands, can be greater than 1 byte in length; however, only the first byte of the character string is used. The remaining bytes are ignored by the instruction.

Basic Function Instructions

These instructions provide a basic set of functions commonly needed by most programs executing on the machine. Because of the basic nature of these instructions, they tend to experience less change in their operation in different machine implementations than the extended function instructions. Therefore, it is recommended that, where possible, programs be limited to using just these basic function instructions to minimize the impacts which can arise in moving to different machine implementations.

© Copyright IBM Corp. 1991, 1993

Chapter 2. Computation and Branching Instructions

This chapter describes all the instructions used for computation and branching. These instructions are arranged in alphabetic order. For an alphabetic summary of all the instructions, see Appendix A, "Instruction Summary."

Add Logical Character (ADDLC) . . . 2-3
Add Numeric (ADDN) . . . 2-6
And (AND) . . . 2-10
Branch (B) . . . 2-13
Clear Bit in String (CLRBTS) . . . 2-15
Compare Bytes Left-Adjusted (CMPBLA) . . . 2-17
Compare Bytes Left-Adjusted with Pad (CMPBLAP) . . . 2-19
Compare Bytes Right-Adjusted (CMPBRA) . . . 2-21
Compare Bytes Right-Adjusted with Pad (CMPBRAP) . . . 2-23
Compare Numeric Value (CMPNV) . . . 2-25
Compress Data (CPRDATA) . . . 2-28
Compute Array Index (CAI) . . . 2-30
Compute Math Function Using One Input Value (CMF1) . . . 2-32
Compute Math Function Using Two Input Values (CMF2) . . . 2-41
Concatenate (CAT) . . . 2-46
Convert BSC to Character (CVTBC) . . . 2-48
Convert Character to BSC (CVTCB) . . . 2-52
Convert Character to Hex (CVTCH) . . . 2-55
Convert Character to MRJE (CVTCM) . . . 2-57
Convert Character to Numeric (CVTCN) . . . 2-62
Convert Character to SNA (CVTCS) . . . 2-65
Convert Decimal Form to Floating-Point (CVTDFFP) . . . 2-74
Convert External Form to Numeric Value (CVTEFN) . . . 2-76
Convert Floating-Point to Decimal Form (CVTFPDF) . . . 2-79
Convert Hex to Character (CVTHC) . . . 2-82
Convert MRJE to Character (CVTMC) . . . 2-84
Convert Numeric to Character (CVTNC) . . . 2-88
Convert SNA to Character (CVTSC) . . . 2-90
Copy Bits Arithmetic (CPYBTA) . . . 2-100
Copy Bits Logical (CPYBTL) . . . 2-102
Copy Bits with Left Logical Shift (CPYBTLLS) . . . 2-104
Copy Bits with Right Arithmetic Shift (CPYBTRAS) . . . 2-106
Copy Bits with Right Logical Shift (CPYBTRLS) . . . 2-108
Copy Bytes Left-Adjusted (CPYBLA) . . . 2-110
Copy Bytes Left-Adjusted with Pad (CPYBLAP) . . . 2-112
Copy Bytes Overlap Left-Adjusted (CPYBOLA) . . . 2-114
Copy Bytes Overlap Left-Adjusted with Pad (CPYBOLAP) . . . 2-116
Copy Bytes Repeatedly (CPYBREP) . . . 2-118
Copy Bytes Right-Adjusted (CPYBRA) . . . 2-120
Copy Bytes Right-Adjusted with Pad (CPYBRAP) . . . 2-122
Copy Bytes to Bits Arithmetic (CPYBBTA) . . . 2-124
Copy Bytes to Bits Logical (CPYBBTL) . . . 2-126
Copy Extended Characters Left-Adjusted With Pad (CPYECLAP) . . . 2-128

© Copyright IBM Corp. 1991, 1993

Copy Hex Digit Numeric to Numeric (CPYHEXNN) . . . 2-132
Copy Hex Digit Numeric to Zone (CPYHEXNZ) . . . 2-134
Copy Hex Digit Zone To Numeric (CPYHEXZN) . . . 2-136
Copy Hex Digit Zone To Zone (CPYHEXZZ) . . . 2-138
Copy Numeric Value (CPYNV) . . . 2-140
Decompress Data (DCPDATA) . . . 2-143
Divide (DIV) . . . 2-146
Divide with Remainder (DIVREM) . . . 2-150
Edit (EDIT) . . . 2-154
Exchange Bytes (EXCHBY) . . . 2-162
Exclusive Or (XOR) . . . 2-164
Extended Character Scan (ECSCAN) . . . 2-167
Extract Exponent (EXTREXP) . . . 2-171
Extract Magnitude (EXTRMAG) . . . 2-174
Multiply (MULT) . . . 2-177
Negate (NEG) . . . 2-181
Not (NOT) . . . 2-184
Or (OR) . . . 2-187
Remainder (REM) . . . 2-190
Scale (SCALE) . . . 2-194
Scan (SCAN) . . . 2-198
Scan with Control (SCANWC) . . . 2-201
Search (SEARCH) . . . 2-209
Set Bit in String (SETBTS) . . . 2-212
Set Instruction Pointer (SETIP) . . . 2-214
Store and Set Computational Attributes (SSCA) . . . 2-216
Subtract Logical Character (SUBLC) . . . 2-220
Subtract Numeric (SUBN) . . . 2-223
Test and Replace Characters (TSTRPLC) . . . 2-227
Test Bit in String (TSTBTS) . . . 2-229
Test Bits Under Mask (TSTBUM) . . . 2-231
Translate (XLATE) . . . 2-233
Translate with Table (XLATEWT) . . . 2-235
| Translate with Table and DBCS Skip (XLATWTDS) . . . 2-237
Trim Length (TRIML) . . . 2-240
Verify (VERIFY) . . . 2-242

Add Logical Character (ADDLC)

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand 3	Operand [4-7]
ADDLC 1023		Sum	Addend 1	Addend 2	
ADDLCI 1823	Indicator options	Sum	Addend 1	Addend 2	Indicator targets
ADDLCB 1C23	Branch options	Sum	Addend 1	Addend 2	Branch targets

Operand 1: Character variable scalar (fixed-length).

Operand 2: Character scalar (fixed-length).

Operand 3: Character scalar (fixed-length).

Operand 4-7:

- *Branch Form*-Branch point, instruction pointer, relative instruction number, or absolute instruction number.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Short forms

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand [3-6]
ADDLCS 1123		Sum/Addend 1	Addend 2	
ADDLCIS 1923	Indicator options	Sum/Addend 1	Addend 2	Indicator targets
ADDLCBS 1D23	Branch options	Sum/Addend 1	Addend 2	Branch targets

Operand 1: Character variable scalar (fixed-length).

Operand 2: Character scalar (fixed-length).

Operand 3-6:

- *Branch Form*-Branch point, instruction pointer, relative instruction number, or absolute instruction number.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Description: The unsigned binary value of the addend 1 operand is added to the unsigned binary value of the addend 2 operand and the result is placed in the sum operand.

Operands 1, 2, and 3 must be the same length; otherwise, the Create Program instruction signals an *invalid operand length* (hex 2A0A) exception. The length can be a maximum of 256 bytes.

The addition operation is performed according to the rules of algebra. The result value is then placed (left-adjusted) in the receiver operand with truncating or padding taking place on the right. The pad value used in this instruction is a byte value of hex 00.

If operands overlap but do not share all of the same bytes, results of operations performed on these operands are not predictable. If overlapped operands share all of the same bytes, the results are predictable when direct addressing is used. If indirect addressing is used (that is, based operands, parameters, strings with variable lengths, and arrays with variable subscripts), the results are not always predictable.

Resultant Conditions: The logical sum of the character scalar operands is:

- zero with no carry out of the leftmost bit position
- not-zero with no carry
- zero with carry
- not-zero with carry.

Exceptions

Exception		Operands 1	2	3	Other
06	Addressing				
	01 Spacing addressing violation	X	X	X	
	02 Boundary alignment	X	X	X	
	03 Range	X	X	X	
	06 Optimized addressability invalid	X	X	X	
08	Argument/parameter				
	01 Parameter reference violation	X	X	X	
10	Damage encountered				
	04 System object damage state	X	X	X	X
	44 Partial system object damage	X	X	X	X
1C	Machine-dependent exception				
	03 Machine storage limit exceeded				X
20	Machine support				
	02 Machine check				X
	03 Function check				X
22	Object access				
	01 Object not found	X	X	X	
	02 Object destroyed	X	X	X	
	03 Object suspended	X	X	X	
	08 object compressed				X
24	Pointer specification				
	01 Pointer does not exist	X	X	X	
	02 Pointer type invalid	X	X	X	
2C	Program execution				
	04 Invalid branch target				X
2E	Resource control limit				
	01 user profile storage limit exceeded				X

Exception		Operands 1	2	3	Other
36	Space management				
	01 space extension/truncation				X

Add Numeric (ADDN)

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand 3	Operand [4-7]
ADDN 1043		Sum	Addend	Augend	
ADDNR 1243		Sum	Addend	Augend	
ADDNI 1843	Indicator options	Sum	Addend	Augend	Indicator targets
ADDNIR 1A43	Indicator options	Sum	Addend	Augend	Indicator targets
ADDNB 1C43	Branch options	Sum	Addend	Augend	Branch targets
ADDNBR 1E43	Branch options	Sum	Addend	Augend	Branch targets

Operand 1: Numeric variable scalar

Operand 2: Numeric scalar

Operand 3: Numeric scalar

Operand 4-7:

- *Branch Form*-Branch point, instruction pointer, relative instruction number, or absolute instruction number.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Short forms

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand [3-6]
ADDNS 1143		Sum/Addend	Augend	
ADDNSR 1343		Sum/Addend	Augend	
ADDNIR 1943	Indicator options	Sum/Addend	Augend	Indicator targets
ADDNISR 1B43	Indicator options	Sum/Addend	Augend	Indicator targets
ADDNBS 1D43	Branch options	Sum/Addend	Augend	Branch targets
ADDNBSR 1F43	Branch options	Sum/Addend	Augend	Branch targets

Operand 1: Numeric variable scalar

Operand 2: Numeric scalar

Operand 3-6:

- *Branch Form*-Branch point, instruction pointer, relative instruction number, or absolute instruction number.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Caution: If operands overlap but do not share all of the same bytes, results of operations performed on these operands are not predictable. If overlapped operands share all of the same bytes, the results are predictable when direct addressing is used. If indirect addressing is used (that is, based operands, parameters, strings with variable lengths, and arrays with variable subscripts), the results are not always predictable.

Description: The Sum is the result of adding the Addend and Augend.

Operands can have floating-point, packed or zoned decimal, signed or unsigned binary type.

Source operands are the Addend and Augend. The receiver operand is the Sum.

If operands are not of the same type, addends are converted according to the following rules:

1. If any one of the operands has floating point type, addends are converted to floating point type.
2. Otherwise, if any one of the operands has zoned or packed decimal type, addends are converted to packed decimal.
3. Otherwise, the binary operands are converted to a like type. Note: unsigned binary(2) scalars are logically treated as signed binary(4) scalars.

Addend and Augend are added according to their type. Floating point operands are added using floating point addition. Packed decimal addends are added using packed decimal addition. Unsigned binary addition is used with unsigned addends. Signed binary addends are added using two's complement binary addition.

Better performance can be obtained if all operands have the same type. Signed and unsigned binary additions execute faster than either packed decimal or floating point additions.

Decimal operands used in floating-point operations cannot contain more than 15 total digit positions.

For a decimal operation, alignment of the assumed decimal point takes place by padding with 0's on the right end of the addend with lesser precision.

Floating-point addition uses exponent comparison and significand addition. Alignment of the binary point is performed, if necessary, by shifting the significand of the value with the smaller exponent to the right. The exponent is increased by one for each binary digit shifted until the two exponents agree.

The operation uses the lengths and the precision of the source and receiver operands to calculate accurate results. Operations performed in decimal are limited to 31 decimal digits in the intermediate result.

The addition operation is performed according to the rules of algebra.

The result of the operation is copied into the sum operand. If this operand is not the same type as that used in performing the operation, the resultant value is converted to its type. If necessary, the resultant value is adjusted to the length of the sum, aligned at the assumed decimal point of the sum operand, or both before being copied. If nonzero digits are truncated on the left end of the resultant value, a *size* (hex 0C0A) exception is signaled.

When the target of the instruction is signed or unsigned binary size. exceptions can be suppressed.

For the optional round form of the instruction, specification of a floating-point receiver operand is invalid.

For fixed-point operations, if nonzero digits are truncated off the left end of the resultant value, a *size* (hex 0C0A) exception is signaled.

For floating-point operations involving a fixed-point receiver field, if nonzero digits would be truncated off the left end of the resultant value, an *invalid floating-point conversion* (hex 0C0C) exception is signaled.

For a floating-point sum, if the exponent of the resultant value is either too large or too small to be represented in the sum field, the *floating-point overflow* (hex 0C06) and *floating-point underflow* (hex 0C07) exceptions are signaled, respectively.

If a decimal to binary conversion causes a *size* (hex 0C0A) exception to be signaled, the binary value contains the correct truncated result only if the decimal value contains 15 or fewer significant nonfractional digits.

Resultant Conditions

- Positive - The algebraic value of the numeric scalar sum operand is positive.
- Negative - The algebraic value of the numeric scalar sum operand is negative.
- Zero - The algebraic value of the numeric scalar sum operand is zero.
- Unordered - The value assigned a floating-point sum operand is NaN.

Exceptions

Exception		Operands 1	2	3	Other
06	Addressing				
	01 Spacing addressing violation	X	X	X	
	02 Boundary alignment	X	X	X	
	03 Range	X	X	X	
	06 Optimized addressability invalid	X	X	X	
08	Argument/parameter				
	01 Parameter reference violation	X	X	X	
0C	Computation				
	02 Decimal data		X	X	
	03 Decimal point alignment		X	X	
	06 Floating-point overflow	X			
	07 Floating-point underflow	X			
	09 Floating-point invalid operand		X	X	X
	0A Size	X			
	0C Invalid floating-point conversion	X			
	0D Floating-point inexact result	X			
10	Damage encountered				
	04 System object damage state	X	X	X	X

Exception		Operands 1	2	3	Other
	44 Partial system object damage	X	X	X	X
1C	Machine-dependent exception				
	03 Machine storage limit exceeded				X
20	Machine support				
	02 Machine check				X
	03 Function check				X
22	Object access				
	01 Object not found	X	X	X	
	02 Object destroyed	X	X	X	
	03 Object suspended	X	X	X	
	08 object compressed				X
24	Pointer specification				
	01 Pointer does not exist	X	X	X	
	02 Pointer type invalid	X	X	X	
2C	Program execution				
	04 Invalid branch target				X
2E	Resource control limit				
	01 user profile storage limit exceeded				X
36	Space management				
	01 space extension/truncation				X

And (AND)

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand 3	Operand [4-5]
AND 1093		Receiver	Source 1	Source 2	
ANDI 1893	Indicator options	Receiver	Source 1	Source 2	Indicator targets
ANDB 1C93	Branch options	Receiver	Source 1	Source 2	Branch targets

Operand 1: Character variable scalar or numeric variable scalar.

Operand 2: Character scalar or numeric scalar.

Operand 3: Character scalar or numeric scalar.

Operand 4-5:

- *Branch Form*-Branch point, instruction pointer, relative instruction number, or absolute instruction number.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Short forms

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand [3-4]
ANDS 1193		Receiver/Source 1	Source 2	
ANDIS 1993	Indicator options	Receiver/Source 1	Source 2	Indicator targets
ANDBS 1D93	Branch options	Receiver/Source 1	Source 2	Branch targets

Operand 1: Character variable scalar or numeric variable scalar.

Operand 2: Character scalar or numeric scalar.

Operand 3-4:

- *Branch Form*-Branch point, instruction pointer, relative instruction number, or absolute instruction number.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Description: The Boolean AND operation is performed on the string values in the source operands. The resulting string is placed in the receiver operand. The operands may be character or numeric scalars. They are both interpreted as bit strings. Substringing is supported for both character and numeric operands.

The length of the operation is equal to the length of the longer of the two source operands. The shorter of the two operands is logically padded on the right with hex 00 values. This assigns hex 00 values to the results for those bytes that correspond to the excess bytes of the longer operand.

The bit values of the result are determined as follows:

Source 1 Bit	Source 2 Bit	Result Bit
0	0	0
0	1	0
1	0	0
1	1	1

The result value is then placed (left-adjusted) in the receiver operand with truncating or padding taking place on the right. The pad value used in this instruction is a byte value of hex 00.

If operands overlap but do not share all of the same bytes, results of operations performed on these operands are not predictable. If overlapped operands share all of the same bytes, the results are predictable when direct addressing is used. If indirect addressing is used (that is, based operands, parameters, strings with variable lengths, and arrays with variable subscripts), the results are not always predictable.

Substring operand references that allow for a null substring reference (a length value of zero) may be specified for operands 1, 2, and 3. The effect of specifying a null substring reference for either or both of the source operands is that the result is all zero and instruction's resultant condition is *zero*. When a null substring reference is specified for the receiver, a result is not set and the instruction's resultant condition is *zero* regardless of the values of the source operands.

When the receiver operand is a numeric variable scalar, it is possible that the result produced will not be a valid value for the numeric type. This can occur due to padding with hex 00, due to truncation, or due to the resultant bit string produced by the instruction. The instruction completes normally and signals no exceptions for these conditions.

Resultant Conditions

- Zero - The bit value for the bits of the scalar receiver operand is either all zero or a null substring reference is specified for the receiver.
- Not zero - The bit value for the bits of the scalar receiver operand is not all zero.

Exceptions

Exception		Operands 1	2	3	Other
06	Addressing				
	01 Spacing addressing violation	X	X	X	
	02 Boundary alignment	X	X	X	
	03 Range	X	X	X	
	06 Optimized addressability invalid	X	X	X	
08	Argument/parameter				
	01 Parameter reference violation	X	X	X	
10	Damage encountered				
	04 System object damage state	X	X	X	X
	44 Partial system object damage	X	X	X	X
1C	Machine-dependent exception				
	03 Machine storage limit exceeded				X
20	Machine support				

Exception		Operands 1	2	3	Other
	02 Machine check				X
	03 Function check				X
22	Object access				
	01 Object not found	X	X	X	
	02 Object destroyed	X	X	X	
	03 Object suspended	X	X	X	
	08 object compressed				X
24	Pointer specification				
	01 Pointer does not exist	X	X	X	
	02 Pointer type invalid	X	X	X	
2C	Program execution				
	04 Invalid branch target				X
2E	Resource control limit				
	01 user profile storage limit exceeded				X
36	Space management				
	01 space extension/truncation				X

Branch (B)

Op Code (Hex)	Operand 1
1011	Branch target

Operand 1: Instruction number, relative instruction number, branch point, instruction pointer, or instruction definition list element.

Description: Control is unconditionally transferred to the instruction indicated in the branch target operand. The instruction number indicated by the branch target operand must be within the instruction stream containing the branch instruction.

The branch target may be an element of an array of instruction pointers or an element of an instruction definition list. The specific element can be identified by using a compound subscript operand.

Exceptions

Exception		Operand 1	Other
06	Addressing		
	01 Spacing addressing violation	X	
	02 Boundary alignment violation	X	
	03 Range	X	
08	Argument/parameter		
	01 Parameter reference violation	X	
10	Damage encountered		
	04 System object damage state	X	X
	44 Partial system object damage	X	X
1C	Machine-dependent exception		
	03 Machine storage limit exceeded		X
20	Machine support		
	02 Machine check		X
	03 Function check		X
22	Object access		
	01 Object not found	X	
	02 Object destroyed	X	
	03 Object suspended	X	
	08 object compressed		X
24	Pointer specification		
	01 Pointer does not exist	X	
	02 Pointer type invalid	X	
2C	Program execution		
	04 Invalid branch target	X	
2E	Resource control limit		

Exception		Operand 1	Other
	01 user profile storage limit exceeded		X
36	Space management		
	01 space extension/truncation		X

Clear Bit in String (CLRBTS)

Op Code (Hex)	Operand 1	Operand 2
102E	Receiver	Offset

Operand 1: Character variable scalar (fixed length) or numeric variable scalar.

Operand 2: Binary scalar.

Description: Clears the bit of the receiver operand as indicated by the bit offset operand.

The selected bit from the receiver operand is set to a value of binary 0.

The receiver operand can be character or numeric. The leftmost bytes of the receiver operand are used in the operation. The receiver operand is interpreted as a bit string with the bits numbered left to right from 0 to the total number of bits in the string minus 1.

The receiver cannot be a variable substring.

The offset operand indicates which bit of the receiver operand is to be cleared, with a offset of zero indicating the leftmost bit of the leftmost byte of the receiver operand.

If a offset value less than zero or beyond the length of the string is specified, a *scalar value invalid* (hex 3203) exception is signaled.

Exceptions

Exception		Operands 1	2	Other
06	Addressing			
	01 Spacing addressing violation	X	X	
	02 Boundary alignment violation	X	X	
	03 Range	X	X	
	06 Optimized addressability invalid	X	X	
08	Argument/parameter			
	01 Parameter reference violation	X	X	
10	Damage encountered			
	04 System object damage state			X
	44 Partial system object damage			X
1C	Machine-dependent exception			
	03 Machine storage limit exceeded			X
20	Machine support			
	02 Machine check			X
	03 Function check			X
22	Object access			
	02 Object destroyed	X	X	
	03 Object suspended	X	X	
	08 object compressed			X

Exception		Operands 1	Operands 2	Other
24	Pointer specification			
	01 Pointer does not exist	X	X	
	02 Pointer type invalid	X	X	
2A	Program creation			
	06 Invalid operand type	X	X	
	07 Invalid operand attribute	X	X	
	08 Invalid operand value range	X	X	
	0A Invalid operand length	X	X	
	0C Invalid operand odt reference	X	X	
	0D Reserved bits are not zero	X	X	
2E	Resource control limit			
	01 user profile storage limit exceeded			X
32	Scalar specification			
	01 Scalar type invalid	X	X	
	03 Scalar value invalid		X	
36	Space management			
	01 space extension/truncation			X

Compare Bytes Left-Adjusted (CMPBLA)

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand 3 [4, 5]
CMPBLAB 1CC2	Branch options	Compare operand 1	Compare operand 2	Branch targets
CMPBLAI 18C2	Indicator options	Compare operand 1	Compare operand 2	Indicator targets

Operand 1: Numeric scalar or character scalar.

Operand 2: Numeric scalar or character scalar.

Operand 3 [*4, 5*]:

- *Branch Form*-Instruction number, relative instruction number, branch point, or instruction pointer.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Description: This instruction compares the logical string values of two left-adjusted compare operands. The logical string value of the first compare operand is compared with the logical string value of the second compare operand (no padding done). Based on the comparison, the resulting condition is used with the extender field to:

- Transfer control conditionally to the instruction indicated in one of the branch target operands (branch form).
- Assign a value to each of the indicator operands (indicator form).

The compare operands can be either character or numeric. Any numeric operands are interpreted as logical character strings.

The compare operands are compared byte by byte, from left to right with no numeric conversions performed. The length of the operation is equal to the length of the shorter of the two compare operands. The comparison begins with the leftmost byte of each of the compare operands and proceeds until all bytes of the shorter compare operand have been compared or until the first unequal pair of bytes is encountered.

Substring operand references that allow for a null substring reference (a length value of zero) may be specified for operands 1 and 2. The effect of specifying a null substring reference for either or both compare operands is that the instruction's resultant condition is *equal*.

Resultant Conditions: The scalar first compare operand's string value is one of the following as compared to the second compare operand.

- higher
- lower
- equal

Exceptions

Exception		Operands 1	2	3 [4, 5]	Other
06	Addressing				
	01 Spacing addressing violation	X	X	X	
	02 Boundary alignment	X	X	X	

Exception		Operands 1	2	3 [4, 5]	Other
	03 Range	X	X	X	
	06 Optimized addressability invalid	X	X	X	
08	Argument/parameter				
	01 Parameter reference violation	X	X	X	
10	Damage encountered				
	04 System object damage state	X	X	X	X
	44 Partial system object damage	X	X	X	X
1C	Machine-dependent exception				
	03 Machine storage limit exceeded				X
20	Machine support				
	02 Machine check				X
	03 Function check				X
22	Object access				
	01 Object not found	X	X	X	
	02 Object destroyed	X	X	X	
	03 Object suspended	X	X	X	
	08 object compressed				X
24	Pointer specification				
	01 Pointer does not exist	X	X	X	
	02 Pointer type invalid	X	X	X	
2C	Program execution				
	04 Branch target invalid				X
2E	Resource control limit				
	01 user profile storage limit exceeded				X
36	Space management				
	01 space extension/truncation				X

Compare Bytes Left-Adjusted with Pad (CMPBLAP)

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand 3	Operand 4 [5, 6]
CMPBLAPB 1CC3	Branch options	Compare operand 1	Compare operand 2	Pad	Branch targets
CMPBLAPI 18C3	Indicator options	Compare operand 1	Compare operand 2	Pad	Indicator targets

Operand 1: Numeric scalar or character scalar.

Operand 2: Numeric scalar or character scalar.

Operand 3: Numeric scalar or character scalar.

Operand 4 [*5, 6*]:

- *Branch Form*-Instruction number, relative instruction number, branch point, or instruction pointer.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Description: This instruction compares the logical string values of two left-adjusted compare operands (padded if needed). The logical string value of the first compare operand is compared with the logical string value of the second compare operand. Based on the comparison, the resulting condition is used with the extender field to:

- Transfer control conditionally to the instruction indicated in one of the branch target operands (branch form).
- Assign a value to each of the indicator operands (indicator form).

The compare operands can be either character or numeric. Any numeric operands are interpreted as logical character strings.

The compare operands are compared byte by byte, from left to right with no numeric conversions being performed.

The length of the operation is equal to the length of the longer of the two compare operands. The shorter of the two compare operands is logically padded on the right with the 1-byte value indicated in the pad operand. If the pad operand is more than 1 byte in length, only its leftmost byte is used. The comparison begins with the leftmost byte of each of the compare operands and proceeds until all the bytes of the longer of the two compare operands have been compared or until the first unequal pair of bytes is encountered. All excess bytes in the longer of the two compare operands are compared to the pad value.

Substring operand references that allow for a null substring reference (a length value of zero) may be specified for operands 1 and 2. The effect of specifying a null substring reference for one of the compare operands is that the other compare operand is compared with an equal length string of pad character values. When a null substring reference is specified for both compare operands, the resultant condition is *equal*.

Substring operand references that allow for a null substring reference (a length value of zero) may not be specified for operand 3.

Resultant Conditions: The scalar first compare operand's string value is one of the following as compared to the second compare operand.

- higher
- lower
- equal

Exceptions

Exception		Operands 1	2	3	4 [5, 6]	Other
06	Addressing					
	01 Spacing addressing violation	X	X	X	X	
	02 Boundary alignment	X	X	X	X	
	03 Range	X	X	X	X	
	06 Optimized addressability invalid	X	X	X	X	
08	Argument/parameter					
	01 Parameter reference violation	X	X	X	X	
10	Damage encountered					
	04 System object damage state	X	X	X	X	X
	44 Partial system object damage	X	X	X	X	X
1C	Machine-dependent exception					
	03 Machine storage limit exceeded					X
20	Machine support					
	02 Machine check					X
	03 Function check					X
22	Object access					
	01 Object not found	X	X	X	X	
	02 Object destroyed	X	X	X	X	
	03 Object suspended	X	X	X	X	
	08 object compressed					X
24	Pointer specification					
	01 Pointer does not exist	X	X	X	X	
	02 Pointer type invalid	X	X	X	X	
2C	Program execution					
	04 Branch target invalid					X
2E	Resource control limit					
	01 user profile storage limit exceeded					X
36	Space management					
	01 space extension/truncation					X

Compare Bytes Right-Adjusted (CMPBRA)

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand 3 [4, 5]
CMPBRAB 1CC6	Branch options	Compare operand 1	Compare operand 2	Branch targets
CMPBRAI 18C6	Indicator options	Compare operand 1	Compare operand 2	Indicator targets

Operand 1: Numeric scalar or character scalar.

Operand 2: Numeric scalar or character scalar.

Operand 3 [4, 5]:

- *Branch Form*-Instruction number, relative instruction number, branch point, or instruction pointer.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Description: This instruction compares the logical string values of two right-adjusted compare operands. The logical string value of the first compare operand is compared with the logical string value of the second compare operand (no padding done). Based on the comparison, the resulting condition is used with the extender field to:

- Transfer control conditionally to the instruction indicated in one of the branch target operands (branch form).
- Assign a value to each of the indicator operands (indicator form).

The compare operands can be either string or numeric. Any numeric operands are interpreted as logical character strings.

The compare operands are compared byte by byte, from left to right with no numeric conversions performed. The length of the operation is equal to the length of the shorter of the two compare operands. The comparison begins with the leftmost byte of each of the compare operands and proceeds until all bytes of the shorter compare operand have been compared or until the first unequal pair of bytes is encountered.

Substring operand references that allow for a null substring reference (a length value of zero) may be specified for operands 1 and 2. The effect of specifying a null substring reference for either or both compare operands is that the instruction's resultant condition is *equal*.

Resultant Conditions: The scalar first compare operand's string value is one of the following as compared to the second compare operand.

- higher
- lower
- equal

Exceptions

Exception		Operands 1	2	3 [4, 5]	Other
06	Addressing				
	01 Spacing addressing violation	X	X	X	
	02 Boundary alignment violation	X	X	X	

Exception		Operands 1	2	3 [4, 5]	Other
	03 Range	X	X	X	
	06 Optimized addressability invalid	X	X	X	
08	Argument/parameter				
	01 Parameter reference violation	X	X	X	
10	Damage encountered				
	04 System object damage state	X	X	X	X
	44 Partial system object damage	X	X	X	X
1C	Machine-dependent exception				
	03 Machine storage limit exceeded				X
20	Machine support				
	02 Machine check				X
	03 Function check				X
22	Object access				
	01 Object not found	X	X	X	
	02 Object destroyed	X	X	X	
	03 Object suspended	X	X	X	
	08 object compressed				X
24	Pointer specification				
	01 Pointer does not exist	X	X	X	
	02 Pointer type invalid	X	X	X	
2C	Program execution				
	04 Branch target invalid			X	X
2E	Resource control limit				
	01 user profile storage limit exceeded				X
36	Space management				
	01 space extension/truncation				X

Compare Bytes Right-Adjusted with Pad (CMPBRAP)

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand 3	Operand 4 [5, 6]
CMPBRAPB 1CC7	Branch options	Compare operand 1	Compare operand 2	Pad	Branch targets
CMPBRAPI 18C7	Indicator options	Compare operand 1	Compare operand 2	Pad	Indicator targets

Operand 1: Numeric scalar or character scalar.

Operand 2: Numeric scalar or character scalar.

Operand 3: Numeric scalar or character scalar.

Operand 4 [*5, 6*]:

- *Branch Form*-Instruction number, relative instruction number, branch point, or instruction pointer.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Description: This instruction compares the logical string values of the right-adjusted compare operands (padded if needed). The logical string value of the first compare operand is compared with the logical string value of the second compare operand. Based on the comparison, the resulting condition is used with the extender field to:

- Transfer control conditionally to the instruction indicated in one of the branch target operands (branch form).
- Assign a value to each of the indicator operands (indicator form).

The compare operands can be either character or numeric. Any numeric operands are interpreted as logical character strings.

The compare operands are compared byte by byte, from left to right with no numeric conversions performed.

The length of the operation is equal to the length of the longer of the two compare operands. The shorter of the two compare operands is logically padded on the left with the 1-byte value indicated in the pad operand. If the pad operand is more than 1 byte in length, only its leftmost byte is used. The comparison begins with the leftmost byte of the longer of the compare operands. Any excess bytes (on the left) in the longer compare operand are compared with the pad value. All other bytes are compared with the corresponding bytes in the other compare operand. The operation proceeds until all bytes in the longer operand are compared or until the first unequal pair of bytes is encountered.

Substring operand references that allow for a null substring reference (a length value of zero) may be specified for operands 1 and 2. The effect of specifying a null substring reference for one of the compare operands is that the other compare operand is compared with an equal length string of pad character values. When a null substring reference is specified for both compare operands, the resultant condition is *equal*.

Substring operand references that allow for a null substring reference (a length value of zero) may not be specified for operand 3.

Resultant Conditions: The scalar first compare operand's string value is one of the following as compared to the second compare operand.

- higher
- lower
- equal

Exceptions

Exception		Operands 1	2	3	4 [5, 6]	Other
06	Addressing					
	01 Spacing addressing violation	X	X	X	X	
	02 Boundary alignment	X	X	X	X	
	03 Range	X	X	X	X	
	06 Optimized addressability invalid	X	X	X	X	
08	Argument/parameter					
	01 Parameter reference violation	X	X	X	X	
10	Damage encountered					
	04 System object damage state	X	X	X	X	X
	44 Partial system object damage	X	X	X	X	X
1C	Machine-dependent exception					
	03 Machine storage limit exceeded					X
20	Machine support					
	02 Machine check					X
	03 Function check					X
22	Object access					
	01 Object not found	X	X	X	X	
	02 Object destroyed	X	X	X	X	
	03 Object suspended	X	X	X	X	
	08 object compressed					X
24	Pointer specification					
	01 Pointer does not exist	X	X	X	X	
	02 Pointer type invalid	X	X	X	X	
2C	Program execution					
	04 Branch target invalid			X	X	
2E	Resource control limit					
	01 user profile storage limit exceeded					X
36	Space management					
	01 space extension/truncation					X

Compare Numeric Value (CMPNV)

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand 3 [4-6]
CMPNVB 1C46	Branch options	Compare operand 1	Compare operand 2	Branch targets
CMPNVI 1846	Indicator options	Compare operand 1	Compare operand 2	Indicator targets

Operand 1: Numeric scalar.

Operand 2: Numeric scalar.

Operand 3 [4-6]:

- *Branch Form*-Instruction number, relative instruction number, branch point, or instruction pointer.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Description: The numeric value of the first compare operand is compared with the signed or unsigned numeric value of the second compare operand. Based on the comparison, the resulting condition is used with the extender field to:

- Transfer control conditionally to the instruction indicated in one of the branch target operands (branch form).
- Assign a value to each of the indicator operands (indicator form).

For a decimal operation, alignment of the assumed decimal point takes place by padding with 0's on the right end of the compare operand with lesser precision.

Decimal operands used in floating-point operations cannot contain more than 15 total digit positions.

When both operands are signed numeric or both are unsigned numeric, the length of the operation is equal to the length of the longer of the two compare operands.

When one operand is signed numeric and the other operand unsigned numeric, the unsigned operand is converted to a signed value with more precision than its current size. The length of the operation is equal to the length of the longer of the two compare operands. A negative signed numeric value will always be less than a positive unsigned value.

Floating-point comparisons use exponent comparison and significand comparison. For a denormalized floating-point number, the comparison is performed as if the denormalized number had first been normalized.

For floating-point, two values compare unordered when at least one comparand is NaN. Every NaN compares unordered with everything including another NaN value.

Floating-point comparisons ignore the sign of zero. Positive zero always compares equal with negative zero.

A *floating-point invalid operand* (hex 0C09) exception is signaled when two floating-point values compare unordered and no branch or indicator option exists for any of the unordered, negation of unordered equal, or negation of equal resultant conditions.

When a comparison is made between a floating-point compare operand and a fixed-point decimal compare operand that contains fractional digit positions, a *floating-point inexact result* (hex 0C0D) exception may be signaled because of the implicit conversion from decimal to floating-point.

Resultant Conditions

- High-The first compare operand has a higher numeric value than the second compare operand.
- Low-The first compare operand has a lower numeric value than the second compare operand.
- Equal-The first compare operand has a equal numeric value than the second compare operand.
- Unordered-The first compare operand is unordered compared to the second compare operand.

Exceptions

Exception		Operands 1	2	3 [4-6]	Other
06	Addressing				
	01 Spacing addressing violation	X	X	X	
	02 Boundary alignment	X	X	X	
	03 Range	X	X	X	
	06 Optimized addressability invalid	X	X	X	
08	Argument/parameter				
	01 Parameter reference violation	X	X	X	
0C	Computation				
	02 Decimal data	X	X		
	03 Decimal point alignment	X	X		
	09 Floating-point invalid operand		X		X
	0D Floating-point inexact result				X
10	Damage encountered				
	04 System object damage state	X	X	X	X
	44 Partial system object damage	X	X	X	X
1C	Machine-dependent exception				
	03 Machine storage limit exceeded				X
20	Machine support				
	02 Machine check				X
	03 Function check				X
22	Object access				
	01 Object not found	X	X	X	
	02 Object destroyed	X	X	X	
	03 Object suspended	X	X	X	
	08 object compressed				X
24	Pointer specification				
	01 Pointer does not exist	X	X	X	
	02 Pointer type invalid	X	X	X	
2C	Program execution				
	04 Branch target invalid			X	
2E	Resource control limit				

Exception		Operands 1	2	3	[4-6]	Other
	01 user profile storage limit exceeded					X
36	Space management					
	01 space extension/truncation					X

Compress Data (CPRDATA)

Op Code (Hex)	Operand 1
1041	Compress Data template

Operand 1: Space pointer.

ILE access

```
CPRDATA (
     compress_data_template : space pointer
)
```

Description: The instruction compresses user data of a specified length. Operand 1 identifies a template which identifies the data to be compressed. The template also identifies the result space to receive the compressed data.

The Compress Data template must be aligned on a 16-byte boundary. The format is as follows:

- Source length — Bin(4)
- Result area length — Bin(4)
- Actual result length — Bin(4)*
- Compression algorithm — Bin(2)

  1 = Simple TERSE algorithm
  2 = IBM LZ1 algorithm
- Reserved (binary 0) — Char(18)
- Source space pointer — Space pointer
- Result space pointer — Space pointer

Note: The input value associated with template entries annotated with an asterisk (*) are ignored by the instruction; these fields are updated by the instruction to return information about instruction execution.

The data at the location specified by the **source space pointer** for the length specified by the **source length** is compressed and stored at the location specified by the **result space pointer**. The **actual result length** is set to the number of bytes in the compressed result. The source data is not modified.

The value of both the *source length* field and *result area length* field must be greater than zero. If the compressed result is longer than the result area (as specified by the **result area length**), the compression is stopped and only *result area length* bytes are stored.

The **compression algorithm** field specifies the algorithm used to compress the data. The *IBM LZ1 algorithm* tends to produce better compression on shorter input strings than the *simple TERSE algorithm*. The algorithm choice is stored in the compressed output data so the Decompress Data instruction will automatically select the correct decompression algorithm.

Only scalar (non-pointer) data is compressed, so any pointers in the data to be compressed are destroyed in the output of the Decompress Data instruction.

Authorization Required

- None

Lock Enforcement

- None

Exceptions

Exception		Operands 1	Other
06	Addressing		
	01 space addressing violation	X	X
	02 boundary alignment	X	
	03 range	X	
	06 optimized addressability invalid	X	
08	Argument/Parameter		
	01 Parameter reference violation	X	
10	Damage encountered		
	44 partial system object damage	X	X
1C	Machine-dependent exception		
	03 machine storage limit exceeded		X
20	Machine support		
	02 machine check		X
	03 function check		X
22	Object access		
	02 object destroyed	X	X
	03 object suspended	X	X
	08 object compressed		X
24	Pointer specification		
	01 pointer does not exist	X	X
	02 pointer type invalid	X	X
2E	Resource control limit		
	01 user profile storage limit exceeded		X
36	Space management		
	01 space extension/truncation	X	X
38	Template specification		
	01 template value invalid	X	
44	Domain		
	01 object domain error	X	

Compute Array Index (CAI)

Op Code (Hex)	Operand 1	Operand 2	Operand 3	Operand 4
1044	Array index	Subscript A	Subscript B	Dimension

Operand 1: Binary(2) variable scalar.

Operand 2: Binary(2) scalar.

Operand 3: Binary(2) scalar.

Operand 4: Binary(2) constant scalar object or immediate operand.

Description: This instruction provides the ability to reduce multidimensional array subscript values into a single index value which can then be used in referencing the single-dimensional arrays of the system. This index value is computed by performing the following arithmetic operation on the indicated operands.

Array Index = Subscript A + ((Subscript B-1) X Dimension)

The numeric value of the subscript B operand is decreased by 1 and multiplied by the numeric value of the dimension operand. The result of this multiplication is added to the subscript A operand and the sum is placed in the array index operand.

All the operands must be binary with any implicit conversions occurring according to the rules of arithmetic operations. The usual rules of algebra are observed concerning the subtraction, addition, and multiplication of operands.

This instruction provides for mapping multidimensional arrays to single-dimensional arrays. The elements of an array with the dimensions (d1, d2, d3, ..., dn) can be defined as a single-dimensional array with d1*d2*d3*...*dn elements. To reference a specific element of the multidimensional array with subscripts (s1,s2,s3,...sn), it is necessary to convert the multiple subscripts to a single subscript for use in the single-dimensional AS/400 array. This single subscript can be computed using the following:

```
s1+((s2-1)*d1)+(s3-1)*d1*d2)+...+((sn-1)*d*d2*d3*...*dm)
```

where m = n-1

The CAI instruction is used to form a single index value from two subscript values. To reduce N subscript values into a single index value, N-1 uses of this instruction are necessary.

Assume that S1, S2, and S3 are three subscript values and that D1 is the size of one dimension, D2 is the size of the second dimension, and the D1D2 is the product of D1 and D2. The following two uses of this instruction reduce the three subscripts to a single subscript.

```
CAI INDEX, S1, S2, D1      Calculates s1+(s2-1)*d1
CAI INDEX, INDEX, S3, D1D2 Calculates s1+(s2-1)*d1+(s3-1)*d2*d1
```

Exceptions

Exception		Operands 1	2	3	4	Other
06	Addressing					
	01 Spacing addressing violation	X	X	X	X	
	02 Boundary alignment	X	X	X	X	
	03 Range	X	X	X	X	
	06 Optimized addressability invalid	X	X	X	X	

Exception		Operands 1	2	3	4	Other
08	Argument/parameter					
	01 Parameter reference violation	X	X	X	X	
0C	Computation					
	0A size	X				
10	Damage encountered					
	04 System object damage state	X	X	X	X	X
	44 Partial system object damage	X	X	X	X	X
1C	Machine-dependent exception					
	03 Machine storage limit exceeded					X
20	Machine support					
	02 Machine check					X
	03 Function check					X
22	Object access					
	01 Object not found	X	X	X	X	
	02 Object destroyed	X	X	X	X	
	03 Object suspended	X	X	X	X	
	08 object compressed					X
24	Pointer specification					
	01 Pointer does not exist	X	X	X	X	
	02 Pointer type invalid	X	X	X	X	
2E	Resource control limit					
	01 user profile storage limit exceeded					X
36	Space management					
	01 space extension/truncation					X

Compute Math Function Using One Input Value (CMF1)

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand 3	Operand [4-5]
CMF1 100B		Receiver	Controls	Source	
CMF1B 1C0B	Branch options	Receiver	Controls	Source	Branch targets
CMF1I 180B	Indicator options	Receiver	Controls	Source	Indicator targets

Operand 1: Numeric variable scalar.

Operand 2: Character(2) scalar (fixed length).

Operand 3: Numeric scalar.

Operand 4-5:

- *Branch Form*-Branch point, instruction pointer, relative instruction number, or absolute instruction number.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Description: The mathematical function, indicated by the controls operand, is performed on the source operand value and the result is placed in the receiver operand.

The calculation is always done in floating-point.

The result of the operation is copied into the receiver operand.

The controls operand must be a character scalar that specifies which mathematical function is to be performed. It must be at least 2 bytes in length and has the following format:

- Controls operand Char(2)
  - Hex 0001 = Sine
  - Hex 0002 = Arc sine
  - Hex 0003 = Cosine
  - Hex 0004 = Arc cosine
  - Hex 0005 = Tangent
  - Hex 0006 = Arc tangent
  - Hex 0007 = Cotangent
  - Hex 0010 = Exponential function
  - Hex 0011 = Logarithm based e (natural logarithm)
  - Hex 0012 = Sine hyperbolic
  - Hex 0013 = Cosine hyperbolic
  - Hex 0014 = Tangent hyperbolic
  - Hex 0015 = Arc tangent hyperbolic
  - Hex 0020 = Square root
  - All other values are reserved

The controls operand mathematical functions are as follows:

- Hex 0001-Sine

  The sine of the numeric value of the source operand, whose value is considered to be in radians, is computed and placed in the receiver operand.

The result is in the range:

`-1 ≤ SIN(x) ≤ 1`

- Hex 0002-Arc sine

  The arc sine of the numeric value of the source operand is computed and the result (in radians) is placed in the receiver operand.

  The result is in the range:

  `-pi/2 ≤ ASIN(x) ≤ +pi/2`

- Hex 0003-Cosine

  The cosine of the numeric value of the source operand, whose value is considered to be in radians, is computed and placed in the receiver operand.

  The result is in the range:

  `-1 ≤ COS(x) ≤ 1`

- Hex 0004-Arc cosine

  The arc cosine of the numeric value of the source operand is computed and the result (in radians) is placed in the receiver operand.

  The result is in the range:

  `0 ≤ ACOS(x) ≤ pi`

- Hex 0005-Tangent

  The tangent of the source operand, whose value is considered to be in radians, is computed and the result is placed in the receiver operand.

  The result is in the range:

  `-infinity ≤ TAN(x) ≤ +infinity`

- Hex 0006-Arc tangent

  The arc tangent of the source operand is computed and the result (in radians) is placed in the receiver operand.

  The result is in the range:

  `-pi/2 ≤ ATAN(x) ≤ pi/2`

- Hex 0007-Cotangent

  The cotangent of the source operand, whose value is considered to be in radians, is computed and the result is placed in the receiver operand.

  The result is in the range:

  `-infinity ≤ COT(x) ≤ +infinity`

- Hex 0010-Exponential function

  The computation e power (source operand) is performed and the result is placed in the receiver operand.

  The result is in the range:

  `0 ≤ EXP(x) ≤ +infinity`

- Hex 0011-Logarithm based e (natural logarithm)

  The natural logarithm of the source operand is computed and the result is placed in the receiver operand.

  The result is in the range:

  `-infinity ≤ LN(x) ≤ +infinity`

- Hex 0012-Sine hyperbolic

  The sine hyperbolic of the numeric value of the source operand is computed and the result (in radians) is placed in the receiver operand.

  The result is in the range:

  -infinity ≤ SINH(x) ≤ +infinity

- Hex 0013-Cosine hyperbolic

  The cosine hyperbolic of the numeric value of the source operand is computed and the result (in radians) is placed in the receiver operand.

  The result is in the range:

  +1 ≤ COSH(x) ≤ +infinity

- Hex 0014-Tangent hyperbolic

  The tangent hyperbolic of the numeric value of the source operand is computed and the result (in radians) is placed in the receiver operand.

  The result is in the range:

  -1 ≤ TANH(x) ≤ +1

- Hex 0015-Arc tangent hyperbolic

  The inverse of the tangent hyperbolic of the numeric value of the source operand is computed and the result (in radians) is placed in the receiver operand.

  The result is in the range:

  -infinity ≤ ATANH(x) ≤ +infinity

- Hex 0020-Square root

  The square root of the numeric value of the source operand is computed and placed in the receiver operand.

  The result is in the range:

  0 ≤ SQRT(x) ≤ +infinity

The following chart shows some special cases for certain arguments (X) of the different mathematical functions.

X Function	Masked NaN	Unmasked NaN	+infinity	-infinity	+0	-0	Maximum Value	Minimum Value	Other
Sine	g	A(e)	A(f)	A(f)	+0	-0	A(1,f)	A(1,f)	B(3)
Arc sine	g	A(e)	A(f)	A(f)	+0	-0	A(6,f)	A(6,f)	-
Cosine	g	A(e)	A(f)	A(f)	+1	+1	A(1,f)	A(1,f)	B(3)
Arc cosine	g	A(e)	A(f)	A(f)	+pi/2	+pi/2	A(6,f)	A(6,f)	-
Tangent	g	A(e)	A(f)	A(f)	+0	-0	A(1,f)	A(1,f)	B(3)
Arc tangent	g	A(e)	+pi/2	-pi/2	+0	-0	-	-	-
Cotangent	g	A(e)	A(f)	A(f)	+inf	-inf	A(1,f)	A(1,f)	B(3)
Exponent	g	A(e)	+inf	+0	+1	+1	C(4,a)	D(5,b)	-
Logarithm	g	A(e)	+inf	A(f)	-inf	-inf	-	-	A(2,f)
Sine hyperbolic	g	A(e)	+inf	-inf	+0	-0	-	-	-
Cosine hyperbolic	g	A(e)	+inf	+inf	+1	+1	-	-	-
Tangent hyperbolic	g	A(e)	+1	-1	+0	-0	-	-	-
Arc tangent hyperbolic	g	A(e)	A(f)	A(f)	+0	-0	A(6,f)	A(6,f)	-
Square root	g	A(e)	+inf	A(f)	+0	-0	-	-	A(2,f)

Figure 2-1. Special cases for arguments of CMF1 mathematical functions.

Capital letters in the chart indicate the exceptions, small letters indicate the returned results, and Arabic numerals indicate the limits of the arguments (X) as defined in the following lists:

A = *Floating-point invalid operand* (hex 0C09) exception (no result stored if unmasked; if masked, occurrence bit is set)

B = *Floating-point inexact result* (hex 0C0D) exception (result is stored whether or not exception is masked)

C = *Floating-point overflow* (hex 0C06) exception (no result is stored if unmasked; if masked, occurrence bit is set)

D = *Floating-point underflow* (hex 0C07) exception (no result is stored if unmasked; occurrence bit is always set)

a = Result follows the rules that depend on round mode

b = Result is +0 or a denormalized value

c = Result is +infinity

d = Result is -infinity

e = Result is the masked form of the input NaN

f = Result is the system default masked NaN

g = Result is the input NaN

inf = Result is infinity

1 = | pi * 2**50 | = Hex 432921FB54442D18

2 = Argument is in the range: -inf < x < -0

3 = | pi * 2**26 | = Hex 41A921FB54442D18

4 = 1n(2**1023) Hex 40862E42FEFA39EF

5 = 1n(2**-1021.4555) = Hex C086200000000000

6 = Argument is in the range: -1 ≤ x ≤ +1

The following chart provides accuracy data for the mathematical functions that can be invoked by this instruction.

Function Name	Sample Selection			Accuracy Data			
				Relative Error (e)		Absolute Error (E)	
	A	Range of x	D	MAX(e)	SD(e)	MAX(E)	SD(E)
Arc cosine	9	0 < = x < = 3.14	U			8.26 * 10**-14	2.11 * 10**-15
Arc sine	10	-1.57 < = x < = 1.57	U	1.02 * 10**-13	2.66 * 10**-15		
Arc tangent	1	-pi/2 < x < pi/2	1			3.33 * 10**-16	9.57 * 10**-17
Arc tangent hyperbolic	14	-3 < = x < = 3	U			1.06 * 10**-14	1.79 * 10**-15
Cosine		(See Sine below)					
Cosine hyperbolic		(See Sine Hyperbolic) below)					
Cotangent	11	-10 < = x < = 100	U	4.83 * 10**-16	1.48 * 10**-16		
		.000001 < = x < = .001	U	4.36 * 10**-16	1.49 * 10**-16		
		4000 < = x < = 4000000	U	5.72 * 10**-16	1.46 * 10**-16		
Exponential	2	-100 < = x < = 300	U	5.70 * 10**-14	1.13 * 10**-14		
Natural	3	0.5 < = x < = 1.5	U			2.77 * 10**-16	8.01 * 10**-17
logarithm	4	-100 < = x < = 700	E	2.17 * 10**-16	7.37 * 10**-17		
Sine cosine	5	-10 < = x < = 100	U			2.22 * 10**-16	1.31 * 10**-16
		.000001 < = x < = .001	U			2.22 * 10**-16	1.56 * 10**-16
		4000 < = x < = 4000000	U			2.22 * 10**-16	1.28 * 10**-16
	6	-10 < = x < = 100	U			3.33 * 10**-16	8.39 * 10**-17
		.000001 < = x < = .001	U			4.33 * 10**-19	1.28 * 10**-19
		4000 < = x < = 4000000	U			3.33 * 10**-16	8.17 * 10**-17
Sine/cosine hyperbolic	12	-100 < = x < = 300	U	6.31 * 10**-16	1.97 * 10**-16		
Square root	7	-100 < = x < = 700	E	4.13 * 10**-16	1.27 * 10**-16		
Tangent	8	-10 < = x < = 100	U	4.59 * 10**-16	1.54 * 10**-16		
		.000001 < = x < = .001	U	4.42 * 10**-16	1.44 * 10**-16	3.25 * 10**-19	8.06 * 10**-20
		4000 < = x < = 4000000	U	4.77 * 10**-16	1.43 * 10**-16		
Tangent hyperbolic	13	-100 < = x < = 300	U	8.35 * 10**-16	3.87 * 10**-17	2.22 * 10**-16	3.17 * 10**-17

Figure 2-2 (Part 1 of 2). Accuracy data for CMF1 mathematical functions.

Algorithm Notes:

1. f(x) = x, and g(x) = ATAN(TAN(x)).
2. f(x) = e**x, and g(x) = e**(1n(e**x)).
3. f(x) = 1n(x), and g(x) = 1n(e**(1n(x))).
4. f(x) = x, and g(x) = 1n(e**x).
5. Sum of squares algorithm. f(x) = 1, and g(x) = SIN(x))**2 + (COS(x))**2.
6. Double angle algorithm. f(x) - SIN(2x), and g(x) = 2*(SIN(x)*COS(x)).
7. f(x) = e(**x, and g(x) = (SQR(e**x))**2.
8. f(x) = TAN(x), and g(x) = SIN(x) / COS(x).
9. f(x) = x, and g(x) = ACOS(COS(x)).
10. f(x) = x, and g(x) = ASIN(SIN(x)).
11. f(x) = COT(x), and g(x) = COS(x) / SIN(x).
12. f(x) = SINH(2x), and g(x) = 2*(SINH(x)*COSH(x)).
13. f(x) = TANH(x), and g(x) = SINH(x) / COSH(x).
14. f(x) = x, and g(x) = ATANH(TANH(x)).

Distribution Note: The sample input arguments were tangents of numbers, x, uniformly distributed between -pi/2 and +pi/2.

Figure 2-2 (Part 2 of 2). Accuracy data for CMF1 mathematical functions.

The vertical columns in the accuracy data chart have the following meanings:

- *Function Name*: This column identifies the principal mathematical functions evaluated with entries arranged in alphabetical order by function name.
- *Sample Selection*: This column identifies the selection of samples taken for a particular math function through the following subcolumns:
  - *A*: identifies the algorithm used against the argument, x, to gather the accuracy samples. The numbers in this column refer to notes describing the functions, f(x) and g(x), which were calculated to test for the anticipated relation where f(x) should equal g(x). An accuracy sample then, is an evaluation of the degree to which this relation held true. The algorithm used to sample the arctangent function, for example, defines g(x) to first calculate the tangent of x to provide an appropriate distribution of input arguments for the arctangent function. Since f(x) is defined simply as the value of x, the relation to be evaluated is then x=ARCTAN(TAN(x)). This type of algorithm, where a function and its inverse are used in tandem, is the usual type employed to provide the appropriate comparison values for the evaluation.
  - "Range of x": gives the range of x used to obtain the accuracy samples. The test values for x are uniformly distributed over this range. It should be noted that x is not always the direct input argument to the function being tested; it is sometimes desirable to distribute the input arguments in a nonuniform fashion to provide a more complete test of the function (see column D below). For each function, accuracy data is given for one or more segments within the valid range of x. In each case, the numbers given are the most meaningful to the function and range under consideration.
  - *D*: identifies the distribution of arguments input to the particular function being sampled. The letter E indicates an exponential distribution. The letter U indicates a uniform distribution. A number refers to a note providing detailed information regarding the distribution.
- *Accuracy Data*: The maximum relative error and standard deviation of the relative error are generally useful and revealing statistics; however, they are useless for the range of a function where its value becomes zero. This is because the slightest error in the argument can cause an unpredictable fluctuation in the magnitude of the answer. When a small argument error would have this effect, the maximum absolute error and standard deviation of the absolute error are given for the range.

- *Relative Error (e)*: The maximum relative error and standard deviation (root mean square) of the relative error are defined:

  MAX(e) = MAX(ABS((f(x) - g(x)) / f(x)))

  where: MAX selects the largest of its arguments and ABS takes the absolute value of its argument.

  SD(e) = SQR((1/N) SUMSQ((f(x) - g(x)) / f(x)))

  where: SQR takes the square root of its argument and SUMSQ takes the summation of the squares of its arguments over all of the test cases.

- *Absolute Error (E)*: The maximum absolute error produced during the testing and the standard deviation (root mean square) of the absolute error are:

  MAX(E) = MAX(ABS(f(x) - g(x)))

  where: the operators are those defined above.

  SD(E) = SQR((1/N) SUMSQ(f(x) - g(x)))

  where: the operators are those defined above.

Limitations: The following are limits that apply to the functions performed by this instruction.

The source and receiver operands must both be specified as floating-point with the same length (4 bytes for short format or 8 bytes for long format).

Resultant Conditions

- Positive-The algebraic value of the receiver operand is positive.
- Negative-The algebraic value of the receiver operand is negative.
- Zero-The algebraic value of the receiver operand is zero.
- Unordered-The value assigned to the floating-point result is NaN.

Exceptions

Exception		Operands 1	2	3	Other
06	Addressing				
	01 Spacing addressing violation	X	X	X	
	02 Boundary alignment violation	X	X	X	
	03 Range	X	X	X	
	06 Optimized addressability invalid	X	X	X	
08	Argument/parameter				
	01 Parameter reference violation	X	X	X	
0C	Computation				
	06 Floating-point overflow	X			
	07 Floating-point underflow	X			
	09 Floating-point invalid operand			X	
	0D Floating-point inexact result	X			
10	Damage encountered				
	04 System object damage state				X

Exception		Operands 1	2	3	Other
	44 Partial system object damage				X
1C	Machine-dependent exception				
	03 Machine storage limit exceeded				X
20	Machine support				
	02 Machine check				X
	03 Function check				X
22	Object access				
	01 Object not found	X	X	X	
	02 Object destroyed	X	X	X	
	03 Object suspended	X	X	X	
	08 object compressed				X
24	Pointer specification				
	01 Pointer does not exist	X	X	X	
	02 Pointer type invalid	X	X	X	
2C	Program execution				
	04 Invalid branch target				X
2E	Resource control limit				
	01 user profile storage limit exceeded				X
	02 Process storage limit exceeded				X
32	Scalar specification				
	01 Scalar type invalid	X	X	X	
	03 Scalar value invalid		X		
36	Space management				
	01 space extension/truncation				X

Compute Math Function Using Two Input Values (CMF2)

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand 3	Operand 4	Operand [5-8]
CMF2 100C		Receiver	Controls	Source 1	Source 2	
CMF2B 1C0C	Branch options	Receiver	Controls	Source 1	Source 2	Branch targets
CMF2I 180C	Indicator options	Receiver	Controls	Source 1	Source 2	Indicator targets

Operand 1: Numeric variable scalar.

Operand 2: Character(2) scalar (fixed length)

Operand 3: Numeric scalar.

Operand 4: Numeric scalar.

Operand 5-8:

- *Branch Form*-Branch point, instruction pointer, relative instruction number, or absolute instruction number.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Description: The mathematical function, indicated by the controls operand, is performed on the source operand values and the result is placed in the receiver operand.

The calculation is always done in floating-point.

The controls operand must be a character scalar that specifies which mathematical function is to be performed. It must be at least 2 bytes in length and have the following format:

- Controls operand Char(2)
  - Hex 0001 = Power (x to the y)
  - All other values are reserved

The computation x power y, where x is the first source operand and y is the second source operand, is performed and the result is placed in the receiver operand.

The following chart shows some special cases for certain arguments of the power function (x**y). Within the chart, the capitalized letters X and Y refer to the absolute value of the arguments x and y; that is, X = |x| and Y = |y|.

x \ y	-inf y=2n+1	y<0, y=2n	y<0 real	y<0	-1	-1/2	+0 or -0	+1/2	+1	y>0 y=2n+1	y>0 y=2n	y>0 real	+inf	M-NaN	UnM-NaN
+inf	+0	+0	+0	+0	+0	+1	+inf	+inf	+inf	+inf	+inf	+inf	b	A(c)	
x>1	+0	+1 / x**Y	+1 / x**Y	+1 / x**Y	+1 / x	+1 / SQRT(x)	+1	SQRT(x)	x	x**y	x**y	x**y	+inf	b	A(c)
x=+1	+1	+1	+1	+1	+1	+1	+1	+1	+1	+1	+1	+1	+1	b	A(c)
0<x<1	+inf	+1 / x**Y	+1 / x**Y	+1 / x**Y	+1 / x	+1 / SQRT(x)	+1	SQRT(x)	x	x**y	x**y	x**y	+0	b	A(c)
x=+0	E(f)	E(f)	E(f)	E(f)	E(f)	E(f)	+1	+0	+0	+0	+0	+0	+0	b	A(c)
x=-0	E(f)	E(g)	E(f)	E(f)	E(g)	E(g)	+1	-0	-0	-0	+0	+0	+0	b	A(c)
0>x>-1	A(a)	-1 / X**Y	+1 / X**Y	A(a)	-1 / X	A(a)	+1	A(a)	x	-X**y	X**y	A(a)	A(a)	b	A(c)
x=-1	A(a)	-1	+1	A(a)	-1	A(a)	+1	A(a)	-1	-1	+1	A(a)	A(a)	b	A(c)
x<-1	A(a)	-1 / X**Y	+1 / X**Y	A(a)	-1 / X	A(a)	+1	A(a)	x	-X**y	X**y	A(a)	A(a)	b	A(c)
x=-inf	A(a)	-0	+0	A(a)	-0	A(a)	+1	A(a)	-inf	-inf	+inf	A(a)	A(a)	b	A(c)
Masked NaN	b	b	b	b	b	b	b	b	b	b	b	b	b	d	A(e)
Unmasked NaN	A(c)	A(c)	A(c)	A(c)	A(c)	A(c)	A(c)	A(c)	A(c)	A(c)	A(c)	A(c)	A(c)	A(e)	A(e)

*Figure 2-3. Special cases of the power function (x**y)*

Capital letters in the chart indicate the exceptions and small letters indicate the returned results as defined in the following list:

A *Floating-point invalid operand* (hex 0C09) exception

E *Divide by zero* (hex 0C0E) exception

a Result is the system default masked NaN

b Result is the same NaN

c Result is the same NaN masked

d Result is the larger NaN

e Result is the larger NAN masked

f Result is +infinity

g Result is -infinity

The following chart provides accuracy data for the mathematical function that can be invoked by this instruction.

Function Name	Sample Selection		Accuracy Data	
	x	y	MAX(e)	SD(e)
Power	1/3	-345 < = y < = 330	4.99 * 10**-16	1.90 * 10**-16
	.75	-1320 < = y < = 1320	2.96 * 10**-16	2.39 * 10**-16
	.9	-3605 < = y < = 3605	1.23 * 10**-16	1.02 * 10**-16
	10	-165 < = y < = 165	7.10 * 10**-16	3.18 * 10**-16
	712	-57 < = y < = 57	1.75 * 10**-15	7.24 * 10**-16

Figure 2-4. Accuracy data for CMF2 mathematical functions.

The vertical columns in the accuracy data chart have the following meanings:

- *Function Name*: This column identifies the mathematical function.
- *Sample Selection*: This column identifies the selection of samples taken for the power function. The algorithm used against the arguments, x and y, to gather the accuracy samples was a test for the anticipated relation where f(x) should equal g(x,y):

  where:

  ```
  f (x)= x
  g(x,y)= (x**y)**(1/y)
  ```

  An accuracy sample then, is an evaluation of the degree to which this relation held true.

  The range of argument values for x and y were selected such that x was held constant at a particular value and y was uniformly varied throughout a range of values which avoided overflowing or underflowing the result field. The particular values selected are indicated in the subcolumns entitled x and y.

- *Accuracy Data*: The maximum relative error and standard deviation (root mean square) of the relative error are generally useful and revealing statistics. These statistics for the relative error, (e), are provided in the following subcolumns:

  MAX(e) = MAX(ABS((f(x) - g(x)) / f(x)))

  where: MAX selects the largest of its arguments and ABS takes the absolute value of its argument.

  SD(e) = SQR((1/N) SUMSQ((f(x) - g(x)) / f(x)))

  where: SQR takes the square root of its argument and SUMSQ takes the summation of the squares of its arguments over all of the test cases.

Limitations: The following are limits that apply to the functions performed by this instruction.

The source and receiver operands must both be specified as floating-point with the same length (4 bytes for short format or 8 bytes for long format).

Resultant Conditions

- Positive-The algebraic value of the receiver operand is positive.
- Negative-The algebraic value of the receiver operand is negative.
- Zero-The algebraic value of the receiver operand is zero.
- Unordered-The value assigned to the floating-point result is NaN.

Exceptions

Exception		Operands 1	2	3	4	Other
06	Addressing					
	01 space addressing violation	X	X	X	X	
	02 boundary alignment violation	X	X	X	X	
	03 range	X	X	X	X	
	06 optimized addressability invalid	X	X	X	X	
08	Argument/parameter					
	01 parameter reference violation	X	X	X	X	
0C	Computation					
	06 floating-point overflow	X				
	07 floating-point underflow	X				
	09 floating-point invalid operand			X	X	
	0C invalid floating-point conversion	X				
	0D floating-point inexact result	X				
	0E floating-point zero divide	X				
10	Damage encountered					
	04 System object damage state					X
	44 partial system object damage					X
1C	Machine-dependent exception					
	03 machine storage limit exceeded					X
20	Machine support					
	02 machine check					X
	03 function check					X
22	Object access					
	01 object not found	X	X	X	X	
	02 object destroyed	X	X	X	X	
	03 object suspended	X	X	X	X	
	08 object compressed					X
24	Pointer specification					
	01 pointer does not exist	X	X	X	X	
	02 pointer type invalid	X	X	X	X	
2C	Program execution					
	04 invalid branch target					X
2E	Resource control limit					
	01 user profile storage limit exceeded					X
	02 process storage limit exceeded					X
32	Scalar specification					
	01 scalar type invalid	X	X	X	X	

Exception		Operands 1	2	3	4	Other
	03 scalar value invalid		X			
36	Space management					
	01 space extension/truncation					X

Concatenate (CAT)

Op Code (Hex)	Operand 1	Operand 2	Operand 3
10F3	Receiver	Source 1	Source 2

Operand 1: Character variable scalar.

Operand 2: Character scalar.

Operand 3: Character scalar.

Description: The character string value of the second source operand is joined to the right end of the character string value of the first source operand. The resulting string value is placed (left-adjusted) in the receiver operand.

The length of the operation is equal to the length of the receiver operand with the resulting string truncated or is logically padded on the right end accordingly. The pad value for this instruction is hex 40.

Substring operand references that allow for a null substring reference (a length value of zero) may be specified for operands 1, 2, and 3. The effect of specifying a null substring reference for one source operand is that the other source operand is used as the result of the concatenation. The effect of specifying a null substring reference for both source operands is that the bytes of the receiver are each set with a value of hex 40. The effect of specifying a null substring reference for the receiver is that a result is not set regardless of the value of the source operands.

Exceptions

Exception		Operands 1	2	3	Other
06	Addressing				
	01 space addressing violation	X	X	X	
	02 boundary alignment	X	X	X	
	03 range	X	X	X	
	06 optimized addressability invalid	X	X	X	
08	Argument/parameter				
	01 parameter reference violation				X
10	Damage encountered				
	04 system object damage state	X	X	X	X
	44 partial system object damage	X	X	X	X
1C	Machine-dependent exception				
	03 machine storage limit exceeded				X
20	Machine support				
	02 machine check				X
	03 function check				X
22	Object access				
	01 object not found	X	X	X	
	02 object destroyed	X	X	X	
	03 object suspended	X	X	X	

Exception		Operands 1	2	3	Other
	08 object compressed				X
24	Pointer specification				
	01 pointer does not exist	X	X	X	
	02 pointer type invalid	X	X	X	
2E	Resource control limit				
	01 user profile storage limit exceeded				X
36	Space management				
	01 space extension/truncation				X

Convert BSC to Character (CVTBC)

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand 3	Operand [4-6]
CVTBC 10AF		Receiver	Controls	Source	
CVTBCB 1CAF	Branch options	Receiver	Controls	Source	Branch targets
CVTBCI 18AF	Indicator options	Receiver	Controls	Source	Indicator targets

Operand 1: Character variable scalar.

Operand 2: Character(3) variable scalar (fixed-length).

Operand 3: Character scalar.

Operand 4-6:

- *Branch Form*-Branch point, instruction pointer, relative instruction number, or absolute instruction number.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

ILE access

```
CVTBC (
  var receiver          : aggregate;
      receiver_length   : unsigned binary;
  var controls          : aggregate;
  var source            : aggregate;
      source_length     : unsigned binary;
  var return_code       : signed binary
)
```

The return_code will be set as follows:

Return_Code	Meaning
-1	Completed Record.
0	Source Exhausted.
1	Truncated Record.

Description: This instruction converts a string value from the BSC (binary synchronous communications) compressed format to a character string. The operation converts the source (operand 3) from the BSC compressed format to character under control of the controls (operand 2) and places the result into the receiver (operand 1).

The source and receiver operands must both be character strings.

The controls operand must be a character scalar that specifies additional information to be used to control the conversion operation. It must be at least 3 bytes in length and have the following format:

- Controls operand — Char(3)
  - Source offset — Bin(2)

— Record separator Char(1)

The **source offset** specifies the offset where bytes are to be accessed from the source operand. If the *source offset* is equal to or greater than the length specified for the source operand (it identifies a byte beyond the end of the source operand), a *template value invalid* (hex 3801) exception is signaled. As output from the instruction, the *source offset* is set to specify the offset that indicates how much of the source is processed when the instruction ends.

The **record separator**, if specified with a value other than hex 01, contains the value used to separate converted records in the source operand. A value of hex 01 specifies that record separators do not occur in the converted records in the source.

Only the first 3 bytes of the controls operand are used. Any excess bytes are ignored.

The operation begins by accessing the bytes of the source operand located at the offset specified in the source offset. This is assumed to be the start of a record. The bytes of the record in the source operand are converted into the receiver record according to the following algorithm.

The strings to be built in the receiver are contained in the source as blank compression entries and strings of consecutive nonblank characters.

The format of the blank compression entries occurring in the source are as follows:

- Blank compression entry Char(2)
  - Interchange group separator Char(1)
  - Count of compressed blanks Char(1)

The **interchange group separator** has a fixed value of hex 1D.

The **count of compressed blanks** provides for describing up to 63 compressed blanks. The count of the number of blanks (up to 63) to be decompressed is formed by subtracting hex 40 from the value of the count field. The count field can vary from a value of hex 41 to hex 7F. If the count field contains a value outside of this range, a *conversion* (hex 0C01) exception is signaled.

Strings of blanks described by blank compression entries in the source are repeated in the receiver the number of times specified by the blank compression count.

Nonblank strings in the source are copied into the receiver intact with no alteration.

If the receiver record is filled with converted data without encountering the end of the source operand, the instruction ends with a resultant condition of *completed record*. This can occur in two ways. If a *record separator* was not specified, the instruction ends when enough bytes have been converted from the source to fill the receiver. If a *record separator* was specified, the instruction ends when a source byte is encountered with that value prior to or just after filling the receiver record. The *source offset* value locates the byte following the last source record (including the *record separator*) for which conversion was completed. When the *record separator* value is encountered, any remaining bytes in the receiver are padded with blanks.

If the end of the source operand is encountered (whether or not in conjunction with a *record separator* or the filling of the receiver), the instruction ends with a resultant condition of *source exhausted*. The *source offset* value locates the byte following the last byte of the source operand. The remaining bytes in the receiver after the converted record are padded with blanks.

If the converted form of a record cannot be completely contained in the receiver, the instruction ends with a resultant condition of *truncated record*. The offset value for the source locates the byte following the last source byte for which conversion was performed, unless a blank compression entry

was being processed. In this case, the source offset is set to locate the byte after the blank compression entry. If the source does not contain record separators, this condition can only occur for the case in which a blank-compression entry was being converted when the receiver record became full.

Any form of overlap between the operands on this instruction yields unpredictable results in the receiver operand.

Resultant Conditions

- Completed record-The receiver record has been completely filled with converted data from a source record.
- Source exhausted-All of the bytes in the source operand have been converted into the receiver operand.
- Truncated record-The receiver record cannot contain all of the converted data from the source record.

Exceptions

Exception		Operands 1	2	3	Other
06	Addressing				
	01 space addressing violation	X	X	X	
	02 boundary alignment violation	X	X	X	
	03 range	X	X	X	
	06 optimized addressability invalid	X	X	X	
08	Argument/parameter				
	01 parameter reference violation	X	X	X	
0C	Computation				
	01 conversion			X	
10	Damage encountered				
	04 System object damage state				X
	44 partial system object damage				X
1C	Machine-dependent exception				
	03 machine storage limit exceeded				X
20	Machine support				
	02 machine check				X
	03 function check				X
22	Object access				
	01 object not found	X	X	X	
	02 object destroyed	X	X	X	
	03 object suspended	X	X	X	
	08 object compressed				X
24	Pointer specification				
	01 pointer does not exist	X	X	X	
	02 pointer type invalid	X	X	X	

Exception		Operands 1	2	3	Other
2C	Program execution				
	04 invalid branch target				X
2E	Resource control limit				
	01 user profile storage limit exceeded				X
32	Scalar specification				
	01 scalar type invalid	X	X	X	
36	Space management				
	01 space extension/truncation				X
38	Template specification				
	01 template value invalid		X		

Convert Character to BSC (CVTCB)

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand 3	Operand [4-5]
CVTCB 108F		Receiver	Controls	Source	
CVTCBB 1C8F	Branch options	Receiver	Controls	Source	Branch targets
CVTCB 188F	Indicator options	Receiver	Controls	Source	Indicator targets

Operand 1: Character variable scalar.

Operand 2: Character(3) variable scalar (fixed-length).

Operand 3: Character scalar.

Operand 4-5:

- *Branch Form*-Branch point, instruction pointer, relative instruction number, or absolute instruction number.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

ILE access

```
CVTCB (
  var receiver          : aggregate;
      receiver_length   : unsigned binary;
  var controls          : aggregate;
  var source            : aggregate;
      source_length     : unsigned binary;
  var return_code       : signed binary
)
```

The return_code will be set as follows:

Return_Code	Meaning
-1	Receiver Overrun.
0	Source Exhausted.

Description: This instruction converts a string value from character to BSC (binary synchronous communications) compressed format. The operation converts the source (operand 3) from character to the BSC compressed format under control of the controls (operand 2) and places the result into the receiver (operand 1).

The source and receiver operands must both be character strings.

The controls operand must be a character scalar that specifies additional information to be used to control the conversion operation. It must be at least 3 bytes in length and have the following format:

- Controls operand — Char(3)
  - Receiver offset — Bin(2)
  - Record separator — Char(1)

The **receiver offset** specifies the offset where bytes are to be placed into the receiver operand. If the *receiver offset* is equal to or greater than the length specified for the receiver operand (it identifies a byte beyond the end of the receiver), a *template value invalid* (hex 3801) exception is signaled. As output from the instruction, the *receiver offset* is set to specify the offset that indicates how much of the receiver has been filled when the instruction ends.

The **record separator**, if specified with a value other than hex 01, contains the value used to separate converted records in the receiver operand. A value of hex 01 specifies that record separators are not to be placed into the receiver to separate converted records.

Only the first 3 bytes of the controls operand are used. Any excess bytes are ignored.

The source operand is assumed to be one record. The bytes of the record in the source operand are converted into the receiver operand at the location specified in the *receiver offset* according to the following algorithm.

The bytes of the source record are interrogated to identify the strings of consecutive blank (hex 40) characters and the strings of consecutive nonblank characters which occur in the source record. Only three or more blank characters are treated as a blank string for purposes of conversion into the receiver.

As the blank and nonblank strings are encountered in the source, they are packaged into the receiver.

Blank strings are reflected in the receiver as one or more blank compression entries. The format of the blank compression entries built into the receiver are as follows:

- Blank compression entry — Char(2)
  - Interchange group separator — Char(1)
  - Count of compressed blanks — Char(1)

The **interchange group separator** has a fixed value of hex 1D.

The **count of compressed blanks** provides for compressing up to 63 blanks. The value of the count field is formed by adding hex 40 to the actual number of blanks (up to 63) to be compressed. The count field can vary from a value of hex 43 to hex 7F.

Nonblank strings are copied into the receiver intact with no alteration or additional control information.

When the end of the source record is encountered, the *record separator* value if specified is placed into the receiver and the instruction ends with a resultant condition of *source exhausted*. The *receiver offset* value locates the byte following the converted record in the receiver. The value of the remaining bytes in the receiver after the converted record is unpredictable.

If the converted form of a record cannot be completely contained in the receiver (including the *record separator* if specified), the instruction ends with a resultant condition of *receiver overrun*. The *receiver offset* remains unchanged. The remaining bytes in the receiver, starting with the byte located by the receiver offset, are unpredictable.

Any form of overlap between the operands on this instruction yields unpredictable results in the receiver operand.

Resultant Conditions

- Source exhausted-All of the bytes in the source operand have been converted into the receiver operand.
- Receiver overrun-An overrun condition in the receiver operand was detected before all of the bytes in the source operand were processed.

Exceptions

Exception		Operands 1	2	3	Other
06	Addressing				
	01 space addressing violation	X	X	X	
	02 boundary alignment violation	X	X	X	
	03 range	X	X	X	
	06 optimized addressability invalid	X	X	X	
08	Argument/parameter				
	01 parameter reference violation	X	X	X	
10	Damage encountered				
	04 System object damage state				X
	44 partial system object damage				X
1C	Machine-dependent exception				
	03 machine storage limit exceeded				X
20	Machine support				
	02 machine check				X
	03 function check				X
22	Object access				
	01 object not found	X	X	X	
	02 object destroyed	X	X	X	
	03 object suspended	X	X	X	
	08 object compressed				X
24	Pointer specification				
	01 pointer does not exist	X	X	X	
	02 pointer type invalid	X	X	X	
2C	Program execution				
	04 invalid branch target				X
2E	Resource control limit				
	01 user profile storage limit exceeded				X
32	Scalar specification				
	01 scalar type invalid	X	X	X	
36	Space management				
	01 space extension/truncation				X
38	Template specification				
	01 template value invalid		X		

Convert Character to Hex (CVTCH)

Op Code (Hex)	Operand 1	Operand 2
1082	Receiver	Source

Operand 1: Character variable scalar.

Operand 2: Character variable scalar.

Description: Each character (8-bit value) of the string value in the source operand is converted to a hex digit (4-bit value) and placed in the receiver operand. The source operand characters must relate to valid hex digits or a *conversion* (hex 0C01) exception is signaled.

Characters	Hex Digits
Hex F0-hex F9	Hex 0-hex 9
Hex C1-hex C6	Hex A-hex F

The operation begins with the two operands left-adjusted and proceeds left to right until all the hex digits of the receiver operand have been filled. If the source operand is too small, it is logically padded on the right with zero characters (hex F0). If the source operand is too large, a *length conformance* (hex 0C08) or an *invalid operand length* (hex 2A0A) exception is signaled.

Substring operand references that allow for a null substring reference (a length value of zero) may be specified for operands 1 and 2. The effect of specifying a null substring reference for the source is that the bytes of the receiver are each set with a value of hex 00. The effect of specifying a null substring reference for the receiver is that no result is set.

Exceptions

Exception		Operands 1	Operands 2	Other
06	Addressing			
	01 space addressing violation	X	X	
	02 boundary alignment	X	X	
	03 range	X	X	
	06 optimized addressability invalid	X	X	
08	Argument/parameter			
	01 parameter reference violation	X	X	
0C	Computation			
	01 conversion		X	
	08 length conformance	X		
10	Damage encountered			
	04 system object damage	X	X	X
	44 partial system object damage	X	X	X
1C	Machine-dependent exception			
	03 machine storage limit exceeded			X
20	Machine support			
	02 machine check			X

Exception		Operands 1	2	Other
	03 function check			X
22	Object access			
	01 object not found	X	X	
	02 object destroyed	X	X	
	03 object suspended	X	X	
	08 object compressed			X
24	Pointer specification			
	01 pointer does not exist	X	X	
	02 pointer type invalid	X	X	
2E	Resource control limit			
	01 user profile storage limit exceeded			X
36	Space management			
	01 space extension/truncation			X

Convert Character to MRJE (CVTCM)

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand 3	Operand [4-5]
CVTCM 108B		Receiver	Controls	Source	
CVTCMB 1C8B	Branch options	Receiver	Controls	Source	Branch targets
CVTCMI 188B	Indicator options	Receiver	Controls	Source	Indicator targets

Operand 1: Character variable scalar.

Operand 2: Character(13) variable scalar (fixed-length).

Operand 3: Character scalar.

Operand 4-5:

- *Branch Form*-Branch point, instruction pointer, relative instruction number, or absolute instruction number.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

ILE access

```
CVTCM (
  var receiver        : aggregate;
      receiver_length : unsigned binary;
  var controls        : aggregate;
  var source          : aggregate;
      source_length   : unsigned binary;
  var return_code     : signed binary
)
```

The return_code will be set as follows:

Return_Code	Meaning
-1	Receiver Overrun.
0	Source Exhausted.

Description: This instruction converts a string of characters to MRJE (MULTI-LEAVING remote job entry) compressed format. The operation converts the source (operand 3) from character to the MRJE compressed format under control of the controls (operand 2) and places the results in the receiver (operand 1).

The source and receiver operands must both be character strings. The source operand cannot be specified as either a signed or unsigned immediate value.

The source operand can be described through the controls operand as being composed of one or more fixed length data fields, which may be separated by fixed length gaps of characters to be ignored during the conversion operation. Additionally, the controls operand specifies the amount of data to be processed from the source to produce a converted record in the receiver. This may be a different value than the length of the data fields in the source. The following diagram shows this structure for the source operand.

Actual Source Operand Bytes

Data Processed as Source Records

The controls operand must be a character scalar that specifies additional information to be used to control the conversion operation. It must be at least 13 bytes in length and have the following format:

- Controls operand — Char(13)
  - Receiver offset — Bin(2)
  - Source offset — Bin(2)
  - Algorithm modifier — Char(1)
  - Source record length — Char(1)
  - Data field length — Bin(2)
  - Gap offset — Bin(2)
  - Gap length — Bin(2)
  - Record control block (RCB) value — Char(1)

As input to the instruction, the **source offset** and **iver offset** fields specify the offsets where bytes of the source and receiver operands are to be processed. If an offset is equal to or greater than the length specified for the operand it corresponds to (i.e. it identifies a byte beyond the end of the operand), a *template value invalid* (hex 3801) exception is signaled. As output from the instruction, the *source offset* and *receiver offset* fields specify offsets that indicate how much of the operation is complete when the instruction ends.

The **algorithm modifier** has the following valid values:

- Hex 00 = Perform full compression.
- Hex 01 = Perform only truncation of trailing blanks.

The **source record length** value specifies the amount of data from the source to be processed. If a *source record length* of 0 is specified, a *template value invalid* (hex 3801) exception is signaled.

The **data field length** value specifies the length of the data fields in the source. Data fields occurring in the source may be separated by gaps of characters, which are to be ignored during the conversion operation. Specification of a *data field length* of 0 indicates that the source operand is one data field. In this case, the *gap length* and *gap offset* values have no meaning and are ignored.

The **gap offset** value specifies the offset to the next gap in the source. This value is both input to and output from the instruction. This is relative to the current byte to be processed in the source as located by the *source offset* field. No validation is done for this offset. It is assumed to be valid relative to the source operand. The *gap offset* value is ignored if the *data field length* is specified with a value of 0.

The **gap length** value specifies the amount of data occurring between data fields in the source operand which is to be ignored during the conversion operation. The *gap length* value is ignored if the *data field length* is specified with a value of 0.

The **record control block (RCB) value** field specifies the RCB value that is to precede the converted form of each record in the receiver. It can have any value.

Only the first 13 bytes of the controls operand are used. Any excess bytes are ignored.

The operation begins by accessing the bytes of the source operand at the location specified by the *source offset*. This is assumed to be the start of a source record. Only the bytes of the data fields in the source are accessed for conversion purposes. Gaps between data fields are ignored, causing the access of data field bytes to occur as if the data fields were contiguous with one another. Bytes accessed from the source for the source record length are considered a source record for the conversion operation. They are converted into the receiver operand at the location specified by the *receiver offset* according to the following algorithm.

The *RCB value* is placed into the first byte of the receiver record.

An SRCB (sub record control byte) value of hex 80 is placed into the second byte of the receiver record.

If the *algorithm modifier* specifies *full compression* (a value of hex 00) then:

The bytes of the source record are interrogated to locate the blank character strings (2 or more consecutive blanks), identical character strings (3 or more consecutive identical characters), and nonidentical character strings occurring in the source. A blank character string occurring at the end of the record is treated as a special case (see following information on trailing blanks).

If the *algorithm modifier* specifies *blank truncation* (a value of hex 01) then:

The bytes of the source record are interrogated to determine if a blank character string exists at the end of the source record. If one exists, it is treated as a string of trailing blanks. All characters prior to it in the record are treated as one string of nonidentical characters.

The strings encountered (blank, identical, or nonidentical) are reflected in the receiver by building one or more SCBs (string control bytes) in the receiver to describe them.

The format of the SCBs built into the receiver is:

- SCB format is `o k l jjjjj`

The bit meanings are:

Bit	Value	Meaning
o	0	End of record; the EOR SCB is hex 00.
	1	All other SCBs.
k	0	The string is compressed.
	1	The string is not compressed.
l		For k = 0:
	0	Blanks (hex 40s) have been deleted.
	1	Nonblank characters have been deleted. The next character in the data stream is the specimen character.
		For k = 1:
		This bit is part of the length field for length of uncompressed data.
jjjjj		Number of characters that have been deleted if k = 0. The value can be 2-31.
ljjjjj		Number of characters to the next SCB (no compression) if k = 1. The value can be 1-63. The uncompressed (nonidentical bytes) follow the SCB in the data stream.

When the end of a source record is encountered, an EOR (end of record) SCB (hex 00) is built into the receiver. Trailing blanks in a record including a record of all blanks are represented in the receiver by an EOR character. However, a record of all blanks is reflected in the compressed result by an RCB, an SRCB, a compression entry describing an 'unlike string' of one blank character, and an EOR character.

Additionally, the *receiver offset*, the *source offset*, and the *gap offset* are updated in the controls operand.

If the end of the source operand is not encountered, the operation then continues by reapplying the above algorithm to the next record in the source operand.

If the end of the source operand is encountered (whether or not in conjunction with a record boundary), the instruction ends with a resultant condition of *source exhausted*. The *source offset* locates the byte following the last source record for which conversion was completed. The *gap offset* value indicates the offset to the next gap relative to the source offset value set for this condition. The *gap offset* value has no meaning and is not set when the *data field length* is 0. The *receiver offset* locates the byte following the last fully converted record in the receiver. The value of the remaining bytes in the receiver after the last converted record is unpredictable.

If the converted form of a record cannot be completely contained in the receiver, the instruction ends with a resultant condition of *receiver overrun*. The *source offset* locates the byte following the last source record for which conversion was completed. The *gap offset* value indicates the offset to the next gap relative to the source offset value set for this condition. The *gap offset* value has no meaning and is not set when the *data field length* is 0. The *receiver offset* locates the byte following the last fully converted record in the receiver. The value of the remaining bytes in the receiver after the last converted record is unpredictable.

Any form of overlap between the operands of this instruction yields unpredictable results in the receiver operand.

Resultant Conditions

- Source exhausted-All complete records in the source operand have been converted into the receiver operand.
- Receiver overrun-An overrun condition in the receiver operand was detected prior to processing all of the bytes in the source operand.

If *source exhausted* and *receiver overrun* occur at the same time, the *source exhausted* condition is recognized first. When *source exhausted* is the resultant condition, the receiver may also be full. In this case, the *receiver offset* may contain a value equal to the length specified for the receiver, and this condition will cause an exception on the next invocation of the instruction. The processing performed for the *source exhausted* condition provides for this case when the instruction is invoked multiple times with the same controls operand template. When the *receiver overrun* condition is the resultant condition, the source always contains data that can be converted.

Exceptions

Exception		Operands 1	2	3	Other
06	Addressing				
	01 space addressing violation	X	X	X	
	02 boundary alignment violation	X	X	X	
	03 range	X	X	X	
	06 optimized addressability invalid	X	X	X	

Exception		Operands 1	2	3	Other
08	Argument/parameter				
	01 parameter reference violation	X	X	X	
10	Damage encountered				
	04 System object damage state				X
	44 partial system object damage				X
1C	Machine-dependent exception				
	03 machine storage limit exceeded				X
20	Machine support				
	02 machine check				X
	03 function check				X
22	Object access				
	01 object not found	X	X	X	
	02 object destroyed	X	X	X	
	03 object suspended	X	X	X	
	08 object compressed				X
24	Pointer specification				
	01 pointer does not exist	X	X	X	
	02 pointer type invalid	X	X	X	
2C	Program execution				
	04 invalid branch target				X
2E	Resource control limit				
	01 user profile storage limit exceeded				X
32	Scalar specification				
	01 scalar type invalid	X	X	X	
36	Space management				
	01 space extension/truncation				X
38	Template specification				
	01 template value invalid			X	

Convert Character to Numeric (CVTCN)

Op Code (Hex)	Operand 1	Operand 2	Operand 3
1083	Receiver	Source	Attributes

Operand 1: Numeric variable scalar or data-pointer-defined numeric scalar.

Operand 2: Character scalar or data-pointer-defined character scalar.

Operand 3: Character(7) scalar or data-pointer-defined character scalar.

Description: The character scalar specified by operand 2 is treated as though it were a numeric scalar with the attributes specified by operand 3. The character string source operand is converted to the numeric forms of the receiver operand and moved to the receiver operand. The value of operand 2, when viewed in this manner, is converted to the type, length, and precision of the numeric receiver, operand 1, following the rules for the Copy Numeric Value instruction.

The length of operand 2 must be large enough to contain the numeric value described by operand 3. If it is not large enough, a *scalar value invalid* (hex 3203) exception is signaled. If it is larger than needed, its leftmost bytes are used as the value, and the rightmost bytes are ignored.

Normal rules of arithmetic conversion apply except for the following. If operand 2 is interpreted as a zoned decimal value, a value of hex 40 in the rightmost byte referenced in the conversion is treated as a positive sign and a zero digit.

If a decimal to binary conversion causes a *size* (hex 0C0A) exception to be signaled, the binary value contains the correct truncated result only if the decimal value contains 15 or fewer significant nonfractional digits.

The format of the attribute operand specified by operand 3 is as follows:

- Scalar attributes — Char(7)
  - Scalar type — Char(1)

    Hex 00 = Signed binary
    Hex 01 = Floating-point
    Hex 02 = Zoned decimal
    Hex 03 = Packed decimal
    Hex 0A = Unsigned binary
  - Scalar length — Bin(2)

    If binary:
    - Length (L) (where L = 2 or 4) — Bits 0-15

    If floating-point:
    - Length (L) (where L = 4 or 8) — Bits 0-15

    If zoned decimal or packed decimal:
    - Fractional digits (F) — Bits 0-7
    - Total digits (T) (where $1 \le T \le 31$ and $0 \le F \le T$) — Bits 8-15
  - Reserved (binary 0) — Bin(4)

Exceptions

Exception		Operands 1	2	3	Other
06	Addressing				
	01 space addressing violation	X	X	X	
	02 boundary alignment	X	X	X	
	03 range	X	X	X	
	04 external data object not found	X	X	X	
	06 optimized addressability invalid	X	X	X	
08	Argument/parameter				
	01 parameter reference violation	X	X	X	
0C	Computation				
	02 decimal data		X	X	
	06 floating-point overflow	X			
	07 floating-point underflow	X			
	09 floating-point invalid operand		X		
	0A size	X			
	0C floating-point conversion	X			
	0D floating-point inexact result	X			
10	Damage encountered				
	04 system object damage state	X	X	X	X
	44 partial system object damage	X	X	X	X
1C	Machine-dependent exception				
	03 machine storage limit exceeded				X
20	Machine support				
	02 machine check				X
	03 function check				X
22	Object access				
	01 object not found	X	X	X	
	02 object destroyed	X	X	X	
	03 object suspended	X	X	X	
	08 object compressed				X
24	Pointer specification				
	01 pointer does not exist	X	X	X	
	02 pointer type invalid	X	X	X	
2E	Resource control limit				
	01 user profile storage limit exceeded				X
32	Scalar specification				
	01 scalar type invalid	X	X	X	
	02 scalar attribute invalid			X	
	03 scalar value invalid			X	

Exception		Operands 1	2	3	Other
36	Space management				
	01 space extension/truncation				X

Convert Character to SNA (CVTCS)

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand 3	Operand [4-5]
CVTCS 10CB		Receiver	Controls	Source	
CVTCSB 1CCB	Branch options	Receiver	Controls	Source	Branch targets
CVTCSI 18CB	Indicator options	Receiver	Controls	Source	Indicator targets

Operand 1: Character variable scalar.

Operand 2: Character(15) variable scalar (fixed length).

Operand 3: Character scalar.

Operand 4-5:

- *Branch Form*-Branch point, instruction pointer, relative instruction number, or absolute instruction number.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

ILE access

```
CVTCS (
  var receiver        : aggregate;
      receiver_length : unsigned binary;
  var controls        : aggregate;
  var source          : aggregate;
      source_length   : unsigned binary;
  var return_code     : signed binary
)
```

The return_code will be set as follows:

Return_Code	Meaning
-1	Receiver Overrun.
0	Source Exhausted.

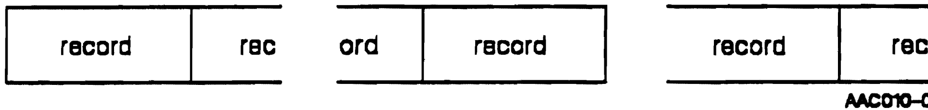
Description: This instruction converts the source (operand 3) from character to SNA (systems network architecture) format under control of the controls (operand 2) and places the result into the receiver (operand 1).

The source and receiver operands must both be character strings. The source operand may not be specified as an immediate operand.

The source operand can be described by the controls operand as being one or more fixed-length data fields that may be separated by fixed-length gaps of characters to be ignored during the conversion operation. Additionally, the controls operand specifies the amount of data to be processed from the source to produce a converted record in the receiver. This may be a different value than the length of the data fields in the source. The following diagram shows this structure for the source operand.

Actual Source Operand Bytes

Data Processed as Source Records



The controls operand must be a character scalar that specifies additional information to be used to control the conversion operation. The operand must be at least 15 bytes in length and has the following format:

- Controls operand — Char(15)
  - Receiver offset — Bin(2)
  - Source offset — Bin(2)
  - Algorithm modifier — Char(1)
  - Source record length — Char(1)
  - Data field length — Bin(2)
  - Gap offset — Bin(2)
  - Gap length — Bin(2)
  - Record separator character — Char(1)
  - Prime compression character — Char(1)
  - Unconverted source record bytes — Char(1)

As input to the instruction, the **source offset** and **receiver offset** fields specify the offsets where the bytes of the source and receiver operands are to be processed. If an offset is equal to or greater than the length specified for the operand, the offset identifies a byte beyond the end of the operand and a *template value invalid* (hex 3801) exception is signaled. When the *source offset* and the *receiver offset* field are output from the instruction, they specify offsets that indicate how much of the operation is complete when the instruction ends.

The **algorithm modifier** specifies the optional functions to be performed. Any combination of functions can be specified as indicated by the bit meanings in the following chart. At least one of the functions must be specified. If all of the *algorithm modifier* bits are zero, a *template value invalid* (hex 3801) exception is signaled. The *algorithm modifier* bit meanings are:

Bits	Meaning
0	0 = Do not perform compression. 1 = Perform compression.
1-2	00 = Do not use record separators and no blank truncation. Do not perform data transparency conversion. 01 = Reserved. 10 = Use record separators and perform blank truncation. Do not perform data transparency conversion. 11 = Use record separators and perform blank truncation. Perform data transparency conversion.
3	0 = Do not perform record spanning. 1 = Perform record spanning. (allowed only when bit 1 = 1)
4-7	(Reserved)

The **source record length** value specifies the amount of data from the source to be processed to produce a converted record in the receiver. Specification of a *source record length* of zero results in a *template value invalid* (hex 3801) exception.

The **data field length** value specifies the length of the data fields in the source. Data fields occurring in the source may be separated by gaps of characters that are to be ignored during the conversion operation. Specification of a *data field length* of zero indicates that the source operand is one data field. In this case, the *gap length* and *gap offset* values have no meaning and are ignored.

The **gap offset** value specifies the offset to the next gap in the source. This value is both input to and output from the instruction. This is relative to the current byte to be processed in the source as located by the source offset value. No validation is done for this offset. It is assumed to be valid relative to the source operand. The *gap offset* value is ignored if the *data field length* is specified with a value of zero.

The **gap length** value specifies the amount of data that is to be ignored between data fields in the source operand during the conversion operation. The *gap length* value is ignored if the *data field length* is zero.

The **record separator character** value specifies the character that precedes the converted form of each record in the receiver. It also serves as a delimiter when the previous record is truncating trailing blanks. The Convert SNA to Character instruction recognizes any value that is less than hex 40. The *record separator* value is ignored if *record separators are not used* as specified in the *algorithm modifier*.

The **prime compression character** value specifies the character to be used as the prime compression character when performing compression of the source data to SNA format. It may have any value. The *prime compression character* value is ignored if the *perform compression* function is not specified in the *algorithm modifier*.

The **unconverted source record bytes** value specifies the number of bytes remaining in the current source record that are yet to be converted.

When the *record spanning* function is specified in the *algorithm modifier*, the *unconverted source record bytes* field is both input to and output from the instruction. On input, a value of hex 00 means it is the start of a new record and the initial conversion step is yet to be performed. That is, a *record separator character* has not yet been placed in the receiver. On input, a nonzero value less than or equal to the *source record length* specifies the number of bytes remaining in the current source record that are yet to be converted into the receiver. This value is assumed to be the valid count of unconverted source record bytes relative to the current byte to be processed in the source as located by the *source offset* value. As such, it is used to determine the location of the next record boundary in the source operand. This value must be less than or equal to the *source record length* value; otherwise, a *template value invalid* (hex 3801) exception is signaled. On output this field is set with a value as defined above that describes the number of bytes of the current source record that have not yet been converted.

When the *record spanning* function is not specified in the *algorithm modifier*, the *unconverted source record bytes* value is ignored.

Only the first 15 bytes of the controls operand are used. Any excess bytes are ignored.

The description of the conversion process is presented as a series of separately performed steps that may be selected in allowable combinations to accomplish the conversion function. It is presented this way to allow for describing these functions separately. However, in the actual execution of the instruction, these functions may be performed in conjunction with one another or separately depending upon which technique is determined to provide the best implementation.

The operation is performed either on a record-by-record basis (record processing) or on a nonrecord basis (string processing). This is determined by the functions selected in the *algorithm modifier*. Specifying the *use record separators* and *perform blank truncation* function indicates **record processing** is to be performed. If this is not specified, in which case *compression* must be specified, it indicates that **string processing** is to be performed.

The operation begins by accessing the bytes of the source operand at the location specified by the *source offset*.

When *record processing* is specified, the *source offset* may locate the start of a full or partial record.

When the *record spanning* function has not been specified in the *algorithm modifier*, the *source offset* is assumed to locate the start of a record.

When the *record spanning* function has been specified in the *algorithm modifier*, the *source offset* is assumed to locate a point at which processing of a possible partially converted record is to be resumed. In this case, the *unconverted source record bytes* value contains the length of the remaining portion of the source record to be converted. The conversion process in this case is started by completing the conversion of the current source record before processing the next full source record.

When *string processing* is specified, the *source offset* locates the start of the source string to be converted.

Only the bytes of the data fields in the source are accessed for conversion purposes. Gaps between data fields are ignored causing the access of data field bytes to occur as if the data fields were contiguous. A string of bytes accessed from the source for a length equal to the *source record length* is considered to be a record for the conversion operation.

When during the conversion process, the end of the source operation is encountered, the instruction ends with a resultant condition of *source exhausted*.

When *record processing* is specified in the *algorithm modifier*, this check is performed at the start of conversion for each record. If the source operand does not contain a full record, the *source exhausted* condition is recognized. The instruction is terminated with status in the controls operand describing the last completely converted record. For *source exhausted*, partial conversion of a source record is not performed.

When *string processing* is specified in the *algorithm modifier*, then *compression* must be specified and the compression function described below defines the detection of *source exhausted*.

If the converted form of the source cannot be completely contained in the receiver, the instruction ends with a resultant condition of *receiver overrun*. See the description of this condition in the conversion process described below to determine the status of the controls operand values and the converted bytes in the receiver for each case.

When *string processing* is specified, the bytes accessed from the source are converted on a string basis into the receiver operand at the location specified by the *receiver offset*. In this case, the *compression* function must be specified and the conversion process proceeds with the compression function defined below.

When *record processing* is specified, the bytes accessed from the source are converted one record at a time into the receiver operand at the location specified by the *receiver offset* performing the functions specified in the *algorithm modifier* in the sequence defined by the following algorithm.

The first function performed is **trailing blank truncation.**: A truncated record is built by logically appending the record data to the *record separator* value specified in the controls operand and removing all blank characters after the last nonblank character in the record. If a record has no trailing blanks, then no actual truncation takes place. A null record, a record consisting entirely of blanks, will be converted as just the *record separator character* with no other data following it. The truncated record then consists of the *record separator* character followed by the truncated record data, the full record data, or no data from the record.

If either the *data transparency conversion* or the *compression* function is specified in the *algorithm modifier*, the conversion process continues for this record with the next specified function.

If not, the conversion process for this record is completed by placing the truncated record into the receiver. If the truncated record cannot be completely contained in the receiver, the instruction ends with a resultant condition of *receiver overrun*. When the *record spanning* function is specified in the *algorithm modifier*, as much of the truncated record as will fit is placed into the receiver and the controls operand is updated to describe how much of the source record was successfully converted into the receiver. When the *record spanning* function is not specified in the *algorithm modifier*, the controls operand is updated to describe only the last fully converted record in the receiver and the value of the remaining bytes in the receiver is unpredictable.

The second function performed is **data transparency conversion.**: *Data transparency conversion* is performed if the function is specified in the *algorithm modifier*. This provides for making the data in a record transparent to the Convert SNA to Character instruction in the area of its scanning for record separator values. Transparent data is built by preceding the data with 2 bytes of **transparency control information**. The first byte has a fixed value of hex 35 and is referred to as the TRN (transparency) control character. The second byte is a 1-byte hexadecimal count, a value ranging from 1 to 255 decimal, of the number of bytes of data that follow and is referred to as the **TRN count**. This contains the length of the data and does not include the TRN control information length.

Transparency conversion can be specified only in conjunction with *record processing* and, as such, is performed on the truncated form of the source record. The transparent record is built by preceding the data that follows the *record separator* in the truncated record with the *TRN control information.* The *TRN count* in this case contains the length of just the truncated data for the record and does not include the *record separator* character. For the special case of a null record, no *TRN control information* is placed after the *record separator* character because there is no record data to be made transparent.

If the *compression* function is specified in the *algorithm modifier*, the conversion process continues for this record with the compression function.

If not, the conversion process for this record is completed by placing the transparent record into the receiver. If the transparent record cannot be completely contained in the receiver, the instruction ends with a resultant condition of *receiver overrun*.

When the *record spanning* function is specified in the *algorithm modifier*, as much of the transparent record as will fit is placed into the receiver and the controls operand is updated to describe how much of the source record was successfully converted into the receiver. The *TRN count* is also adjusted to describe the length of the data successfully converted into the receiver; thus, the transparent data for the record is not spanned out of the receiver. The remaining bytes of the transparent record, if any, will be processed as a partial source record on the next invocation of the instruction and will be preceded by the appropriate *TRN control information*. For the special case where only 1 to 3 bytes are available at the end of the receiver, (not enough room for the *record separator*, the transparency control, and a byte of data) then just the *record separator* is placed in the receiver for the record being converted. This can cause up to 2 bytes of unused space at the end of the receiver. The value of these unused bytes is unpredictable.

When the *record spanning* function is not specified in the *algorithm modifier*, the controls operand is updated to describe only the last fully converted record in the receiver and the value of the remaining bytes in the receiver is unpredictable.

The third function performed is compression.: *Compression* is performed if the function is specified in the *algorithm modifier*. This provides for reducing the size of strings of duplicate characters in the source data. The source data to be compressed may have assumed a partially converted form at this point as a result of processing for functions specified in the *algorithm modifier*. Compressed data is built by concatenating one or more compression strings together to describe the bytes that make up the converted form of the source data prior to the compression step. The bytes of the converted source data are interrogated to locate the prime compression character strings (two or more consecutive prime compression characters), duplicate character strings (three or more duplicate nonprime characters) and nonduplicate character strings occurring in the source.

The character strings encountered (prime, duplicate and nonduplicate) are reflected in the compressed data by building one or more compression strings to describe them. Compression strings are comprised of an SCB (string control byte) possibly followed by one or more bytes of data related to the character string to be described.

The format of an SCB and the description of the data that may follow it are:

- SCB — Char(1)
  - Control — Bits 0-1

    00 = n nonduplicate characters are between this SCB and the next one; where n is the value of the count field (1-63).
    01 = Reserved
    10 = This SCB represents n deleted prime compression characters; where n is the value of the count field (2-63). The next byte is the next SCB.
    11 = This SCB represents n deleted duplicate characters; where n is the value of the count field (3-63). The next byte contains a specimen of the deleted characters. The byte following the specimen character contains the next SCB.
  - Count — Bits 2-7

    This contains the number of characters that have been deleted for a prime or duplicate string, or the number of characters to the next SCB for a nonduplicate string. A count value of zero cannot be produced.

When *record processing* is specified, the compression is performed as follows.

The compression function is performed on just the converted form of the current source record including the *record separator* character. The converted form of the source record prior to the compression step may be a truncated record or a transparent record as described above, depending upon the functions selected in the algorithm modifier. The *record separator* and *TRN control information* is always converted as a nonduplicate compression entry to provide for length adjustment of the *TRN count*, if necessary.

The conversion process for this record is completed by placing the compressed record into the receiver. If the compressed record cannot be completely contained in the receiver, the instruction ends with a resultant condition of *receiver overrun*.

When the *record spanning* function is specified in the *algorithm modifier*, as much of the compressed record as will fit is placed into the receiver and the controls operand is updated to describe how much of the source record was successfully converted into the receiver. The last compression entry placed into the receiver may be adjusted if necessary to a length that provides for filling out the receiver. This length adjustment applies only to compression entries for nonduplicate strings. Compression entries for duplicate strings are placed in the receiver only if they fit with no adjustment. For the

special case where data transparency conversion is specified, the transparent data being described is not spanned out of the receiver. This is provided for by performing length adjustment on the *TRN count* of a transparent record, which may be included in the compressed data so that it describes only the source data that was successfully converted into the receiver. For the special case where only 2 to 5 bytes are available at the end of the receiver, not enough room for the compression entry for a non-duplicate string containing the *record separator* and the TRN control, and up to a 2-byte compression entry for some of the transparent data, the nonduplicate compression entry is adjusted to describe only the *record separator*. By doing this, no more than 3 bytes of unused space will remain in the receiver. The value of these unused bytes is unpredictable. Unconverted source record bytes, if any, will be processed as a partial source record on the next invocation of the instruction and will be preceded by the appropriate *TRN control information* when performing transparency conversion.

When the *record spanning* function is not specified in the *algorithm modifier*, the controls operand is updated to describe only the last fully converted record in the receiver. The value of the remaining bytes in the receiver is unpredictable.

When *string processing* is specified, the compression is performed as follows.

The compression function is performed on the data for the entire source operand on a compression string basis. In this case, the fields in the controls operand related to record processing are ignored.

The conversion process for the source operand is completed by placing the compressed data into the receiver.

When the compressed data cannot be completely contained in the receiver, the instruction ends with a resultant condition of *receiver overrun*. As much of the compressed data as will fit is placed into the receiver and the controls operand is updated to describe how much of the source data was successfully converted into the receiver. The last compression entry placed into the receiver may be adjusted if necessary to a length that provides for filling out the receiver. This length adjustment applies only to compression entries for nonduplicate strings. Compression entries for duplicate strings are placed in the receiver only if they fit with no adjustment. By doing this, no more than 1 byte of unused space will remain in the receiver.

When the compressed data can be completely contained in the receiver, the instruction ends with a resultant condition of *source exhausted*. The compressed data is placed into the receiver and the controls operand is updated to indicate that all of the source data was successfully converted into the receiver.

At this point, either conversion of a source record has been completed or conversion has been interrupted due to detection of the *source exhausted* or *receiver overrun* conditions. For record processing, if neither of the above conditions has been detected either during conversion of or at completion of conversion for the current record, the conversion process continues on the next source record with the blank truncation step described above.

At completion of the instruction, the *receiver offset* locates the byte following the last converted byte in the receiver. The value of the remaining bytes in the receiver after the last converted byte are unpredictable. The *source offset* locates the byte following the last source byte for which conversion was completed. When the *record spanning* function is specified in the *algorithm modifier*, the *unconverted source record bytes* field specifies the length of the remaining source record bytes yet to be converted. When the *record spanning* function is not specified in the *algorithm modifier*, the *unconverted source record bytes* field has no meaning and is not set. The *gap offset* value indicates the offset to the next gap relative to the source offset value set for this condition. The *gap offset* value has no meaning and is not set when the *data field length* is zero.

Limitations: The following are limits that apply to the functions performed by this instruction.

Any form of overlap between the operands on this instruction yields unpredictable results in the receiver operand.

Substring operand references that allow for a null substring reference (a length value of zero) may not be specified for this instruction.

Resultant Conditions

- Source exhausted - All bytes in the source operand have been converted into the receiver operand.
- Receiver overrun - An overrun condition in the receiver operand was detected before all of the bytes in the source operand were processed.

Programming Notes:

If the source operand does not end on a record boundary, in which case the last record is spanned out of the source, this instruction performs conversion only up to the start of that partial record. In this case, the user of the instruction must move this partial record to combine it with the rest of the record in the source operand to provide for its being processed correctly upon the next invocation of the instruction. If full records are provided, the instruction performs its conversions out to the end of the source operand and no special processing is required.

For the special case of a tie between the *source exhausted* and *receiver overrun* conditions, the *source exhausted* condition is recognized first. That is, when *source exhausted* is the resultant condition, the receiver may also be full. In this case, the *receive offset* may contain a value equal to the length specified for the receiver, which would cause an exception to be detected on the next invocation of the instruction. The processing performed for the *source exhausted* condition should provide for this case if the instruction is to be invoked multiple times with the same controls operand template. When the *receiver overrun* condition is the resultant condition, the source will always contain data that can be converted.

Exceptions

Exception		Operands 1	2	3	Other
06	Addressing				
	01 Spacing addressing violation	X	X	X	
	02 Boundary alignment violation	X	X	X	
	03 Range	X	X	X	
	06 Optimized addressability invalid	X	X	X	
08	Argument/parameter				
	01 Parameter reference violation	X	X	X	
10	Damage encountered				
	04 System object damage state				X
	44 Partial system object damage				X
1C	Machine-dependent exception				
	03 Machine storage limit exceeded				X
20	Machine support				
	02 Machine check				X

Exception		Operands 1	2	3	Other
	03 Function check				X
22	Object access				
	01 Object not found	X	X	X	
	02 Object destroyed	X	X	X	
	03 Object suspended	X	X	X	
	08 object compressed				X
24	Pointer specification				
	01 Pointer does not exist	X	X	X	
	02 Pointer type invalid	X	X	X	
2C	Program execution				
	04 Invalid branch target				X
2E	Resource control limit				
	01 user profile storage limit exceeded				X
32	Scalar specification				
	01 Scalar type invalid	X	X	X	
36	Space management				
	01 space extension/truncation				X
38	Template specification				
	01 Template value invalid		X		

Convert Decimal Form to Floating-Point (CVTDFFP)

Op Code (Hex)	Operand 1	Operand 2	Operand 3
107F	Receiver	Decimal exponent	Decimal significand

Operand 1: Floating-point variable scalar.

Operand 2: Packed scalar or zoned scalar.

Operand 3: Packed scalar or zoned scalar.

Description: This instruction converts the decimal form of a floating-point value specified by a decimal exponent and a decimal significand to binary floating-point format, and places the result in the receiver operand. The decimal exponent (operand 2) and decimal significand (operand 3) are considered to specify a decimal form of a floating-point number. The value of this number is considered to be as follows:

```
Value = S * (10**E)
```

where:

S = The value of the decimal significand operand.
E = The value of the decimal exponent operand.
* Denotes multiplication.
** Denotes exponentiation.

The decimal exponent must be specified as a decimal integer value; no fractional digit positions may be specified in its definition. The decimal exponent is a signed integer value specifying a power of 10 which gives the floating-point value its magnitude. A decimal exponent value too large or too small to be represented in the receiver will result in the signaling of the appropriate *floating-point overflow* (hex 0C06) or *floating-point underflow* (hex 0C07) exception.

The decimal significand must be specified as a decimal value with a single integer digit position and optional fractional digit positions. The decimal significand is a signed decimal value specifying decimal digits which give the floating-point value its precision. The significant digits of the decimal significand are considered to start with the leftmost nonzero decimal digit and continue to the right to the end of the decimal significand value. Significant digits beyond 7 for a short float receiver, and beyond 15 for a long float receiver exceed the precision provided for in the binary floating-point receiver. These excess digits do participate in the conversion to provide for uniqueness of the conversion as well as for proper rounding.

The decimal form floating-point value specified by the decimal exponent and decimal significand operands is converted to a binary floating-point number and rounded to the precision of the result field as follows:

Source values which, in magnitude M, are in the range where (10**31-1) * 10**-31 <= M <= (10**31-1) *10**+31 are converted subject to the normal rounding error defined for the floating-point rounding modes.

Source values which, in magnitude M, are in the range where (10**31-1) * 10**-31 > M > (10**31-1) *10**+31 are converted such that the rounding error incurred on the conversion may exceed that defined above. For round to nearest, this error will not exceed by more than .47 units in the least significant digit position of the result in relation to the error that would be incurred for normal rounding. For the other floating-point rounding modes, this error will not exceed 1.47 units in the least significant digit position of the result.

The converted and rounded value is then assigned to the floating-point receiver.

Exceptions

Exception		Operands 1	2	3	Other
06	Addressing				
	01 Spacing addressing violation	X	X	X	
	02 Boundary alignment violation	X	X	X	
	03 Range	X	X	X	
	06 Optimized addressability invalid	X	X	X	
08	Argument/parameter				
	01 Parameter reference violation	X	X	X	
0C	Computation				
	02 Decimal data		X	X	
	06 Floating-point overflow	X			
	07 Floating-point underflow	X			
	0D Floating-point inexact result	X			
10	Damage encountered				
	04 System object damage state				X
	44 Partial system object damage				X
1C	Machine-dependent exception				
	03 Machine storage limit exceeded				X
20	Machine support				
	02 Machine check				X
	02 Function check				X
22	Object access				
	01 Object not found	X	X	X	
	02 Object destroyed	X	X	X	
	03 Object suspended	X	X	X	
	08 object compressed				X
24	Pointer specification				
	01 Pointer does not exist	X	X	X	
	02 Pointer type invalid	X	X	X	
2E	Resource control limit				
	01 user profile storage limit exceeded				X
32	Scalar specification				
	01 Scalar type invalid	X	X	X	
36	Space management				
	01 space extension/truncation				X

Convert External Form to Numeric Value (CVTEFN)

Op Code (Hex)	Operand 1	Operand 2	Operand 3
1087	Receiver	Source	Mask

Operand 1: Numeric variable scalar or data-pointer-defined numeric scalar.

Operand 2: Character scalar or data-pointer-defined character scalar.

Operand 3: Character(3) scalar, null, or data-pointer-defined character(3) scalar.

ILE access

```
CVTEFN (
  var receiver            : any numeric type;
  var receiver_attributes : aggregate;
  var source              : aggregate;
  var source_length       : unsigned binary;
  var mask                : aggregate
)
```

Description: This instruction scans a character string for a valid decimal number in display format, removes the display character, and places the results in the receiver operand. The operation begins by scanning the character string value in the source operand to make sure it is a valid decimal number in display format.

The character string defined by operand 2 consists of the following optional entries:

- Currency symbol - This value is optional and, if present, must precede any sign and digit values. The valid symbol is determined by operand 3. The currency symbol may be preceded in the field by blank (hex 40) characters.
- Sign symbol - This value is optional and, if present, may precede any digit values (a leading sign) or may follow the digit values (a trailing sign). Valid signs are positive (hex 4E) and negative (hex 60). The sign symbol, if it is a leading sign, may be preceded by blank characters. If the sign symbol is a trailing sign, it must be the rightmost character in the field. Only one sign symbol is allowed.
- Decimal digits - Up to 31 decimal digits may be specified. Valid decimal digits are in the range of hex F0 through hex F9 (0-9). The first decimal digit may be preceded by blank characters (hex 40), but hex 40 values located to the right of the leftmost decimal digit are invalid.

The decimal digits may be divided into two parts by the decimal point symbol: an integer part and a fractional part. Digits to the left of the decimal point are interpreted as integer values. Digits to the right are interpreted as a fractional values. If no decimal point symbol is included, the value is interpreted as an integer value. The valid decimal point symbol is determined by operand 3. If the decimal point symbol precedes the leftmost decimal digit, the digit value is interpreted as a fractional value, and the leftmost decimal digit must be adjacent to the decimal point symbol. If the decimal point follows the rightmost decimal digit, the digit value is interpreted as an integer value, and the rightmost decimal digit must be adjacent to the decimal point.

Decimal digits in the integer portion may optionally have comma symbols separating groups of three digits. The leftmost group may contain one, two, or three decimal digits, and each succeeding group must be preceded by the comma symbol and contain three digits. The comma symbol must be adjacent to a decimal digit on either side. The valid comma symbol is determined by operand 3.

Decimal digits in the fractional portion may not be separated by commas and must be adjacent to one another.

Examples of external formats follow. The following symbols are used.

$	currency symbol
.	decimal point
,	comma
D	digit (hex F0-F9)
	blank (hex 40)
+	positive sign
-	negative sign

Format	Comments
$+DDDD.DD	Currency symbol, leading sign, no comma separators
DD,DDD-	Comma symbol, no fraction, trailing sign
-.DDD	No integer, leading sign
$DDD,DDD-	No fraction, comma symbol, trailing sign
$ + DD.DD	Embedded blanks before digits

Operand 3 must be a 3-byte character scalar. Byte 1 of the string indicates the byte value that is to be used for the currency symbol. Byte 2 of the string indicates the byte value to be used for the comma symbol. Byte 3 of the string indicates the byte value to be used for the decimal point symbol. If operand 3 is null, the currency symbol (hex 5B), comma (hex 6B), and decimal point (hex 4B) are used.

If the syntax rules are violated, a *conversion* (hex 0C01) exception is signaled. If not, a zoned decimal value is formed from the digits of the display format character string. This number is placed in the receiver operand following the rules of a normal arithmetic conversion.

If a decimal to binary conversion causes a *size* (hex 0C0A) exception to be signaled, the binary value contains the correct truncated result only if the decimal value contains 15 or fewer significant nonfractional digits.

Exceptions

Exception		Operands 1	2	3	Other
06	Addressing				
	01 Spacing addressing violation	X	X	X	
	02 Boundary alignment	X	X	X	
	03 Range	X	X	X	
	06 Optimized addressability invalid	X	X	X	
08	Argument/parameter				
	01 Parameter reference violation	X	X	X	
0C	Computation				
	01 Conversion		X		
	0A Size	X			
10	Damage encountered				
	04 System object damage	X	X	X	X
	44 Partial system object damage	X	X	X	X

Exception		Operands 1	2	3	Other
1C	Machine-dependent exception				
	03 Machine storage limit exceeded				X
20	Machine support				
	02 Machine check				X
	03 Function check				X
22	Object access				
	01 Object not found	X	X	X	
	02 Object destroyed	X	X	X	
	03 Object suspended	X	X	X	
	08 object compressed				X
24	Pointer specification				
	01 Pointer does not exist	X	X	X	
	02 Pointer type invalid	X	X	X	
2E	Resource control limit				
	01 user profile storage limit exceeded				X
32	Scalar specification				
	01 Scalar type invalid	X	X	X	
	02 Scalar attribute invalid			X	
36	Space management				
	01 space extension/truncation				X

Convert Floating-Point to Decimal Form (CVTFPDF)

Op Code (Hex)	Operand 1	Operand 2	Operand 3
CVTFPDF 10BF	Decimal exponent	Decimal significand	Source
CVTFPDFR 12BF	Decimal exponent	Decimal significand	Source

Operand 1: Packed variable scalar or zoned variable scalar.

Operand 2: Packed variable scalar or zoned variable scalar.

Operand 3: Floating-point scalar.

Description: This instruction converts a binary floating-point value to a decimal form of a floating-point value specified by a decimal exponent and a decimal significand, and places the result in the decimal exponent and decimal significand operands.

The value of this number is considered to be as follows:

```
Value = S * (10**E)
```

where:

S = The value of the decimal significand operand.

E = The value of the decimal exponent operand.

* Denotes multiplication.

** Denotes exponentiation.

The decimal exponent must be specified as a decimal integer value. No fractional digit positions are allowed. It must be specified with at least five digit positions. The decimal exponent provides for containing a signed integer value specifying a power of 10 which gives the floating-point value its magnitude.

The decimal significand must be specified as a decimal value with a single integer digit position and optional fractional digit positions. The decimal significand provides for containing a signed decimal value specifying decimal digit is which give the floating-point value its precision. The decimal significand is formed as a normalized value, that is, the leftmost digit position is nonzero for a nonzero source value.

When the source contains a representation of a normalized binary floating-point number with decimal significand digits beyond the leftmost 7 digits for a short floating-point source or beyond the leftmost 15 digits for a long floating-point source, the precision allowed for the binary floating-point source is exceeded.

When the source contains a representation of a denormalized binary floating-point number, it may provide less precision than the precision of a normalized binary floating-point number, depending on the particular source value. Decimal significand digits exceeding the precision of the source are set as a result of the conversion to provide for uniqueness of conversion and are correct, except for rounding errors. These digits are only as precise as the floating-point calculations that produced the source value. The *floating-point inexact result* (hex 0C0D) exception provides a means of detecting loss of precision in floating-point calculations.

The binary floating-point source is converted to a decimal form floating-point value and rounded to the precision of the decimal significand operand as follows:

- The decimal significand is formed as a normalized value and the decimal exponent is set accordingly.
- For the nonround form of the instruction, the value to be assigned to the decimal significand is adjusted to the precision of the decimal significand, if necessary, according to the current float rounding mode in effect for the process. For the optional round form of the instruction, the decimal round algorithm is used for the precision adjustment of the decimal significand. The decimal round algorithm overrides the current floating-point rounding mode that is in effect for the process.
- Source values which, in magnitude M, are in the range where (10**31-1) * 10**-31 <= M <= (10**31-1) * 10**+31 are converted subject to the normal rounding error defined for the floating-point rounding modes and the optional round form of the instruction.
- Source values which, in magnitude M, are in the range where (10**31-1) * 10**-31 > M > (10**31-1) * 10**+31 are converted such that the rounding error incurred on the conversion may exceed that defined above. For round to nearest and the optional round form of the instruction, this error will not exceed by more than .47 units in the least significant digit position of the result, the error that would be incurred for a correctly rounded result. For the other floating-point rounding modes, this error will not exceed 1.47 units in the least significant digit position of the result.
- If necessary, the decimal exponent value is adjusted to compensate for rounding.
- The converted and rounded value is then assigned to the decimal exponent and decimal significand operands.

A *size* (hex 0C0A) exception cannot occur on the assignment of the decimal exponent or the decimal significand values.

Limitations: The following are limits that apply to the functions performed by this instruction.

The result of the operation is unpredictable for any type of overlap between the decimal exponent and decimal significand operands.

Exceptions

Exception		Operands 1	2	3	Other
06	Addressing				
	01 Spacing addressing violation	X	X	X	
	02 Boundary alignment violation	X	X	X	
	03 Range	X	X	X	
	06 Optimized addressability invalid	X	X	X	
08	Argument/parameter				
	01 Parameter reference violation	X	X	X	
0C	Computation				
	0C Invalid floating-point conversion	X	X		
	0D Floating-point inexact result		X		
10	Damage encountered				
	04 System object damage state				X
	44 Partial system object damage				X
1C	Machine-dependent exception				
	03 Machine storage limit exceeded				X

Exception		Operands 1	2	3	Other
20	Machine support				
	02 Machine check				X
	03 Function check				X
22	Object access				
	01 Object not found	X	X	X	
	02 Object destroyed	X	X	X	
	03 Object suspended	X	X	X	
	08 object compressed				X
24	Pointer specification				
	01 Pointer does not exist	X	X	X	
	02 Pointer type invalid	X	X	X	
2E	Resource control limit				
	01 user profile storage limit exceeded				X
32	Scalar specification				
	04 Scalar type invalid	X	X	X	
36	Space management				
	01 space extension/truncation				X

Convert Hex to Character (CVTHC)

Op Code (Hex)	Operand 1	Operand 2
1086	Receiver	Source

Operand 1: Character variable scalar.

Operand 2: Character variable scalar.

Description: Each hex digit (4-bit value) of the string value in the source operand is converted to a character (8-bit value) and placed in the receiver operand.

Hex Digits	Characters
Hex 0-9 =	Hex F0-F9
Hex A-F =	Hex C1-C6

The operation begins with the two operands left-adjusted and proceeds left to right until all the characters of the receiver operand have been filled. If the source operand contains fewer hex digits than needed to fill the receiver, the excess characters are assigned a value of hex F0. If the source operand is too large, a *length conformance* (hex 0C08) or an *invalid operand length* (hex 2A0A) exception is signaled.

Substring operand references that allow for a null substring reference (a length value of zero) may be specified for operands 1 and 2. The effect of specifying a null substring reference for the source is that the bytes of the receiver are each set with a value of hex F0. The effect of specifying a null substring reference for the receiver is that no result is set.

Exceptions

Exception		Operands 1	2	Other
06	Addressing			
	01 Spacing addressing violation	X	X	
	02 Boundary alignment	X	X	
	03 Range	X	X	
	06 Optimized addressability invalid	X	X	
08	Argument/parameter			
	01 Parameter reference violation	X	X	
0C	Computation			
	08 Length conformance		X	
10	Damage encountered			
	04 System object damage state	X	X	X
	44 Partial system object damage	X	X	X
1C	Machine-dependent exception			
	03 Machine storage limit exceeded			X
20	Machine support			
	02 Machine check			X
	02 Function check			X

Exception		Operands 1	Operands 2	Other
22	Object access			
	01 Object not found	X	X	
	02 Object destroyed	X	X	
	03 Object suspended	X	X	
	08 object compressed			X
24	Pointer specification			
	01 Pointer does not exist	X	X	
	02 Pointer type invalid	X	X	
2E	Resource control limit			
	01 user profile storage limit exceeded			X
36	Space management			
	01 space extension/truncation			X

Convert MRJE to Character (CVTMC)

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand 3	Operand [4-5]
CVTMC 10AB		Receiver	Controls	Source	
CVTMCB 1CAB	Branch options	Receiver	Controls	Source	Branch targets
CVTMCI 18AB	Indicator options	Receiver	Controls	Source	Indicator targets

Operand 1: Character variable scalar.

Operand 2: Character(6) variable scalar (fixed-length).

Operand 3: Character scalar.

Operand 4-5:

- *Branch Form*-Branch point, instruction pointer, relative instruction number, or absolute instruction number.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

ILE access

```
CVTMC (
  var receiver        : aggregate;
      receiver_length : unsigned binary;
  var controls        : aggregate;
  var source          : aggregate;
      source_length   : unsigned binary;
  var return_code     : signed binary
)
```

The return_code will be set as follows:

Return_Code	Meaning
-1	Receiver Overrun.
0	Source Exhausted.

Description: This instruction converts a character string from the MRJE (MULTI-LEAVING remote job entry) compressed format to character format. The operation converts the source (operand 3) from the MRJE compressed format to character format under control of the controls (operand 2) and places the results in the receiver (operand 1).

The source and receiver operands must both be character strings. The source operand cannot be specified as either a signed or unsigned immediate value.

The controls operand must be a character scalar that specifies additional information to be used to control the conversion operation. It must be at least 6 bytes in length and have the following format:

- Controls operand — Char(6)
  - Receiver offset — Bin(2)
  - Source offset — Bin(2)

— Algorithm modifier Char(1)

— Receiver record length Char(1)

As input to the instruction, the **source offset** and **receiver offset** fields specify the offsets where bytes of the source and receiver operands are to be processed. If an offset is equal to or greater than the length specified for the operand it corresponds to (it identifies a byte beyond the end of the operand), a *template value invalid* (hex 3801) exception is signaled. As output from the instruction, the *source offset* and *receiver offset* fields specify offsets that indicate how much of the operation is complete when the instruction ends.

The **algorithm modifier** has the following valid values:

- Hex 00 = Do not move SRCBs (sub record control bytes) from the source into the receiver.
- Hex 01 = Move SRCBs from the source into the receiver.

The **receiver record length** value specifies the record length to be used to convert source records into the receiver operand. This length applies to only the string portion of the receiver record and does not include the optional SRCB field. If a *receiver record length* of 0 is specified, a *template value invalid* (hex 3801) exception is signaled.

Only the first 6 bytes of the controls operand are used. Any excess bytes are ignored.

The operation begins by accessing the bytes of the source operand at the location specified by the *source offset*. This is assumed to be the start of a record. The bytes of the records in the source operand are converted into the receiver operand at the location specified by the *receiver offset* according to the following algorithm.

The first byte of the source record is considered to be an RCB (record control byte) that is to be ignored during conversion.

The second byte of the source record is considered to be an SRCB. If an *algorithm modifier* of value hex 00 was specified, the SRCB is ignored. If an *algorithm modifier* of value hex 01 was specified, the SRCB is copied into the receiver.

The strings to be built in the receiver record are described in the source after the SRCB by one or more SCBs (string control bytes).

The format of the SCBs in the source are as follows:

o k l jjjjj

The bit meanings are:

Bit	Value	Meaning
o	0	End of record;the EOR SCB is hex 00.
	1	All other SCBs.
k	0	The string is compressed.
	1	The string is not compressed.
l		For k = 0:
	0	Blanks (hex 40s) have been deleted.
	1	Nonblank characters have been deleted. The next character in the data stream is the specimen character.
		For k = 1:
		This bit is part of the length field for length of uncompressed data.

Bit	Value	Meaning
jjjjj		Number of characters that have been deleted if k = 0. The value can be 1-31.
ljjjjj		Number of characters to the next SCB (no compression) if k=1. The value can be 1-63.
		The uncompressed (nonidentical bytes) follow the SCB in the data stream.

A length of 0 encountered in an SCB results in the signaling of a *conversion* (hex 0C01) exception.

Strings of blanks or nonblank identical characters described in the source record are repeated in the receiver the number of times indicated by the SCB count value.

Strings of nonidentical characters described in the source record are moved into the receiver for the length indicated by the SCB count value.

When an EOR (end of record) SCB (hex 00) is encountered in the source, the receiver is padded with blanks out to the end of the current record.

If the converted form of a source record is larger than the *receiver record length*, the instruction is terminated by signaling a *length conformance* (hex 0C08) exception.

If the end of the source operand is not encountered, the operation then continues by reapplying the above algorithm to the next record in the source operand.

If the end of the source operand is encountered (whether or not in conjunction with a record boundary, EOR SCB in the source), the instruction ends with a resultant condition of *source exhausted*. The *receiver offset* locates the byte following the last fully converted record in the receiver. The *source offset* locates the byte following the last source record for which conversion is complete. The value of the remaining bytes in the receiver after the last converted record are unpredictable.

If the converted form of a record cannot be completely contained in the receiver, the instruction ends with a resultant condition of *receiver overrun*. The *receiver offsert* locates the byte following the last fully converted record in the receiver. The *source offset* locates the byte following the last source record for which conversion is complete. The value of the remaining bytes in the receiver after the last converted record is unpredictable.

If the *source exhausted* and the *receiver overrun* conditions occur at the same time, the *source exhausted* condition is recognized first. In this case, the *receiver offset* may contain a value equal to the length specified for the receiver which causes an exception to be signaled on the next invocation of the instruction. The processing performed for the *source exhausted* condition provides for this case if the instruction is invoked multiple times with the same controls operand template. When the *receiver overrun* condition is the resultant condition, the source always contains data that can be converted.

Limitations: The following are limits that apply to the functions performed by this instruction.

Any form of overlap between the operands on this instruction yields unpredictable results in the receiver operand.

Resultant Conditions

- Source exhausted - All full records in the source operand have been converted into the receiver operand.
- Receiver overrun - An overrun condition in the receiver operand was detected prior to processing all of the bytes in the source operand.

Exceptions

Exception		Operands 1	2	3	Other
06	Addressing				
	01 Spacing addressing violation	X	X	X	
	02 Boundary alignment violation	X	X	X	
	03 Range	X	X	X	
	06 Optimized addressability invalid	X	X	X	
08	Argument/parameter				
	01 Parameter reference violation	X	X	X	
0C	Computation				
	01 Conversion			X	
	08 Length conformance	X			
10	Damage encountered				
	04 System object damage state				X
	44 Partial system object damage				X
1C	Machine-dependent exception				
	03 Machine storage limit exceeded				X
20	Machine support				
	02 Machine check				X
	03 Function check				X
22	Object access				
	01 Object not found	X	X	X	
	02 Object destroyed	X	X	X	
	03 Object suspended	X	X	X	
	08 object compressed				X
24	Pointer specification				
	01 Pointer does not exist	X	X	X	
	02 Pointer type invalid	X	X	X	
2C	Program execution				
	04 Invalid branch target				X
2E	Resource control limit				
	01 user profile storage limit exceeded				X
32	Scalar specification				
	01 Scalar type invalid	X	X	X	
36	Space management				
	01 space extension/truncation				X
38	Template specification				
	01 Template value invalid		X		

Convert Numeric to Character (CVTNC)

Op Code (Hex)	Operand 1	Operand 2	Operand 3
10A3	Receiver	Source	Attributes

Operand 1: Character variable scalar or data-pointer-defined character scalar.

Operand 2: Numeric scalar or data-pointer-defined numeric scalar.

Operand 3: Character(7) scalar or data-pointer-defined character(7) scalar.

Description: The source numeric value (operand 2) is converted and copied to the receiver character string (operand 1). The receiver operand is treated as though it had the attributes supplied by operand 3. Operand 1, when viewed in this manner, receives the numeric value of operand 2 following the rules of the Copy Numeric Value instruction.

The format of operand 3 is as follows:

- Scalar attributes — Char(7)
  - Scalar type — Char(1)

    Hex 00 = Signed binary
    Hex 01 = Floating-point
    Hex 02 = Zoned decimal
    Hex 03 = Packed decimal
    Hex 0A = Unsigned binary
  - Scalar length — Bin(2)

    If binary:
    - Length (L) (where L = 2 or 4) — Bits 0-15

    If floating-point:
    - Length (where L = 4 or 8) — Bits 0-15

    If zoned decimal or packed decimal:
    - Fractional digits (F) — Bits 0-7
    - Total digits (T) — Bits 8-15
      (where $1 \leq T \leq 31$ and $0 \leq F \leq T$)
  - Reserved (binary 0) — Bin(4)

The byte length of operand 1 must be large enough to contain the numeric value described by operand 3. If it is not large enough, a *scalar value invalid* (hex 3203) exception is signaled. If it is larger than needed, the numeric value is placed in the leftmost bytes and the unneeded rightmost bytes are unchanged by the instruction.

If a decimal to binary conversion causes a *size* (hex 0C0A) exception to be signaled, the binary value contains the correct truncated result only if the decimal value contains 15 or fewer significant nonfractional digits.

Exceptions

Exception		Operands 1	2	3	Other
06	Addressing				

Exception		Operands 1	2	3	Other
	01 Spacing addressing violation	X	X	X	
	02 Boundary alignment	X	X	X	
	03 Range	X	X	X	
	04 External data object not found	X	X	X	
	06 Optimized addressability invalid	X	X	X	
08	Argument/parameter				
	01 Parameter reference violation	X	X	X	
0C	Computation				
	02 Decimal data		X		
	06 Floating-point overflow	X			
	07 Floating-point underflow	X			
	09 Floating-point invalid operand		X		
	0A Size	X			
	0C Invalid floating-point conversion	X			
	0D Floating-point inexact result	X			
10	Damage encountered				
	04 System object damage state	X	X	X	X
	44 Partial system object damage	X	X	X	X
1C	Machine-dependent exception				
	03 Machine storage limit exceeded				X
20	Machine support				
	02 Machine check				X
	03 Function check				X
22	Object access				
	01 Object not found	X	X	X	
	02 Object destroyed	X	X	X	
	03 Object suspended	X	X	X	
	08 object compressed				X
24	Pointer specification				
	01 Pointer does not exist	X	X	X	
	02 Pointer type invalid	X	X	X	
2E	Resource control limit				
	01 user profile storage limit exceeded				X
32	Scalar specification				
	01 Scalar type invalid	X	X	X	
	02 Scalar attribute invalid				
	03 Scalar value invalid			X	
36	Space management				
	01 space extension/truncation				X

Convert SNA to Character (CVTSC)

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand 3	Operand [4-6]
CVTSC 10DB		Receiver	Controls	Source	
CVTSCB 1CDB	Branch options	Receiver	Controls	Source	Branch targets
CVTSCI 18DB	Indicator options	Receiver	Controls	Source	Indicator targets

Operand 1: Character variable scalar.

Operand 2: Character(14) variable scalar (fixed length).

Operand 3: Character scalar.

Operand 4-6:

- *Branch Form*-Branch point, instruction pointer, relative instruction number, or absolute instruction number.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

ILE access

```
CVTSC (
  var receiver        : aggregate;
      receiver_length : unsigned binary;
  var controls        : aggregate;
  var source          : aggregate;
      source_length   : unsigned binary;
  var return_code     : signed binary
)
```

The return_code will be set as follows:

Return_Code	Meaning
-1	Receiver Overrun.
0	Source Exhausted.
1	Escape Code Encountered

Description: This instruction converts a string value from SNA (systems network architecture) format to character. The operation converts the source (operand 3) from SNA format to character under control of the controls (operand 2) and places the result into the receiver (operand 1).

The source and receiver operands must both be character strings. The source operand may not be specified as an immediate operand.

The controls operand must be a character scalar that specifies additional information to be used to control the conversion operation. It must be at least 14 bytes in length and have the following format:

- Controls operand base template — Char(14)
  - Receiver offset — Bin(2)

– Source offset	Bin(2)
– Algorithm modifier	Char(1)
– Receiver record length	Char(1)
– Record separator	Char(1)
– Prime compression	Char(1)
– Unconverted receiver record bytes	Char(1)
– Conversion status	Char(2)
– Unconverted transparency string bytes	Char(1)
– Offset into template to translate table	Bin(2)
• Controls operand optional template extension	Char(64)
– Record separator translate table	Char(64)

Upon input to the instruction, the **source offset** and **receiver offset** fields specify the offsets where bytes of the source and receiver operands are to be processed. If an offset is equal to or greater than the length specified for the operand it corresponds to (it identifies a byte beyond the end of the operand), a *template value invalid* (hex 3801) exception is signaled. As output from the instruction, the *source offset* and *receiver offset* are set to specify offsets that indicate how much of the operation is complete when the instruction ends.

The **algorithm modifier** specifies the optional functions to be performed. Any combination of functions not precluded by the bit definitions below is valid except that at least one of the functions must be specified. All *algorithm modifier* bits cannot be zero. Specification of an invalid *algorithm modifier* value results in a *template value invalid* (hex 3801) exception. The meaning of the bits in the algorithm modifier is the following:

Bits	Meaning
0	0 = Do not perform decompression. Interpret a source character value of hex 00 as null. 1 = Perform decompression. Interpret a source character value of hex 00 as a record separator.
1-2	00 = No record separators in source, no blank padding. Do not perform data transparency conversion. 01 = Reserved. 10 = Record separators in source, perform blank padding. Do not perform data transparency conversion. 11 = Record separators in source, perform blank padding. Perform data transparency conversion.
3-4	00 = Do not put record separators into receiver. 01 = Move record separators from source to receiver (allowed only when bit 1 = 1) 10 = Translate record separators from source to receiver (allowed only when bit 1 = 1) 11 = Move record separator from controls to receiver.
5-7	Reserved

The **receiver record length** value specifies the record length to be used to convert source records into the receiver operand. This length applies only to the data portion of the receiver record and does not include the optional record separator. Specification of a *receiver record length* of zero results in a *template value invalid* (hex 3801) exception. The *receiver record length* value is ignored if no *record separator processing* is requested in the *algorithm modifier*.

The **record separator value** specifies the character that is to precede the converted form of each record in the receiver. The *record separator character* specified in the controls operand is used only for the case where the *move record separator from controls to receiver* function is specified in the *algorithm modifier* or where a missing record separator in the source is detected.

The **prime compression** value specifies the character to be used as the prime compression character when performing decompression of the SNA format source data to character. It may have any value. The *prime compression* value is ignored if the *perform decompression* function is not specified in the *algorithm modifier*.

The **unconverted receiver record bytes** value specifies the number of bytes remaining in the current receiver record that are yet to be set with converted bytes from the source.

When *record separator processing* is specified in the *algorithm modifier*, this value is both input to and output from the instruction. On input, a value of hex 00 means it is the start of processing for a new record, and the initial conversion step is yet to be performed. This indicates that for the case where a function for putting record separators into the receiver is specified in the *algorithm modifier*, a *record separator character* has yet to be placed in the receiver. On input, a nonzero value less than or equal to the *receiver record length* specifies the number of bytes remaining in the current receiver record that are yet to be set with converted bytes from the source. This value is assumed to be the valid count of unconverted receiver record bytes relative to the current byte to be processed in the receiver as located by the *receiver offset* field. As such, it is used to determine the location of the next record boundary in the receiver operand. This value must be less than or equal to the *receiver record length* value; otherwise, a *template value invalid* (hex 3801) exception is signaled. On output, this field is set with a value as defined above which describes the number of bytes of the current receiver record not yet containing converted data.

When *record separator processing* is not specified in the *algorithm modifier*, this value is ignored.

The **conversion status** field specifies status information for the operation to be performed. The meaning of the bits in the conversion status is the following:

Bits	Meaning
0	0 = No transparency string active. 1 = Transparency string active. Unconverted transparency string bytes value contains the remaining string length.
1-15	Reserved

This field is both input to and output from the instruction. It provides for checkpointing the conversion status over successive executions of the instruction.

If the *conversion status* indicates *transparency string active*, but the *algorithm modifier* does not *specify perform data transparency conversion*, a *template value invalid* (hex 3801) exception is signaled.

The **unconverted transparency string bytes** field specifies the number of bytes remaining to be converted for a partially processed transparency string in the source.

When *perform data transparency conversion* is specified in the *algorithm modifier*, the *unconverted transparency string bytes* field can be both input to and output from the instruction.

On input, when the no *transparency string active* status is specified in the *conversion status*, this value is ignored.

On input, when *transparency string active* status is specified in the *conversion status*, this value contains a count for the remaining bytes to be converted for a transparency string in the source. A value of hex 00 means the count field for a transparency string is the first byte of data to be processed from the source operand. A value of hex 01 through hex FF specifies the count of the remaining bytes to be converted for a transparency string. This value is assumed to be the valid count of unconverted transparency string bytes relative to the current byte to be processed in the source as located by the *source offset* field.

On output, this value is set if necessary along with the *transparency string active* status to describe a partially converted transparency string. A value of hex 00 will be set if the count field is the next byte to be processed for a transparency string. A value of hex 01 through hex FF specifying the number of remaining bytes to be converted for a transparency string, will be set if the count field has already been processed.

When *do not perform data transparency conversion* is specified in the *algorithm modifier*, the *unconverted transparency string bytes* value is ignored.

The **offset into template to translate table** value specifies the offset from the beginning of the template to the *record separator translate table*. This value is ignored unless the *translate record separators from source to* receiver function is specified in the *algorithm modifier*.

The **record separator translate table** value specifies the translate table to be used in translating record separators specified in the source to the record separator value to be placed into the receiver. It is assumed to be 64 bytes in length, providing for translation of record separator values of from hex 00 to hex 3F. This translate table is used only when the *translate record separators from source to receiver* function is specified in the *algorithm modifier*. See the record separator conversion function under the conversion process described below for more detail on the usage of the translate table.

Only the first 14 bytes of the controls operand base template and the optional 64-byte extension area specified for the record separator translate table are used. Any excess bytes are ignored.

The description of the conversion process is presented as a series of separately performed steps, which may be selected in allowable combinations to accomplish the conversion function. It is presented this way to allow for describing these functions separately. However, in the actual execution of the instruction, these functions may be performed in conjunction with one another or separately, depending upon which technique is determined to provide the best implementation.

The operation is performed either on a record-by-record basis, record processing, or on a nonrecord basis, string processing. This is determined by the functions selected in the *algorithm modifier*. Specifying the *record separators in source*, *perform blank padding* or *move record separator from controls to receiver* indicates **record processing** is to be performed. If neither of these functions is specified, in which case *decompression* must be specified, it indicates that **string processing** is to be performed.

The operation begins by accessing the bytes of the source operand at the location specified by the *source offset*.

When *record processing* is specified, the *source offset* may locate a point at which processing of a partially converted record is to be resumed or processing for a full record is to be started. The *unconverted receiver record bytes* field indicates whether conversion processing is to be started with a partial or a full record. Additionally, the *transparency string active* indicator in the *conversion status* field indicates whether conversion of a transparency string is active for the case of resumption of processing for a partially converted record. The conversion process is started by completing the conversion of a partial source record if necessary before processing the first full source record.

When *string processing* is specified, the *source offset* is assumed to locate the start of a compression entry.

When during the conversion process the end of the receiver operand is encountered, the instruction ends with a *resultant condition* or *receiver overrun.*

When *record processing* is specified in the *algorithm modifier*, this check is performed at the start of conversion for each record. A *source exhausted* condition would be detected before a *receiver overrun* condition if there is no source data to convert. If the receiver operand does not have room for a full record, the *receiver overrun* condition is recognized. The instruction is terminated with status in the

controls operand describing the last completely converted record. For *receiver overrun*, partial conversion of a source record is not performed.

When *string processing* is specified in the *algorithm modifier*, then decompression must be specified and the decompression function described below defines the detection of *receiver overrun.*

When during the conversion process the end of the source operand is encountered, the instruction ends with a resultant condition of *source exhausted.* See the description of this condition in the conversion process described below to determine the status of the controls operand values and the converted bytes in the receiver for each case.

When *string processing* is specified, the bytes accessed from the source are converted on a string basis into the receiver operand at the location specified by the *receiver offset*. In this case, the *decompression* function must be specified and the conversion process is accomplished with just the decompression function defined below.

When *record processing* is specified, the bytes accessed from the source are converted one record at a time into the receiver operand at the location specified by the *receiver offset* performing the functions specified in the *algorithm modifier* in the sequence defined by the following algorithm.

Record separator conversion is performed as requested in the *algorithm modifier* during the initial record separator processing performed as each record is being converted. This provides for controlling the setting of the *record separator* value in the receiver.

When the *record separators in source* option is specified, the following algorithm is used to locate them. A record separator is recognized in the source when a character value less than hex 40 is encountered. When *do not perform decompression* is specified, a source character value of hex 00 is recognized as a null value rather than as a record separator. In this case, the processing of the current record continues with the next source byte and the receiver is not updated. When *perform data transparency conversion* is specified, a character value of hex 35 is recognized as the start of a transparency string rather than as a record separator.

If the *do not put record separators into the receiver* function is specified, the record separator, if any, from the source record being processed is removed from the converted form of the source record and will not be placed in the receiver.

If the *move record separators from the source to the receiver* function is specified, the record separator from the source record being processed is left as is in the converted form of the source record and will be placed in the receiver.

If the *translate record separators from the source to the receiver* function is specified, the record separator from the source record being processed is translated using the specified translate table, replaced with its translated value in the converted form of the source record and, will be placed in the receiver. The translation is performed as in the translate instruction with the *record separator* value serving as the source byte to be translated. It is used as an index into the specified translate table to select the byte in the translate table that contains the value to which the record separator is to be set. If the selected translate table byte is equal to hex FF, it is recognized as an escape code. The instruction ends with a resultant condition of *escape code* encountered, and the controls operand is set to describe the *conversion status* as of the processing completed just prior to the conversion step for the record separator. If the selected translate table byte is not equal to hex FF, the record separator in the converted form of the record is set to its value.

If the *move record separator from controls to receiver* function is specified, the controls *record separator* value is used in the converted form of the source record and will be placed into the receiver.

When the *record separators in source do blank padding* function is requested, an assumed record separator will be used if a record separator is missing in the source data. In this case, the controls *record separator character* is used as the record separator to precede the converted record if record separators are to be placed in the receiver. The conversion process continues, bypassing the record separator conversion step that would normally be performed. The condition of a missing record separator is detected when during initial processing for a full record, the first byte of data is not a record separator character.

Decompression is performed if the function is specified in the *algorithm modifier*. This provides for converting strings of duplicate characters in compressed format in the source back to their full size in the receiver. Decompression of the source data is accomplished by concatenating together character strings described by the compression strings occurring in the source. The *source offset* value is assumed to locate the start of a compression string. Processing of a partial decompressed record is performed if necessary.

The character strings to be built into the receiver are described in the source by one or more compression strings. Compression strings are comprised of an SCB (string control byte) possibly followed by one or more bytes of data related to the character string to be built into the receiver.

The format of an SCB and the description of the data that may follow it is as follows:

- SCB — Char(1)
  - Control — Bits 0-1

    00 = n nonduplicate characters are between this SCB and the next one; where n is the value of the count field (1-63).
    01 = Reserved.
    10 = This SCB represents n deleted prime compression characters; where n is the value of the count field (1-63). The next byte is the next SCB.
    11 = This SCB represents n deleted duplicate characters; where n is the value of the count field (1-63). The next byte contains a specimen of the deleted characters. The byte following the specimen character contains the next SCB.
  - Count — Bits 2-7

    This contains the number of characters that have been deleted for a prime or duplicate string, or the number of characters to the next SCB for a nonduplicate string. A count value of zero is invalid and results in the signaling of a *conversion* (hex 0C01) exception.

Strings of prime compression characters or duplicate characters described in the source are repeated in the decompressed character string the number of times indicated by the *SCB count* value.

Strings of nonduplicate characters described in the source record are formed into a decompressed character string for the length indicated by the *SCB count* value.

If the end of the source is encountered prior to the end of a compression string, a *conversion* (hex 0C01) exception is signaled.

When *record processing* is specified, decompression is performed one record at a time. In this case, a *conversion* (hex 0C01) exception is signaled if a compression string describes a character string that would span a record boundary in the receiver. If the source contains record separators, the case of a missing record separator in the source is detected as defined under the initial description of the conversion process. *Record separator conversion*, as requested in the *algorithm modifier*, is performed as the initial step in the building of the decompressed record. A record separator to be placed into the receiver is in addition to the data to be converted into receiver for the length specified in the *receiver record length* field. The decompression of compression strings from the source continues until a *record separator character* for the next record is recognized when the source contains record separators, or until the decompressed data required to fill the receiver record has been processed or the end

of the source is encountered whether record separators are in the source or not. Transparency strings encountered in the decompressed character string are not scanned for a record separator value. If the end of the source is encountered, the data decompressed to that point appended to the optional record separator for this record forms a partial decompressed record. Otherwise, the decompressed character strings appended to the optional record separator for this record form the decompressed record. The conversion process then continues for this record with the next specified function.

When *string processing* is specified, decompression is performed on a compression string basis with no record oriented processing implied. The conversion process for each compression string from the source is completed by placing the decompressed character string into the receiver. The conversion process continues decompressing compression strings from the source until the end of the source or the receiver is encountered. When the end of the source operand is encountered, the instruction ends with a resultant condition of *source exhausted*. When a character string cannot be completely contained in the receiver, the instruction ends with a resultant condition of *receiver overrun*. For either of the above ending conditions, the controls operand is updated to describe the status of the conversion operation as of the last completely converted compression entry. Partial conversion of a compression entry is not performed.

Data transparency conversion is performed if *perform data transparency conversion* is specified in the *algorithm modifier*. This provides for correctly identifying record separators in the source even if the data for a record contains value that could be interpreted as record separator values. Processing of active transparency strings is performed if necessary.

A nontransparent record is built by appending the nontransparent and transparent data converted from the record to the record separator for the record. The nontransparent record may be produced from either a partial record from the source or a full record from the source. This is accomplished by first accessing the record separator for a full record. The case of a missing record separator in the source is detected as defined under the initial description of the conversion process. Record separator conversion as requested in the algorithm modifier is performed if it has not already been performed by a prior step; the rest of the source record is scanned for values of less than hex 40.

A value greater than or equal to hex 40 is considered nontransparent data and is concatenated onto the record being built as is.

A value equal to hex 35 identifies the start of a transparency string. A transparency string is comprised of 2 bytes of transparency control information followed by the data to be made transparent to scanning for record separators. The first byte has a fixed value of hex 35 and is referred to as the TRN (transparency) control character. The second byte is a 1-byte hexadecimal count, a value remaining from 1 to 255 decimal, of the number of bytes of data that follow and is referred to as the **TRN count**. A *TRN count* of zero is invalid and causes a *conversion* (hex 0C01) exception. This contains the length of the transparent data and does not include the TRN control information length. The transparent data is concatenated to the nontransparent record being built and is not scanned for record separator characters.

A value equal to hex 00 is recognized as the record separator for the next record only when perform decompression is specified in the algorithm modifier. In this case, the nontransparent record is complete. When do not perform decompression is specified in the algorithm modifier, a value equal to hex 00 is ignored and is not included as part of the nontransparent data built for the current record.

A value less than hex 40 but not equal to hex 35 is considered to be the record separator for the next record, and the forming of the nontransparent record is complete.

The building of the nontransparent record is completed when the length of the data converted into the receiver equals the receiver record length if the record separator for the next record is not encountered prior to that point.

If the end of the source is encountered prior to completion of building the nontransparent record, the nontransparent record built up to this point is placed in the receiver and the instruction ends with a resultant condition of *source exhausted*. The controls operand is updated to describe the status for the partially converted record. This includes describing a partially converted transparency string, if necessary, by setting the *active transparency string status* and the *unconverted transparency string bytes* field.

If the building of the nontransparent record is completed prior to encountering the end of the source, the conversion process continues with the blank padding function described below.

k padding is performed if the function is specified in the *algorithm modifier* This provides for expanding out to the size specified by the *receiver record length* the source records for which trailing blanks have been truncated. The padded record may be produced from either a partial record from the source or a full record from the source.

The record separator for this record is accessed. The case of a missing record separator in the source is detected as defined under the initial description of the conversion process. *Record separator conversion* as requested in the *algorithm modifier*, is performed if it has not already been performed by a prior step.

The nontruncated data, if any, for the record is appended to the optional record separator for the record. The nontruncated data is determined by scanning the source record for the record separator for the next record. This scan is concluded after processing enough data to completely fill the receiver record or upon encountering the record separator for the next record. The data processed prior to concluding the scan is considered the nontruncated data for the record.

The blanks, if any, required to pad the record out to the nontruncated data for the record, concluding the forming of the padded record.

If the end of the source is encountered during the forming of the padded record, the data processed up to that point, appended to the optional record separator for the record, is placed into the receiver and the instruction ends with a resultant condition of *source exhausted*. The controls operand is updated to describe the status of the partially converted record.

If the forming of the padded record is concluded prior to encountering the end of the source, the conversion of the record is completed by placing the converted form of the record into the receiver.

At this point, either conversion of a source record has been completed or conversion has been interrupted due to detection of the *source exhausted* or *receiver overrun* condition. For *record processing*, if neither of the above conditions has been detected either during conversion of or at completion of conversion for the current record, the conversion process continues on the next source record with the decompression function described above.

At completion of the instruction, the *receiver offset* locates the byte following the last converted byte in the receiver. The value of the remaining bytes in the receiver after the last converted byte are unpredictable. The *source offset* locates the byte following the last source byte for which conversion was completed. When record processing is specified, the *unconverted receiver record bytes* field specifies the length of the receiver record bytes not yet containing converted data. When *perform data transparency conversion* is specified in the *algorithm modifier*, the *conversion status* indicates whether conversion of a transparency string was active and the *unconverted transparency string bytes* field specifies the length of the remaining bytes to be processed for an active transparency string.

This instruction does not provide support for compression entries in the source describing data that would span records in the receiver. SNA data from some systems may violate this restriction and as such be incompatible with the instruction. A provision can be made to avoid this incompatibility by performing the conversion of the SNA data through two invocations of this instruction. The first invoca-

tion would specify *decompression* with *no record separator processing*. The second invocation would specify *record separator processing* with no *decompression*. This technique provides for separating the decompression step from record separator processing; thus, the incompatibility is avoided.

This instruction can end with the *escape code encountered* condition. In this case, it is expected that the user of the instruction will want to do some special processing for the record separator causing the condition. In order to resume execution of the instruction, the user will have to set the appropriate value for the *record separator* into the receiver and update the controls operand *source offset* and *receiver offset* fields correctly to provide for restarting processing at the right points in the receiver and source operands.

For the special case of a tie between the *source exhausted* and *receiver overrun* conditions, the *source exhausted* condition is recognized first. That is, when *source exhausted* is the resultant condition, the receiver may also be full. In this case, the *receiver offset* may contain a value equal to the length specified for the receiver, which would cause an exception to be detected on the next invocation of the instruction. The processing performed for the *source exhausted* condition should provide for this case if the instruction is to be invoked multiple times with the same controls operand template. When the *receiver overrun* condition is the resultant condition, the source will always contain data that can be converted.

This instruction will, in certain cases, ignore what would normally have been interpreted as a record separator value of hex 00. This applies (hex 00 is ignored) for the special case when *do not perform decompression* and *record separators in source* are specified in the *algorithm modifier*. Note that this does not apply when *perform decompression* is specified, or when *do not perform decompression* and *no record separators in source* and *move record separator from controls to receiver* are specified in the *algorithm modifier*.

Limitations: The following are limits that apply to the functions performed by this instruction.

Any form of overlap between the operands on this instruction yields unpredictable results in the receiver operand.

Resultant Conditions

- Source exhausted-The end of the source operand is encountered and no more bytes from the source can be converted.
- Receiver overrun-An overrun condition in the receiver operand is detected before all of the bytes in the source operand have been processed.
- Escape code encountered-A record separator character is encountered in the source operand that is to be treated as an escape code.

Exceptions

Exception		Operands 1	2	3	Other
06	Addressing				
	01 Spacing addressing violation	X	X	X	
	02 Boundary alignment violation	X	X	X	
	03 Range	X	X	X	
	06 Optimized addressability invalid	X	X	X	
08	Argument/parameter				
	01 Parameter reference violation	X	X	X	

Exception		Operands 1	2	3	Other
0C	Computation				
	01 Conversion			X	
10	Damage encountered				
	04 System object damage state				X
	44 Partial system object damage				X
1C	Machine-dependent exception				
	03 Machine storage limit exceeded				X
20	Machine support				
	02 Machine check				X
	03 Function check				X
22	Object access				
	01 Object not found	X	X	X	
	02 Object destroyed	X	X	X	
	03 Object suspended	X	X	X	
	08 object compressed				X
24	Pointer specification				
	01 Pointer does not exist	X	X	X	
	02 Pointer type invalid	X	X	X	
2C	Program execution				
	04 Invalid branch target				X
2E	Resource control limit				
	01 user profile storage limit exceeded				X
32	Scalar specification				
	01 Scalar type invalid	X	X	X	
36	Space management				
	01 space extension/truncation				X
38	Template specification				
	01 Template value invalid		X		

Copy Bits Arithmetic (CPYBTA)

Op Code (Hex)	Operand 1	Operand 2	Operand 3	Operand 4
102C	Receiver	Source	Offset	Length

Operand 1: Character variable scalar (fixed length) or numeric variable scalar.

Operand 2: Character variable scalar (fixed length) or numeric variable scalar.

Operand 3: Signed or unsigned binary immediate.

Operand 4: Signed or unsigned binary immediate.

Description: This instruction copies the signed bit string source operand starting at the specified offset for a specified length right adjusted to the receiver and pads on the left with the sign of the bit string source.

The selected bits from the source operand are treated as an signed bit string and copied to the receiver value.

The source operand can be character or numeric. The leftmost bytes of the source operand are used in the operation. The source operand is interpreted as a bit string with the bits numbered left to right from 0 to the total number of bits in the string minus 1.

The offset operand indicates which bit of the source operand is to be copied; with a offset of zero indicating the leftmost bit of the leftmost byte of the source operand.

The length operand indicates the number of bits that are to be copied.

If the sum of the offset plus the length exceed the length of the source an *invalid operand length* (hex 2A0A) exception will be raised.

Limitations: The length of the receiver cannot exceed four bytes.

The offset must have a non-negative value.

The length operand must be an immediate value between 1 and 32.

Exceptions

Exception		Operands 1	2	3	4	Other
06	Addressing					
	01 Spacing addressing violation	X	X			
	02 Boundary alignment violation	X	X			
	03 Range	X	X			
	06 Optimized addressability invalid	X	X			
08	Argument/parameter					
	01 Parameter reference violation	X	X			
10	Damage encountered					
	04 System object damage state					X
	44 Partial system object damage					X

Exception		Operands 1	2	3	4	Other
1C	Machine-dependent exception					
	03 Machine storage limit exceeded					X
20	Machine support					
	02 Machine check					X
	03 Function check					X
22	Object access					
	02 Object destroyed	X	X			
	03 Object suspended	X	X			
	08 object compressed					X
24	Pointer specification					
	01 Pointer does not exist	X	X			
	02 Pointer type invalid	X	X			
2E	Resource control limit					
	01 user profile storage limit exceeded					X
36	Space management					
	01 space extension/truncation					X

Copy Bits Logical (CPYBTL)

Op Code (Hex)	Operand 1	Operand 2	Operand 3	Operand 4
101C	Receiver	Source	Offset	Length

Operand 1: Character variable scalar (fixed length) or numeric variable scalar.

Operand 2: Character variable scalar (fixed length) or numeric variable scalar.

Operand 3: Signed or unsigned binary immediate.

Operand 4: Signed or unsigned binary immediate.

Description: Copies the unsigned bit string source operand starting at the specified offset for a specified length to the receiver.

If the receiver is shorter than the length, the left most bits are removed to make the source bit string conform to the length of the receiver. No exceptions are generated when truncation occurs.

The selected bits from the source operand are treated as an unsigned bit string and copied right adjusted to the receiver and padded on the left with binary zeros.

The source operand can be character or numeric. The leftmost bytes of the source operand are used in the operation. The source operand is interpreted as a bit string with the bits numbered left to right from 0 to the total number of bits in the string minus 1.

The offset operand indicates which bit of the source operand is to be copied, with a offset of zero indicating the leftmost bit of the leftmost byte of the source operand.

The length operand indicates the number of bits that are to be copied.

If the sum of the offset plus the length exceed the length of the source an *invalid operand length* (hex 2A0A) exception will be signaled.

Limitations: The length of the receiver cannot exceed four bytes.

The offset must have a non-negative value.

The length operand must be an immediate value between 1 and 32.

Exceptions

Exception		Operands 1	2	3	4	Other
06	Addressing					
	01 Spacing addressing violation	X	X			
	02 Boundary alignment violation	X	X			
	03 Range	X	X			
	06 Optimized addressability invalid	X	X			
08	Argument/parameter					
	01 Parameter reference violation	X	X	X	X	
10	Damage encountered					

Exception		Operands 1	2	3	4	Other
	04 System object damage state					X
	44 Partial system object damage					X
1C	Machine-dependent exception					
	03 Machine storage limit exceeded					X
20	Machine support					
	02 Machine check					X
	03 Function check					X
22	Object access					
	02 Object destroyed	X	X			
	03 Object suspended	X	X			
	08 object compressed					X
24	Pointer specification					
	01 Pointer does not exist	X	X			
	02 Pointer type invalid	X	X			
2E	Resource control limit					
	01 user profile storage limit exceeded					X
36	Space management					
	01 space extension/truncation					X

Copy Bits with Left Logical Shift (CPYBTLLS)

Op Code (Hex)	Operand 1	Operand 2	Operand 3
102F	Receiver	Source	Shift control

Operand 1: Character variable scalar or numeric variable scalar.

Operand 2: Character scalar or numeric scalar.

Operand 3: Character(2) scalar (fixed length) or unsigned binary(2) scalar.

Description: This instruction copies the bit string value of the source operand to the bit string defined by the receiver operand with a left logical shift of the source bit string value under control of the shift control operand.

The operation results in copying the shifted bit string value of the source to the bit string of the receiver while padding the receiver with bit values of 0 and truncating bit values of the source as is appropriate for the specific operation.

No indication is given of truncation of bit values from the shifted source value. This is true whether the values truncated are 0 or 1.

The operation is performed such that the bit string of the source is considered to be extended on the left and right by an unlimited number of bit string positions of value 0. Additionally, a receiver bit string view (window) with the attributes of the receiver is considered to overlay this conceptual bit string value of the source starting at the leftmost bit position of the original source value. A left logical shift of the conceptual bit string value of the source is then performed relative to the receiver bit string view according to the shift criteria specified in the shift control operand. After the shift, the bit string value then contained within the receiver bit string view is copied to the receiver.

The source and the receiver can be either character or numeric. Any numeric operands are interpreted as logical character strings. Due to the operation being treated as a character string operation, the source operand may not be specified as a signed immediate operand. Additionally, for a source operand specified as an unsigned immediate value, only a 1-byte immediate value may be specified.

The shift control operand may be specified as an immediate operand, as a character(2) scalar, or as an unsigned binary(2) scalar. It provides an unsigned binary value indicating the number of bit positions for which the left logical shift of the source bit string value is to be performed. A zero value specifies no shift.

Exceptions

Exception		Operands 1	2	3	Other
06	Addressing				
	01 Spacing addressing violation	X	X	X	
	02 Boundary alignment violation	X	X	X	
	03 Range	X	X	X	
	06 Optimized addressability invalid	X	X	X	
08	Argument/parameter				
	01 Parameter reference violation	X	X	X	
10	Damage encountered				

Exception		Operands 1	2	3	Other
	04 System object damage state				X
	44 Partial system object damage				X
1C	Machine-dependent exception				
	03 Machine storage limit exceeded				X
20	Machine support				
	02 Machine check				X
	03 Function check				X
22	Object access				
	02 Object destroyed	X	X	X	
	03 Object suspended	X	X	X	
	08 object compressed				X
24	Pointer specification				
	01 Pointer does not exist	X	X	X	
	02 Pointer type invalid	X	X	X	
2E	Resource control limit				
	01 user profile storage limit exceeded				X
32	Scalar specification				
	01 Scalar type invalid	X	X	X	
36	Space management				
	01 space extension/truncation				X

Copy Bits with Right Arithmetic Shift (CPYBTRAS)

Op Code (Hex)	Operand 1	Operand 2	Operand 3
101B	Receiver	Source	Shift Control

Operand 1: Character variable or numeric variable scalar.

Operand 2: Character scalar or numeric scalar.

Operand 3: Character(2) scalar (fixed length) or unsigned binary(2) scalar.

Description: The instruction copies the bit string value of the source operand to the bit string defined by the receiver operand with a right arithmetic shift of the source bit string value under control of the shift control operand.

The operation results in copying the shifted bit string value of the source to the bit string of the receiver while padding the receiver with bit values of 0 or 1 depending on the high order bit value of the source, and truncating bit values of the source as is appropriate for the specific operation.

No indication is given of truncation of bit values from the shifted source value. This is true whether the values truncated are 0 or 1.

The operation is performed such that the bit string of the source is considered a signed numeric binary value, with the value of the sign bit of the source conceptually extended on the left an unlimited number of bit string positions. A right arithmetic shift of the conceptual bit string value of the source is then performed according to the shift criteria specified in the shift control operand. No indication is given of truncation of bit values from the shifted conceptual source value. This is true whether the values truncated are 0 or 1. After the shift, the conceptual bit string value is then copied to the receiver, right aligned.

Viewing the bit string value of the source and the bit string value copied to the receiver as signed numeric, the sign of the value copied to the receiver will be the same as the sign of the source.

A right shift of one bit position is equivalent to dividing the signed numeric bit string value of the source by 2 with rounding downward, and assigning a signed numeric bit string equivalent to that result to the receiver. For example, if the signed numeric view of the source bit string is +9, shifting one bit position right yields +4. However if the signed numeric view of the source bit string is -9, shifting one bit position right yields -5.

If all the significant bits of the conceptual source bit string are shifted out of the field, the resulting conceptual bit string value will be all zero bits for positive numbers, and all one bits for negative numbers.

The source and the receiver can be either character or numeric. Any numeric operands are interpreted as logical character strings. Due to the operation being treated as a character string operation, the source operand may not be specified as a signed immediate operand. Additionally, for a source operand specified as an unsigned immediate value, only a 1-byte immediate value may be specified.

The shift control operand may be specified as an immediate operand, as a character(2) scalar, or as a unsigned binary(2) scalar. It provides an unsigned binary value indicating the number of bit positions for which the right logical shift of the source bit string value is to be performed. A zero value specifies no shift.

Exceptions

Exception		Operands 1	2	3	Other
06	Addressing				
	01 Spacing addressing violation	X	X	X	
	02 Boundary alignment violation	X	X	X	
	03 Range	X	X	X	
	06 Optimized addressability invalid	X	X	X	
08	Argument/parameter				
	01 Parameter reference violation	X	X	X	
10	Damage encountered				
	04 System object damage state				X
	44 Partial system object damage				X
1C	Machine-dependent exception				
	03 Machine storage limit exceeded				X
20	Machine support				
	02 Machine check				X
	03 Function check				X
22	Object access				
	02 Object destroyed	X	X	X	
	03 Object suspended	X	X	X	
	08 object compressed				X
24	Pointer specification				
	01 Pointer does not exist	X	X	X	
	02 Pointer type invalid	X	X	X	
2E	Resource control limit				
	01 user profile storage limit exceeded				X
32	Scalar specification				
	01 Scalar type invalid	X	X	X	
36	Space management				
	01 space extension/truncation				X

Copy Bits with Right Logical Shift (CPYBTRLS)

Op Code (Hex)	Operand 1	Operand 2	Operand 3
103F	Receiver	Source	Shift control

Operand 1: Character variable scalar or numeric variable scalar.

Operand 2: Character scalar or numeric scalar.

Operand 3: Character(2) scalar (fixed length) or unsigned binary(2) scalar.

Description: This instruction copies the bit string value of the source operand to the bit string defined by the receiver operand with a right logical shift of the source bit string value under control of the shift control operand.

The operation results in copying the shifted bit string value of the source to the bit string of the receiver while padding the receiver with bit values of 0 and truncating bit values of the source as is appropriate for the specific operation.

No indication is given of truncation of bit values from the shifted source value. This is true whether the values truncated are 0 or 1.

The operation is performed such that the bit string of the source is considered to be extended on the left and right by an unlimited number of bit string positions of value 0. Additionally, a receiver bit string view (window) with the attributes of the receiver is considered to overlay this conceptual bit string value of the source starting at the leftmost bit position of the original source value. A right logical shift of the conceptual bit string value of the source is then performed relative to the receiver bit string view according to the shift criteria specified in the shift control operand. After the shift, the bit string value then contained within the receiver bit string view is copied to the receiver.

The source and the receiver can be either character or numeric. Any numeric operands are interpreted as logical character strings. Due to the operation being treated as a character string operation, the source operand may not be specified as a signed immediate operand. Additionally, for a source operand specified as an unsigned immediate value, only a 1-byte immediate value may be specified.

The shift control operand may be specified as an immediate operand, as a character(2) scalar, or as a unsigned binary(2) scalar. It provides an unsigned binary value indicating the number of bit positions for which the right logical shift of the source bit string value is to be performed. A zero value specifies no shift.

Exceptions

Exception		Operands 1	2	3	Other
06	Addressing				
	01 Spacing addressing violation	X	X	X	
	02 Boundary alignment violation	X	X	X	
	03 Range	X	X	X	
	06 Optimized addressability invalid	X	X	X	
08	Argument/parameter				
	01 Parameter reference violation	X	X	X	
10	Damage encountered				

Exception		Operands 1	2	3	Other
	04 System object damage state				X
	44 Partial system object damage				X
1C	Machine-dependent exception				
	03 Machine storage limit exceeded				X
20	Machine support				
	02 Machine check				X
	03 Function check				X
22	Object access				
	02 Object destroyed	X	X	X	
	03 Object suspended	X	X	X	
	08 object compressed				X
24	Pointer specification				
	01 Pointer does not exist	X	X	X	
	02 Pointer type invalid	X	X	X	
2E	Resource control limit				
	01 user profile storage limit exceeded				X
32	Scalar specification				
	01 Scalar type invalid	X	X	X	
36	Space management				
	01 space extension/truncation				X

Copy Bytes Left-Adjusted (CPYBLA)

Op Code (Hex)	Operand 1	Operand 2
10B2	Receiver	Source

Operand 1: Character variable scalar, numeric variable scalar, data-pointer-defined character scalar, or data-pointer-defined numeric scalar.

Operand 2: Character scalar, numeric scalar, data-pointer-defined character scalar, or data-pointer-defined numeric scalar.

Description: The logical string value of the source operand is copied to the logical string value of the receiver operand (no padding done). The operands can be either character or numeric. Any numeric operands are interpreted as logical character strings.

The length of the operation is equal to the length of the shorter of the two operands. The copying begins with the two operands left-adjusted and proceeds until the shorter operand has been copied.

Substring operand references that allow for a null substring reference (a length value of zero) may be specified for operands 1 and 2. The effect of specifying a null substring reference for either operand is that no result is set.

If either operand is a character variable scalar, it may have a length as great as 16776191 bytes.

Exceptions

Exception		Operands 1	Operands 2	Other
06	Addressing			
	01 Spacing addressing violation	X	X	
	02 Boundary alignment	X	X	
	03 Range	X	X	
	04 External data object not found	X	X	
	06 Optimized addressability invalid	X	X	
08	Argument/parameter			
	01 Parameter reference violation	X	X	
10	Damage encountered			
	04 System object damage state	X	X	X
	44 Partial system object damage	X	X	X
1C	Machine-dependent exception			
	03 Machine storage limit exceeded			X
20	Machine support			
	02 Machine check			X
	03 Function check			X
22	Object access			
	01 Object not found	X	X	
	02 Object destroyed	X	X	

Exception		Operands 1	Operands 2	Other
	03 Object suspended	X	X	
	08 object compressed			X
24	Pointer specification			
	01 Pointer does not exist	X	X	
	02 Pointer type invalid	X	X	
2E	Resource control limit			
	01 user profile storage limit exceeded			X
32	Scalar specification			
	01 Scalar type invalid	X	X	
36	Space management			
	01 space extension/truncation			X

Copy Bytes Left-Adjusted with Pad (CPYBLAP)

Op Code (Hex)	Operand 1	Operand 2	Operand 3
10B3	Receiver	Source	Pad

Operand 1: Character variable scalar or numeric variable scalar, data-pointer-defined character scalar, or data-pointer-defined numeric scalar.

Operand 2: Character scalar, numeric scalar, data-pointer-defined character scalar, or data-pointer-defined numeric scalar.

Operand 3: Character scalar or numeric scalar.

Description: The logical string value of the source operand is copied to the logical string value of the receiver operand (padded if needed).

The operands can be either character or numeric. Any numeric operands are interpreted as logical character strings.

The length of the operation is equal to the length of the receiver operand. If the source operand is shorter than the receiver operand, the source operand is copied to the leftmost bytes of the receiver operand, and each excess byte of the receiver operand is assigned the single byte value in the pad operand. If the pad operand is more than 1 byte in length, only its leftmost byte is used. If the source operand is longer than the receiver operand, the leftmost bytes of the source operand (equal in length to the receiver operand) are copied to the receiver operand.

Substring operand references that allow for a null substring reference (a length value of zero) may be specified for operands 1 and 2. The effect of specifying a null substring reference for the source is that the bytes of the receiver are each set with the single byte value of the pad operand. The effect of specifying a null substring reference for the receiver is that no result is set.

If either of the first two operands is a character variable scalar, it may have a length as great as 16776191.

Exceptions

Exception		Operands 1	2	3	Other
06	Addressing				
	01 Spacing addressing violation	X	X	X	
	02 Boundary alignment	X	X	X	
	03 Range	X	X	X	
	04 External data object not found	X	X		
	06 Optimized addressability invalid	X	X	X	
08	Argument/parameter				
	01 Parameter reference violation	X	X	X	
10	Damage encountered				
	04 System object damage state	X	X	X	X
	44 Partial system object damage	X	X	X	X
1C	Machine-dependent exception				

Exception		Operands 1	2	3	Other
	03 Machine storage limit exceeded				X
20	Machine support				
	02 Machine check				X
	03 Function check				X
22	Object access				
	01 Object not found	X	X	X	
	02 Object destroyed	X	X	X	
	03 Object suspended	X	X	X	
	08 object compressed				X
24	Pointer specification				
	01 Pointer does not exist	X	X	X	
	02 Pointer type invalid	X	X	X	
2E	Resource control limit				
	01 user profile storage limit exceeded				X
32	Scalar specification				
	01 Scalar type invalid	X	X		
36	Space management				
	01 space extension/truncation				X

Copy Bytes Overlap Left-Adjusted (CPYBOLA)

Op Code (Hex)	Operand 1	Operand 2
10BA	Receiver	Source

Operand 1: Character variable scalar or numeric variable scalar.

Operand 2: Character variable scalar or numeric variable scalar.

Description: The logical string value of the source operand is copied to the logical string value of the receiver operand (no padding done). The operands can be either character or numeric. Any numeric operands are interpreted as logical character strings.

The length of the operation is equal to the length of the shorter of the two operands. The copying begins with the two operands left-adjusted and proceeds until the shorter operand has been copied. The excess bytes in the longer operand are not included in the operation.

Predictable results occur even if two operands overlap because the source operand is, in effect, first copied to an intermediate result.

Substring operand references that allow for a null substring reference (a length value of zero) may be specified for operands 1 and 2. The effect of specifying a null substring reference for either operand is that no result is set.

Exceptions

Exception		Operands 1	Operands 2	Other
06	Addressing			
	01 Spacing addressing violation	X	X	
	02 Boundary alignment	X	X	
	03 Range	X	X	
	06 Optimized addressability invalid	X	X	
08	Argument/parameter			
	01 Parameter reference violation	X	X	
10	Damage encountered			
	04 System object damage state	X	X	X
	44 Partial system object damage	X	X	X
1C	Machine-dependent exception			
	03 Machine storage limit exceeded			X
20	Machine support			
	02 Machine check			X
	03 Function check			X
22	Object access			
	01 Object not found	X	X	
	02 Object destroyed	X	X	
	03 Object suspended	X	X	

Exception		Operands 1	2	Other
	08 object compressed			X
24	Pointer specification			
	01 Pointer does not exist	X	X	
	02 Pointer type invalid	X	X	
2E	Resource control limit			
	01 user profile storage limit exceeded			X
32	Scalar specification			
	01 Scalar type invalid	X	X	
36	Space management			
	01 space extension/truncation			X

Copy Bytes Overlap Left-Adjusted with Pad (CPYBOLAP)

Op Code (Hex)	Operand 1	Operand 2	Operand 3
10BB	Receiver	Source	Pad

Operand 1: Character variable scalar or numeric variable scalar.

Operand 2: Character variable scalar or numeric variable scalar.

Operand 3: Character scalar or numeric scalar.

Description: The logical string value of the source operand is copied to the logical string value of the receiver operand.

The operands can be either character or numeric. Any numeric operands are interpreted as logical character strings.

The length of the operation is equal to the length of the receiver operand. If the source operand is shorter than the receiver operand, the source operand is copied to the leftmost bytes of the receiver operand and each excess byte of the receiver operand is assigned the single byte value in the pad operand. If the pad operand is more than 1 byte in length, only its leftmost byte is used. If the source operand is longer than the receiver operand, the leftmost bytes of the source operand (equal in length to the receiver operand) are copied to the receiver operand.

Predictable results occur even if two operands overlap because the source operand is, in effect, first copied to an intermediate result.

Substring operand references that allow for a null substring reference (a length value of zero) may be specified for operands 1 and 2. The effect of specifying a null substring reference for the source is that the bytes of the receiver are each set with the single byte value of the pad operand. The effect of specifying a null substring reference for the receiver is that no result is set.

Exceptions

Exception		Operands 1	2	3	Other
06	Addressing				
	01 Spacing addressing violation	X	X	X	
	02 Boundary alignment	X	X	X	
	03 Range	X	X	X	
	06 Optimized addressability invalid	X	X	X	
08	Argument/parameter				
	01 Parameter reference violation	X	X	X	
10	Damage encountered				
	04 System object damage state	X	X	X	X
	44 Partial system object damage	X	X	X	X
1C	Machine-dependent exception				
	03 Machine storage limit exceeded			X	X
20	Machine support				
	02 Machine check				X

Exception		Operands 1	2	3	Other
	03 Function check				X
22	Object access				
	01 Object not found	X	X	X	
	02 Object destroyed	X	X	X	
	03 Object suspended	X	X	X	
	08 object compressed				X
24	Pointer specification				
	01 Pointer does not exist	X	X	X	
	02 Pointer type invalid	X	X	X	
2E	Resource control limit				
	01 user profile storage limit exceeded				X
36	Space management				
	01 space extension/truncation				X

Copy Bytes Repeatedly (CPYBREP)

Op Code (Hex)	Operand 1	Operand 2
10BE	Receiver	Source

Operand 1: Numeric variable scalar or character variable scalar (fixed-length).

Operand 2: Numeric scalar or character scalar (fixed length).

Description: The logical string value of the source operand is repeatedly copied to the receiver operand until the receiver is filled. The operands can be either character or numeric. Any numeric operands are interpreted as logical character strings.

The operation begins with the two operands left-adjusted and continues until the receiver operand is completely filled. If the source operand is shorter than the receiver, it is repeatedly copied from left to right (all or in part) until the receiver operand is completely filled. If the source operand is longer than the receive operand, the leftmost bytes of the source operand (equal in length to the receiver operand) are copied to the receiver operand.

Substring operand references that allow for a null substring reference (a length value of zero) may be specified for operands 1 and 2. The effect of specifying a null substring reference for either operand is that no result is set.

If either operand is a character variable scalar, it may have a length as great as 16776191.

Exceptions

Exception		Operands 1	Operands 2	Other
06	Addressing			
	01 Spacing addressing violation	X	X	
	02 Boundary alignment	X	X	
	03 Range	X	X	
	06 Optimized addressability invalid	X	X	
08	Argument/parameter			
	01 Parameter reference violation	X	X	
10	Damage encountered			
	04 System object damage state	X	X	X
	44 Partial system object damage	X	X	X
1C	Machine-dependent exception			
	03 Machine storage limit exceeded			X
20	Machine support			
	02 Machine check			X
	03 Function check			X
22	Object access			
	01 Object not found	X	X	
	02 Object destroyed	X	X	
	03 Object suspended	X	X	

Exception		Operands 1	Operands 2	Other
	08 object compressed			X
24	Pointer specification			
	01 Pointer does not exist	X	X	
	02 Pointer type invalid	X	X	
2E	Resource control limit			
	01 user profile storage limit exceeded			X
36	Space management			
	01 space extension/truncation			X

Copy Bytes Right-Adjusted (CPYBRA)

Op Code (Hex)	Operand 1	Operand 2
10B6	Receiver	Source

Operand 1: Character variable scalar, numeric variable scalar, data-pointer-defined character scalar, or data-pointer-defined numeric scalar.

Operand 2: Character scalar, numeric scalar, data-pointer-defined character scalar, or data-pointer-defined numeric scalar.

Description: The logical string value of the source operand is copied to the logical string value of the receiver operand (no padding done). The operands can be either character or numeric. Any numeric operands are interpreted as logical character strings.

The length of the operation is equal to the length of the shorter of the two operands. The rightmost bytes (equal to the length of the shorter of the two operands) of the source operand are copied to the rightmost bytes of the receiver operand. The excess bytes in the longer operand are not included in the operation.

Substring operand references that allow for a null substring reference (a length value of zero) may be specified for operands 1 and 2. The effect of specifying a null substring reference for either operand is that no result is set.

Exceptions

Exception		Operands 1	Operands 2	Other
06	Addressing			
	01 Spacing addressing violation	X	X	
	02 Boundary alignment	X	X	
	03 Range	X	X	
	04 External data object not found	X	X	
	06 Optimized addressability invalid	X	X	
08	Argument/parameter			
	01 Parameter reference violation	X	X	
10	Damage encountered			
	04 System object damage state	X	X	X
	44 Partial system object damage	X	X	X
1C	Machine-dependent exception			
	03 Machine storage limit exceeded			X
20	Machine support			
	02 Machine check			X
	03 Function check			X
22	Object access			
	01 Object not found	X	X	
	02 Object destroyed	X	X	

Exception		Operands 1	Operands 2	Other
	03 Object suspended	X	X	
	08 object compressed			X
24	Pointer specification			
	01 Pointer does not exist	X	X	
	02 Pointer type invalid	X	X	
2E	Resource control limit			
	01 user profile storage limit exceeded			X
32	Scalar specification			
	01 Scalar type invalid	X	X	
36	Space management			
	01 space extension/truncation			X

Copy Bytes Right-Adjusted with Pad (CPYBRAP)

Op Code (Hex)	Operand 1	Operand 2	Operand 3
10B7	Receiver	Source	Pad

Operand 1: Character variable scalar, numeric variable scalar, data-pointer-defined character scalar, or data-pointer-defined numeric scalar.

Operand 2: Character scalar, numeric scalar, data-pointer-defined character scalar, or data-pointer-defined numeric scalar.

Operand 3: Character scalar or numeric scalar.

Description: The logical string value of the source operand is copied to the logical string value of the receiver operand (padded if needed). The operands can be either character or numeric. Any numeric operands are interpreted as logical character strings.

The length of the operation is equal to the length of the receiver operand. If the source operand is shorter than the receiver operand, the source operand is copied to the rightmost bytes of receiver operand, and each excess byte is assigned the single byte value in the pad operand. If the pad operand is more than 1 byte in length, only its leftmost byte is used. If the source operand is longer than the receiver operand, the rightmost bytes of the source operand (equal in length to the receiver operand) are copied to the receiver operand.

Substring operand references that allow for a null substring reference (a length value of zero) may be specified for operands 1 and 2. The effect of specifying a null substring reference for the source is that the bytes of the receiver are each set with the single byte value of the pad operand. The effect of specifying a null substring reference for the receiver is that no result is set.

Exceptions

Exception		Operands 1	2	3	Other
06	Addressing				
	01 Spacing addressing violation	X	X	X	
	02 Boundary alignment	X	X	X	
	03 Range	X	X	X	
	04 External data object not found	X	X		
	06 Optimized addressability invalid	X	X	X	
08	Argument/parameter				
	01 Parameter reference violation	X	X	X	
10	Damage encountered				
	04 System object damage state	X	X	X	X
	44 Partial system object damage	X	X	X	X
1C	Machine-dependent exception				
	03 Machine storage limit exceeded				X
20	Machine support				
	02 Machine check				X
	03 Function check				X

Exception		Operands 1	2	3	Other
22	Object access				
	01 Object not found	X	X	X	
	02 Object destroyed	X	X	X	
	03 Object suspended	X	X	X	
	08 object compressed				X
24	Pointer specification				
	01 Pointer does not exist	X	X	X	
	02 Pointer type invalid	X	X	X	
2E	Resource control limit				
	01 user profile storage limit exceeded				X
32	Scalar specification				
	01 Scalar type invalid	X	X		
36	Space management				
	01 space extension/truncation				X

Copy Bytes to Bits Arithmetic (CPYBBTA)

Op Code (Hex)	Operand 1	Operand 2	Operand 3	Operand 4
104C	Receiver	Offset	Length	Source

Operand 1: Character variable scalar (fixed length) or numeric variable scalar.

Operand 2: Signed binary immediate or unsigned binary immediate.

Operand 3: Signed binary immediate or unsigned binary immediate.

Operand 4: Character variable scalar (fixed length) or numeric variable scalar.

Description: This instruction copies a byte string from the source operand to a bit string in the receiver operand.

The source operand is interpreted as a signed binary value and may be sign extended or truncated on the left to fit into the bit string in the receiver operand. No indication is given when truncation occurs.

The location of the bit string in the receiver operand is specified by the offset operand. The value of the offset operand specifies the bit offset from the start of the receiver operand to the start of the bit string. Thus, an offset operand value of 0 specifies that the bit string starts at the leftmost bit position of the receiver operand.

The length of the bit string in the receiver operand is specified by the length operand. The value of the length operand specifies the length of the bit string in bits.

Limitations: The following are limits that apply to the functions performed by this instruction.

If the source operand and the bit string in the receiver operand overlap, the results are unpredictable.

A source operand longer than 4 bytes may not be specified.

If the offset operand is signed binary immediate, a negative value may not be specified.

A length operand with a value less than 1 or greater than 32 may not be specified.

The bit string specified by the offset operand and the length operand may not extend outside the receiver operand.

Exceptions

Exception		Operands 1	2	3	4	Other
06	Addressing					
	01 Spacing addressing violation	X			X	
	02 Boundary alignment violation	X			X	
	03 Range	X			X	
	06 Optimized addressability invalid	X			X	
08	Argument/parameter					
	01 Parameter reference violation	X			X	
10	Damage encountered					

Exception		Operands 1	2	3	4	Other
	04 System object damage state					X
	44 Partial system object damage					X
1C	Machine-dependent exception					
	03 Machine storage limit exceeded					X
20	Machine support					
	02 Machine check					X
	03 Function check					X
22	Object access					
	02 Object destroyed	X			X	
	03 Object suspended	X			X	
	08 object compressed					X
24	Pointer specification					
	01 Pointer does not exist	X			X	
	02 Pointer type invalid	X			X	
2E	Resource control limit					
	01 user profile storage limit exceeded					X
36	Space management					
	01 space extension/truncation					X

Copy Bytes to Bits Logical (CPYBBTL)

Op Code (Hex)	Operand 1	Operand 2	Operand 3	Operand 4
103C	Receiver	Offset	Length	Source

Operand 1: Character variable scalar (fixed length) or numeric variable scalar.

Operand 2: Signed binary immediate or unsigned binary immediate.

Operand 3: Signed binary immediate or unsigned binary immediate.

Operand 4: Character variable scalar (fixed length) or numeric variable scalar.

Description: This instruction copies a byte string from the source operand to a bit string in the receiver operand.

The source operand is interpreted as an unsigned binary value and may be padded on the left with 0's or truncated on the left to fit into the bit string in the receiver operand. No indication is given when truncation occurs.

The location of the bit string in the receiver operand is specified by the offset operand. The value of the offset operand specifies the bit offset from the start of the receiver operand to the start of the bit string. Thus, an offset operand value of 0 specifies that the bit string starts at the leftmost bit position of the receiver operand.

The length of the bit string in the receiver operand is specified by the length operand. The value of the length operand specifies the length of the bit string in bits.

Limitations: The following are limits that apply to the functions performed by this instruction.

If the source operand and the bit string in the receiver operand overlap, the results are unpredictable.

A source operand longer than 4 bytes may not be specified.

If the offset operand is signed binary immediate, a negative value may not be specified.

A length operand with a value less than 1 or greater than 32 may not be specified.

The bit string specified by the offset operand and the length operand may not extend outside the receiver operand.

Exceptions

Exception		Operands 1	2	3	4	Other
06	Addressing					
	01 Spacing addressing violation	X			X	
	02 Boundary alignment violation	X			X	
	03 Range	X			X	
	06 Optimized addressability invalid	X			X	
08	Argument/parameter					
	01 Parameter reference violation	X			X	
10	Damage encountered					

Exception		Operands 1	2	3	4	Other
	04 System object damage state					X
	44 Partial system object damage					X
1C	Machine-dependent exception					
	03 Machine storage limit exceeded					X
20	Machine support					
	02 Machine check					X
	03 Function check					X
22	Object access					
	02 Object destroyed	X			X	
	03 Object suspended	X			X	
	08 object compressed					X
24	Pointer specification					
	01 Pointer does not exist	X			X	
	02 Pointer type invalid	X			X	
2E	Resource control limit					
	01 user profile storage limit exceeded					X
36	Space management					
	01 space extension/truncation					X

Copy Extended Characters Left-Adjusted With Pad (CPYECLAP)

Op Code (Hex)	Operand 1	Operand 2	Operand 3
1053	Receiver	Source	Pad

Operand 1: Data-pointer-defined character scalar.

Operand 2: Data-pointer-defined character scalar.

Operand 3: Character(3) scalar or null.

Description: The extended character string value of the source operand is copied to the receiver operand.

The operation is performed at the length of the receiver operand. If the source operand is shorter than the receiver, the source operand is copied to the leftmost bytes of the receiver and the excess bytes of the receiver are assigned the appropriate value from the pad operand.

The pad operand, operand 3, is three bytes in length and has the following format:

- Pad operand — Char(3)
  - Single byte pad value — Char(1)
  - Double byte pad value — Char(2)

If the **pad operand** is more than three bytes in length, only its leftmost three bytes are used. Specifying a null *pad operand* results in default pad values of hex 40, for single byte, and hex 4040, for double byte, being used. The **single byte pad value** and the first byte of the **double byte pad value** cannot be either a shift out control character (SO = hex 0E) value or a shift in control character (SI = hex 0F) value. Specification of such an invalid value results in the signaling of the *scalar value invalid* (hex 3203) exception.

Operands 1 and 2 must be specified as Data Pointers which define either a simple (single byte) character data field or one of the extended (double byte) character data fields.

Support for usage of a Data Pointer defining an extended character scalar value is limited to this instruction. Usage of such a data pointer defined value on any other instruction is not supported and results in the signaling of the *scalar type invalid* (hex 3201) exception.

For more information on support for extended character data fields, refer to the Set Data Pointer Attributes, Materialize Pointer, and Create Cursor instructions.

Four data types are supported for data pointer definition of extended (double byte) character fields, OPEN, EITHER, ONLYNS and ONLYS. Except for ONLYNS, the double byte character data must be surrounded by a shift out control character (SO hex 0E) and a shift in control character (SI = hex 0F).

- The ONLYNS field only contains double byte data with no SO, SI delimiters surrounding it.
- The ONLYS field can only contain double byte character data within a SO..SI pair.
- The EITHER field can consist of double byte character or single byte character data but only one type at a time. If double byte character data is present it must be surrounded by an SO .. SI pair.
- The OPEN field can consist of a mixture of double byte character and single byte character data. If double byte character data is present it must be surrounded by an SO .. SI pair.

Specifying an extended character value which violates the above restrictions results in the signaling of the *invalid extended character data* (hex 0C12) exception.

The valid copy operations which can be specified on this instruction are the following:

		Op 1			
		Onlyns	Onlys	Open	Either
O	Onlyns	yes	yes	yes	yes
p	Onlys	yes	yes	yes	yes
	Open	no	no	yes	no
2	Either	no	no	yes	yes

Figure 2-5. Valid copy operations for CPYECLAP

Specifying a copy operation other than the valid operations defined above results in the signaling of the *invalid extended character operation* (hex 0C13) exception.

When the copy operation is for a source of type ONLYNS (no SO/SI delimiters) being copied to a receiver which is not ONLYNS, SO and SI delimiters are implicitly added around the source value as part of the copy operation.

When the source value is longer than can be contained in the receiver, truncation is necessary and the following truncation rules apply:

1. Truncation is on the right (like simple character copy operations).
2. When the string to be truncated is a single byte character string, or an extended character string when the receiver is ONLYNS, bytes beyond those that fit into the receiver are truncated with no further processing needed.
3. When the string to be truncated is an extended character string and the receiver is not ONLYNS, the bytes that fall at the end of the receiver are truncated as follows:
   a. When the last byte that would fit in the receiver is the first byte of an extended character, that byte is truncated and replaced with an SI character.
   b. When the last byte that would fit in the receiver is the second byte of an extended character, both bytes of that extended character are truncated and replaced with a SI character followed by a *single byte pad value*. This type of truncation can only occur when converting to an OPEN field.

When the source value is shorter than that which can be contained in the receiver, padding is necessary. One of three types of padding is performed:

1. Double byte (DB) - the source value is padded on the right with *double byte pad values* out to the length of the receiver.
2. Double byte concatenated with a SI value (DB||SI) - the source double byte value is padded on the right with *double byte pad values* out to the second to last byte of the receiver and an SI delimiter is placed in the last byte of the receiver.
3. Single byte (SB) - the source value is padded on the right with *single byte pad values* out to the length of the receiver.

The type of padding performed is determined by the type of operands involved in the operation:

1. If the receiver is ONLYNS, DB padding is performed.
2. If the receiver is ONLYS, DB||SI padding will be performed.
3. If the receiver is EITHER and the source contained a double byte value, DB||SI padding is performed.
4. If the receiver is EITHER and the source contained a single byte value, SB padding is performed.

5. If the receiver is OPEN, SB padding is performed.

The above padding rules cover all the operand combinations which are allowed on the instruction. A complete understanding of the operand combinations allowed (prior diagram), and the values which can be contained in the different operand types is necessary to appreciate that these rules do cover all the valid combinations.

Exceptions

Exception		Operands 1	2	3	Other
06	Addressing				
	01 Spacing addressing violation	X	X	X	
	02 Boundary alignment	X	X	X	
	03 Range	X	X	X	
	04 External data object not found	X	X		
	06 Optimized addressability invalid	X	X	X	
08	Argument/parameter				
	01 Parameter reference violation	X	X	X	
0C	Computation				
	12 Invalid extended character data		X		
	13 Invalid extended character operation				X
10	Damage encountered				
	04 System object damage state	X	X	X	X
	44 Partial system object damage	X	X	X	X
1C	Machine-dependent exception				
	03 Machine storage limit exceeded				X
20	Machine support				
	02 Machine check				X
	03 Function check				X
22	Object access				
	01 Object not found	X	X	X	
	02 Object destroyed	X	X	X	
	03 Object suspended	X	X	X	
	08 object compressed				X
24	Pointer specification				
	01 Pointer does not exist	X	X	X	
	02 Pointer type invalid	X	X	X	
2E	Resource control limit				
	01 user profile storage limit exceeded				X
32	Scalar specification				
	01 Scalar type invalid	X	X		
	01 Scalar value invalid			X	

Exception		Operands 1	2	3	Other
36	Space management				
	01 space extension/truncation				X

Copy Hex Digit Numeric to Numeric (CPYHEXNN)

Op Code (Hex)	Operand 1	Operand 2
1092	Receiver	Source

Operand 1: Numeric variable scalar or character variable scalar (fixed-length).

Operand 2: Numeric scalar or character scalar (fixed-length).

Description: The numeric hex digit value (rightmost 4 bits) of the leftmost byte referred to by the source operand is copied to the numeric hex digit value (rightmost 4 bits) of the leftmost byte referred to by the receiver operand. The operands can be either character strings or numeric. Any numeric operands are interpreted as logical character strings.

Exceptions

Exception		Operands 1	Operands 2	Other
06	Addressing			
	01 Spacing addressing violation	X	X	
	02 Boundary alignment	X	X	
	03 Range	X	X	
	06 Optimized addressability invalid	X	X	
08	Argument/parameter			
	01 Parameter reference violation	X	X	
10	Damage encountered			
	04 System object damage state	X	X	X
	44 Partial system object damage	X	X	X
1C	Machine-dependent exception			
	03 Machine storage limit exceeded			X
20	Machine support			
	02 Machine check			X
	03 Function check			X
22	Object access			
	01 Object not found	X	X	
	02 Object destroyed	X	X	
	03 Object suspended	X	X	
	08 object compressed			X
24	Pointer specification			
	01 Pointer does not exist	X	X	
	02 Pointer type invalid	X	X	
2E	Resource control limit			
	01 user profile storage limit exceeded			X
36	Space management			

Exception	Operands 1	2	Other
01 space extension/truncation			X

Copy Hex Digit Numeric to Zone (CPYHEXNZ)

Op Code (Hex)	Operand 1	Operand 2
1096	Receiver	Source

Operand 1: Numeric variable scalar or character variable scalar (fixed-length).

Operand 2: Numeric scalar or character scalar (fixed-length).

Description: The numeric hex digit value (rightmost 4 bits) of the leftmost byte referred to by the source operand is copied to the numeric hex digit value (rightmost 4 bits) of the leftmost byte referred to by the receiver operand. The operands can be either character strings or numeric. Any numeric operands are interpreted as logical character strings.

Exceptions

Exception		Operands 1	Operands 2	Other
06	Addressing			
	01 Spacing addressing violation	X	X	
	02 Boundary alignment	X	X	
	03 Range	X	X	
	06 Optimized addressability invalid	X	X	
08	Argument/parameter			
	01 Parameter reference violation	X	X	
10	Damage encountered			
	04 System object damage state	X	X	X
	44 Partial system object damage	X	X	X
1C	Machine-dependent exception			
	03 Machine storage limit exceeded			X
20	Machine support			
	02 Machine check			X
	03 Function check			X
22	Object access			
	01 Object not found	X	X	
	02 Object destroyed	X	X	
	03 Object suspended	X	X	
	08 object compressed			X
24	Pointer specification			
	01 Pointer does not exist	X	X	
	02 Pointer type invalid	X	X	
2E	Resource control limit			
	01 user profile storage limit exceeded			X
36	Space management			

Exception	Operands 1	Operands 2	Other
01 space extension/truncation			X

Copy Hex Digit Zone To Numeric (CPYHEXZN)

Op Code (Hex)	Operand 1	Operand 2
109A	Receiver	Source

Operand 1: Numeric variable scalar or character variable scalar (fixed-length).

Operand 2: Numeric scalar or character scalar (fixed-length).

Description: The zone hex digit value (leftmost 4 bits) of the leftmost byte referred to by the source operand is copied to the numeric hex digit value (rightmost 4 bits) of the leftmost byte referred to by the receiver operand.

The operands can be either character strings or numeric. Any numeric operands are interpreted as logical character strings.

Exceptions

Exception		Operands 1	Operands 2	Other
06	Addressing			
	01 Spacing addressing violation	X	X	
	02 Boundary alignment	X	X	
	03 Range	X	X	
	06 Optimized addressability invalid	X	X	
08	Argument/parameter			
	01 Parameter reference violation	X	X	
10	Damage encountered			
	04 System object damage state	X	X	X
	44 Partial system object damage	X	X	X
1C	Machine-dependent exception			
	03 Machine storage limit exceeded			X
20	Machine support			
	02 Machine check			X
	03 Function check			X
22	Object access			
	01 Object not found	X	X	
	02 Object destroyed	X	X	
	03 Object suspended	X	X	
	08 object compressed			X
24	Pointer specification			
	01 Pointer does not exist	X	X	
	02 Pointer type invalid	X	X	
2E	Resource control limit			
	01 user profile storage limit exceeded			X

Exception		Operands 1	2	Other
36	Space management			
	01 space extension/truncation			X

Copy Hex Digit Zone To Zone (CPYHEXZZ)

Op Code (Hex)	Operand 1	Operand 2
109E	Receiver	Source

Operand 1: Numeric variable scalar or character variable scalar (fixed-length).

Operand 2: Numeric scalar or character scalar (fixed-length).

Description: The zone hex digit value (leftmost 4 bits) of the leftmost byte referred to by the source operand is copied to the zone hex digit value (leftmost 4 bits) of the leftmost byte referred to by the receiver operand.

The operands can be either character strings or numeric. Any numeric operands are interpreted as logical character strings.

Exceptions

Exception		Operands 1	Operands 2	Other
06	Addressing			
	01 Spacing addressing violation	X	X	
	02 Boundary alignment	X	X	
	03 Range	X	X	
	06 Optimized addressability invalid	X	X	
08	Argument/parameter			
	01 Parameter reference violation	X	X	
10	Damage encountered			
	04 System object damage state	X	X	X
	44 Partial system object damage	X	X	X
1C	Machine-dependent exception			
	03 Machine storage limit exceeded			X
20	Machine support			
	02 Machine check			X
	03 Function check			X
22	Object access			
	01 Object not found	X	X	
	02 Object destroyed	X	X	
	03 Object suspended	X	X	
	08 object compressed			X
24	Pointer specification			
	01 Pointer does not exist	X	X	
	02 Pointer type invalid	X	X	
2E	Resource control limit			
	01 user profile storage limit exceeded			X

Exception		Operands 1	2	Other
36	Space management			
	01 space extension/truncation			X

Copy Numeric Value (CPYNV)

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand [3-6]
CPYNV 1042		Receiver	Source	
CPYNVR 1242		Receiver	Source	
CPYNVB 1C42	Branch options	Receiver	Source	Branch targets
CPYNVBR 1E42	Branch options	Receiver	Source	Branch targets
CPYNVI 1842	Indicator options	Receiver	Source	Indicator targets
CPYNVIR 1A42	Indicator options	Receiver	Source	Indicator targets

Operand 1: Numeric variable scalar or data-pointer-defined numeric scalar.

Operand 2: Numeric scalar or data pointer-defined-numeric scalar.

Operand 3-6:

- *Branch Form*-Branch point, instruction pointer, relative instruction number, or absolute instruction number.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

ILE access

```
LBCPYNV (
  var receiver            : any numeric type;
  var receiver_attributes : aggregate;
  var source              : any numeric type;
  var source_attributes   : aggregate;
)
```

Description: The numeric value of the source operand is copied to the numeric receiver operand.

Both operands must be numeric. If necessary, the source operand is converted to the same type as the receiver operand before being copied to the receiver operand. The source value is adjusted to the length of the receiver operand, aligned at the assumed decimal point of the receiver operand, or both before being copied to it. If significant digits are truncated on the left end of the source value, a *size* (hex 0C0A) exception is signaled. When the receiver is binary, this *size* (hex 0C0A) exception may be suppressed by using the *suppress binary size exception program attribute* on the Create Program (CRTPG) instruction.

If a decimal to binary conversion causes a *size* (hex 0C0A) exception to be signaled or if the *size* (hex 0C0A) exception is suppressed, the binary value contains the correct truncated result only if the decimal value contains 15 or fewer significant nonfractional digits.

Conversions between floating-point integers and integer formats (binary or decimal with no fractional digits) is exact, except when an exception is signaled.

An *invalid floating-point conversion* (hex 0C0C) exception is signaled when an attempt is made to convert from floating-point to binary or decimal and the result would represent infinity or NaN, or nonzero digits would be truncated from the left end of the resultant value.

For the optional round form of the instruction, a floating-point receiver operand is invalid.

For a fixed-point operation, if significant digits are truncated from the left end of the source value, a *size* (hex 0C0A) exception is signaled. When the receiver is binary, this *size* (hex 0C0A) exception may be suppressed by using the *suppress binary size exception program attribute* on the Create Program (CRTPG) instruction.

For a floating-point receiver, if the exponent of the resultant value is too large or too small to be represented in the receiver field, the *floating-point overflow* (hex 0C06) and *floating-point underflow* (hex 0C07) exceptions are signaled, respectively.

Resultant Conditions

- Positive-The algebraic value of the numeric scalar receiver operand is positive.
- Negative-The algebraic value of the numeric scalar receiver operand is negative.
- Zero-The algebraic value of the numeric scalar receiver operand is zero.
- Unordered-The value assigned a floating-point receiver operand is NaN.

Exceptions

Exception		Operands 1	Operands 2	Other
06	Addressing			
	01 Spacing addressing violation	X	X	
	02 Boundary alignment	X	X	
	03 Range	X	X	
	04 External data object not found	X	X	
	06 Optimized addressability invalid	X	X	
08	Argument/parameter			
	01 Parameter reference violation	X	X	
0C	Computation			
	02 Decimal data		X	
	06 Floating-point overflow	X		
	07 Floating-point underlow	X		
	09 Floating-point invalid operand	X		X
	0A Size	X		
	0C Invalid floatin-point conversion	X		
	0A Floating-point inexact result	X		
10	Damage encountered			
	04 System object damage state	X	X	X
	44 Partial system object damage	X	X	X

Exception		Operands 1	2	Other
1C	Machine-dependent exception			
	03 Machine storage limit exceeded			X
20	Machine support			
	02 Machine check			X
	03 Function check			X
22	Object access			
	01 Object not found	X	X	
	02 Object destroyed	X	X	X
	03 Object suspended	X	X	X
	08 object compressed			X
24	Pointer specification			
	01 Pointer does not exist	X	X	X
	02 Pointer type invalid	X	X	X
2C	Program execution			
	04 Invalid branch target			X
2E	Resource control limit			
	01 user profile storage limit exceeded			X
32	Scalar specification			
	01 Scalar type invalid		X	X
36	Space management			
	01 space extension/truncation			X

Decompress Data (DCPDATA)

Op Code (Hex)	Operand 1
1051	Decompress Data template

Operand 1: Space pointer.

ILE access

```
DCPDATA (
     decompress_data_template : space pointer
)
```

Description: The instruction decompresses user data. Operand 1 identifies a template which identifies the data to be decompressed. The template also identifies the result space to receive the decompressed data.

The Decompress Data template must be aligned on a 16-byte boundary. The format is as follows:

- Reserved (binary 0) — Char(4)
- Result area length — Bin(4)
- Actual result length — Bin(4)*
- Reserved (binary 0) — Char(20)
- Source space pointer — Space pointer
- Result space pointer — Space pointer

Note: The input value associated with template entries annotated with an asterisk (*) are ignored by the instruction; these fields are updated by the instruction to return information about instruction execution.

The data at the location specified by the **source space pointer** is decompressed and stored at the location specified by the **result space pointer**. The **actual result length** is set to the number of bytes in the decompressed result. The Source data is not modified.

The **result area length** field value must be greater than zero. The length of the source data is not supplied in the template because this length is contained within the compressed data.

If the decompressed result data will not fit in the result area (as specified by the *result area length*), the decompression is stopped and only as many decompressed bytes as will fit in the result area are stored. The *actual result length* is always set to the full length of the result, which may be larger than the *result area length*.

The compressed data (previously compressed with CPRDATA) contains a **signature** which is checked by DCPDATA. The *signature* indicates which compression algorithm was used to compress the data. If the *signature* is invalid, an *invalid compressed data* (hex 0C14) exception is signaled. It is possible that the *signature* appears valid even though the compressed data has been corrupted. In almost all cases, the DCPDATA instruction will signal the *invalid compressed data* (hex 0C14) exception. Data corruption will not be detected only in the case when the decompression algorithm applied to the corrupted data produces the correct number of decompressed bytes.

It is not possible to corrupt the compressed data in such a way that the DCPDATA instruction would fail (that is, function check) or fail to terminate (that is, loop).

Authorization Required

- None

Lock Enforcement

- None

Exceptions

Exception		Operands 1	Other
06	Addressing		
	01 space addressing violation	X	X
	02 boundary alignment	X	
	03 range	X	
	06 optimized addressability invalid	X	
08	Argument/Parameter		
	01 Parameter reference violation	X	
0C	Computation		
	14 invalid compressed data		X
10	Damage encountered		
	44 partial system object damage	X	X
1C	Machine-dependent exception		
	03 machine storage limit exceeded		X
20	Machine support		
	02 machine check		X
	03 function check		X
22	Object access		
	02 object destroyed	X	X
	03 object suspended	X	X
	08 object compressed		X
24	Pointer specification		
	01 pointer does not exist	X	X
	02 pointer type invalid	X	X
	03 pointer addressing invalid object	X	X
2E	Resource control limit		
	01 user profile storage limit exceeded		X
36	Space management		
	01 space extension/truncation	X	X
38	Template specification		
	01 template value invalid	X	

Exception		Operands 1	Other
44	Domain		
	01 object domain error	X	

Divide (DIV)

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand 3	Operand [4-7]
DIV 104F		Quotient	Dividend	Divisor	
DIVR 124F		Quotient	Dividend	Divisor	
DIVI 184F	Indicator options	Quotient	Dividend	Divisor	Indicator targets
DIVIR 1A4F	Indicator options	Quotient	Dividend	Divisor	Indicator targets
DIVB 1C4F	Branch options	Quotient	Dividend	Divisor	Branch targets
DIVBR 1E4F	Branch options	Quotient	Dividend	Divisor	Branch targets

Operand 1: Numeric variable scalar.

Operand 2: Numeric scalar.

Operand 3: Numeric scalar.

Operand 4-7:

- *Branch Form*-Branch point, instruction pointer, relative instruction number, or absolute instruction number.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Short forms

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand [3-6]
DIVS 114F		Quotient/Dividend	Divisor	
DIVSR 134F		Quotient/Dividend	Divisor	
DIVIS 194F	Indicator options	Quotient/Dividend	Divisor	Indicator targets
DIVISR 1B4F	Indicator options	Quotient/Dividend	Divisor	Indicator targets
DIVBS 1D4F	Branch options	Quotient/Dividend	Divisor	Branch targets
DIVBSR 1F4F	Branch options	Quotient/Dividend	Divisor	Branch targets

Operand 1: Numeric variable scalar.

Operand 2: Numeric scalar.

Operand 3-6:

- *Branch Form*-Branch point, instruction pointer, relative instruction number, or absolute instruction number.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Description: The Quotient is the result of dividing the Dividend by the Divisor.

Operands can have floating-point, packed or zoned decimal, signed or unsigned binary type.

Source operands are the Dividend and Divisor. The receiver operand is the Quotient.

If operands are not of the same type, source operands are converted according to the following rules:

1. If any one of the operands has floating point type, source operands are converted to floating point type.
2. Otherwise, if any one of the operands has zoned or packed decimal type, source operands are converted to packed decimal.
3. Otherwise, the binary operands are converted to a like type. Note: unsigned binary(2) scalars are logically treated as signed binary(4) scalars.

Source operands are divided according to their type. Floating point operands are divided using floating point division. Packed decimal operands are divided using packed decimal division. Unsigned binary division is used with unsigned source operands. Signed binary operands are divided using two's complement binary division.

Better performance can be obtained if all operands have the same type. Signed and unsigned binary division execute faster than either packed decimal or floating point division.

Decimal operands used in floating-point operations cannot contain more than 15 total digit positions.

If the divisor has a numeric value of zero, a *zero divide* (hex 0C0B) or *floating-point zero divide* (hex 0C0E) exception is signaled respectively for fixed-point versus floating-point operations. If the dividend has a value of zero, the result of the division is a zero quotient value.

If the divisor has a numeric value of 0, a *zero divide* (hex 0C0B) exception is signaled. If the dividend has a value of 0, the result of the division is a zero value quotient.

For a decimal operation, the precision of the result of the divide operation is determined by the number of fractional digit positions specified for the quotient. In other words, the divide operation will be performed so as to calculate a resultant quotient of the same precision as that specified for the quotient operand. If necessary, internal alignment of the assumed decimal point for the dividend and divisor operands is performed to ensure the correct precision for the resultant quotient value. These internal alignments are not subject to detection of the decimal point alignment exception. An internal quotient value will be calculated for any combination of decimal attributes which may be specified for the instruction's operands. However, the assignment of the result to the quotient operand is subject to detection of the *size* (hex 0C0A) exception thereby limiting the assignment to, at most, the rightmost 31 digits of the calculated result.

Floating-point division uses exponent subtraction and significand division.

If the dividend operand is shorter than the divisor operand, it is logically adjusted to the length of the divisor operand.

For fixed-point computations and for the significand division of a floating-point computation, the division operation is performed according to the rules of algebra. Unsigned binary is treated as a positive number for the algebra.

For a floating-point computation, the operation is performed as if to infinite precision.

The result of the operation is copied into the quotient operand. If this operand is not the same type as that used in performing the operation, the resultant value is converted to its type. If necessary, the resultant value is adjusted to the length of the quotient operand, aligned at the assumed decimal point of the quotient operand, or both before being copied to it. If significant digits are truncated on the left end of the resultant value, a *size* (hex 0C0A) exception is signaled.

If a decimal to binary conversion causes a *size* (hex 0C0A) exception to be signaled, the binary value contains the correct truncated result only if the decimal value contains 15 or fewer significant nonfractional digits.

For the optional round form of the instruction, specification of a floating-point receiver operand is invalid.

For fixed-point operations in programs that request to be notified of *size* (hex 0C0A) exceptions, if nonzero digits are truncated from the left end of the resultant value, a *size* (hex 0C0A) exception is signaled.

For floating-point operations that involve a fixed-point receiver field, if nonzero digits would be truncated from the left end of the resultant value, an *invalid floating-point conversion* (hex 0C0C) exception is signaled.

For a floating-point quotient operand, if the exponent of the resultant value is either too large or too small to be represented in the quotient field, the *floating-point overflow* (hex 0C06) and *floating-point underflow* (hex 0C07) exceptions are signaled, respectively.

Resultant Conditions

- Positive-The algebraic value of the numeric scalar quotient is positive.
- Negative-The algebraic value of the numeric scalar quotient is negative.
- Zero-The algebraic value of the numeric scalar quotient is or zero.
- Unordered-The value assigned a floating-point quotient operand is NaN.

Exceptions

Exception		Operands 1	2	3	Other
06	Addressing				
	01 Spacing addressing violation	X	X	X	
	02 Boundary alignment	X	X	X	
	03 Range	X	X	X	
	06 Optimized addressability invalid	X	X	X	
08	Argument/parameter				
	01 Parameter reference violation	X	X	X	
0C	Computation				
	02 Decimal data		X	X	
	06 Floating-point overflow	X			

Exception		Operands 1	2	3	Other
	07 Floating-point underlow	X			
	09 Floating-point invalid operand		X	X	X
	0A Size	X			
	0B Zero divide		X		
	0C Invalid floatin-point conversion	X			
	0D Floating-point inexact result	X			
	0E Floating-point divide by zero			X	
10	Damage encountered				
	04 System object damage state	X	X	X	X
	44 Partial system object damage	X	X	X	X
1C	Machine-dependent exception				
	03 Machine storage limit exceeded				X
20	Machine support				
	02 Machine check				X
	03 Function check				X
22	Object access				
	01 Object not found	X	X	X	
	02 Object destroyed	X	X	X	
	03 Object suspended	X	X	X	
	08 object compressed				X
24	Pointer specification				
	01 Pointer does not exist	X	X	X	
	02 Pointer type invalid	X	X	X	
2C	Program execution				
	04 Invalid branch target				X
2E	Resource control limit				
	01 user profile storage limit exceeded				X
36	Space management				
	01 space extension/truncation				X

Divide with Remainder (DIVREM)

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand 3	Operand 4	Operand [5-7]
DIVREM 1074		Quotient	Dividend	Divisor	Remainder	
DIVREMR 1274		Quotient	Dividend	Divisor	Remainder	
DIVREMI 1874	Indicator options	Quotient	Dividend	Divisor	Remainder	Indicator targets
DIVREMIR 1A74	Indicator options	Quotient	Dividend	Divisor	Remainder	Indicator targets
DIVREMB 1C74	Branch options	Quotient	Dividend	Divisor	Remainder	Branch targets
DIVREMBR 1E74	Branch options	Quotient	Dividend	Divisor	Remainder	Branch targets

Operand 1: Numeric variable scalar.

Operand 2: Numeric scalar.

Operand 3: Numeric scalar.

Operand 4: Numeric variable scalar.

Operand 5-7:

- *Branch Form*-Branch point, instruction pointer, relative instruction number, or absolute instruction number.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand 3	Operand [4-6]
DIVREMS 1174		Quotient/Dividend	Divisor	Remainder	
DIVREMSR 1374		Quotient/Dividend	Divisor	Remainder	
DIVREMIS 1974	Indicator options	Quotient/Dividend	Divisor	Remainder	Indicator targets
DIVREMISR 1B74	Indicator options	Quotient/Dividend	Divisor	Remainder	Indicator targets
DIVREMBS 1D74	Branch options	Quotient/Dividend	Divisor	Remainder	Branch targets
DIVREMBSR 1F74	Branch options	Quotient/Dividend	Divisor	Remainder	Branch targets

Operand 1: Numeric variable scalar.

Operand 2: Numeric scalar.

Operand 3: Numeric variable scalar.

Operand 4-6:

- *Branch Form*-Branch point, instruction pointer, relative instruction number, or absolute instruction number.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Description: The Quotient is the result of dividing the Dividend by the Divisor. The Remainder is the Dividend minus the product of the Divisor and Quotient.

Operands can have packed or zoned decimal, signed or unsigned binary type.

Source operands are the Dividend and Divisor. The receiver operands are the Quotient and Remainder.

If operands are not of the same type, source operands are converted according to the following rules:

1. If any one of the operands has zoned or packed decimal type, source operands are converted to packed decimal.
2. Otherwise, the binary operands are converted to a like type. Note: unsigned binary(2) scalars are logically treated as signed binary(4) scalars.

Source operands are divided according to their type. Packed decimal operands are divided using packed decimal division. Unsigned binary division is used with unsigned source operands. Signed binary operands are divided using two's complement binary division.

Better performance can be obtained if all operands have the same type. Signed and unsigned binary division execute faster than packed decimal division.

Floating-point is not supported for this instruction.

If the divisor operand has a numeric value of 0, a *zero divide* (hex 0C0B) exception is signaled. If the dividend operand has a value of 0, the result of the division is a zero value quotient and remainder.

For a decimal operation, the precision of the result of the divide operation is determined by the number of fractional digit positions specified for the quotient. In other words, the divide operation will be performed so as to calculate a resultant quotient of the same precision as that specified for the quotient operand. If necessary, internal alignment of the assumed decimal point for the dividend and divisor operands is performed to ensure the correct precision for the resultant quotient value. These internal alignments are not subject to detection of the decimal point alignment exception. An internal quotient value will be calculated for any combination of decimal attributes which may be specified for the instruction's operands. However, the assignment of the result to the quotient operand is subject to detection of the size exception thereby limiting the assignment to, at most, the rightmost 31 digits of the calculated result.

If the dividend operand is shorter than the divisor operand, it is logically adjusted to the length of the divisor operand.

The division operation is performed according to the rules of algebra. Unsigned binary is treated as a positive number for the algebra. The quotient result of the operation is copied into the quotient operand. If this operand is not the same type as that used in performing the operation, the resultant value is converted to its type. If necessary, the resultant value is adjusted to the length of the quotient operand, aligned at the assumed decimal point of the quotient operand, or both before being copied to

it. If significant digits are truncated on the left end of the resultant value, a *size* (hex 0C0A) exception is signaled.

After the quotient numeric value has been determined, the numeric value of the remainder operand is calculated as follows:

```
Remainder = Dividend - (Quotient*Divisor)
```

If the optional round form of this instruction is being used, the rounding applies to the quotient but not the remainder. The quotient value used to calculate the remainder is the resultant value of the division. The resultant value of the calculation is copied into the remainder operand. The sign of the remainder is the same as that of the dividend operand unless the remainder has a value of 0, in which case its sign is positive. If the remainder operand is not the same type as that used in performing the operation, the resultant value is converted to its type. If necessary, the resultant value is adjusted to the length of the remainder operand, aligned at the assumed decimal point of the remainder operand, or both before being copied to it. If significant digits are truncated off the left end of the resultant value, a *size* (hex 0C0A) exception is signaled.

If a decimal to binary conversion causes a *size* (hex 0C0A) exception to be signaled (in programs that request size exceptions to be signaled), the binary value contains the correct truncated result only if the decimal value contains 15 or fewer significant nonfractional digits.

Resultant Conditions: The algebraic value of the numeric scalar quotient is

- positive
- negative
- zero

Exceptions

Exception		Operands 1	2	3	4	Other
06	Addressing					
	01 Spacing addressing violation	X	X	X	X	
	02 Boundary alignment	X	X	X	X	
	03 Range	X	X	X	X	
	06 Optimized addressability invalid	X	X	X	X	
08	Argument/parameter					
	01 Parameter reference violation	X	X	X	X	
0C	Computation					
	02 Decimal data		X	X		
	0A Size	X			X	
	0B Zero divide			X		
10	Damage encountered					
	04 System object damage state	X	X	X	X	X
	44 Partial system object damage	X	X	X	X	X
1C	Machine-dependent exception					
	03 Machine storage limit exceeded					X
20	Machine support					

Exception		Operands 1	2	3	4	Other
	02 Machine check					X
	03 Function check					X
22	Object access					
	01 Object not found	X	X	X	X	
	02 Object destroyed	X	X	X	X	
	03 Object suspended	X	X	X	X	
	08 object compressed					X
24	Pointer specification					
	01 Pointer does not exist	X	X	X	X	
	02 Pointer type invalid	X	X	X	X	
2C	Program execution					
	04 Invalid branch target					X
2E	Resource control limit					
	01 user profile storage limit exceeded					X
36	Space management					
	01 space extension/truncation					X

Edit (EDIT)

Op Code (Hex)	Operand 1	Operand 2	Operand 3
10E3	Receiver	Source	Edit Mask

Operand 1: Character variable scalar or data-pointer-defined character scalar.

Operand 2: Numeric scalar or data-pointer-defined numeric scalar.

Operand 3: Character variable scalar or data-pointer-defined character scalar.

ILE access

```
LBEDIT (
  var receiver          : aggregate;
  var receiver_length   : unsigned binary;
  var source            : signed binary; OR
                          unsigned binary; OR
                          packed decimal;
  var source_attributes : aggregate;
  var mask              : aggregate;
  var mask_length       : unsigned binary
)

OR

EDITPD (
  var receiver        : aggregate;
      receiver_length : unsigned binary;
  var source          : packed decimal;
      source_length   : unsigned binary;
  var mask            : aggregate;
      mask_length     : unsigned binary
)
```

Description: The value of a numeric scalar is transformed from its internal form to character form suitable for display at a source/sink device. The following general editing functions can be performed during transforming of the source operand to the receiver operand:

- Unconditional insertion of a source value digit with a zone as a function of the source value's algebraic sign
- Unconditional insertion of a mask operand character string
- Conditional insertion of one of two possible mask operand character strings as a function of the source value's algebraic sign
- Conditional insertion of a source value digit or a mask operand replacement character as a function of source value leading zero suppression
- Conditional insertion of either a mask operand character string or a series of replacement characters as a function of source value leading zero suppression
- Conditional floating insertion of one of two possible mask operand character strings as a function of both the algebraic sign of the source value and leading zero suppression

The operation is performed by transforming the source (operand 2) under control of the edit mask (operand 3) and placing the result in the receiver (operand 1).

The mask operand (operand 3) is limited to no more than 256 bytes.

Mask Syntax: The source field is converted to packed decimal format. The edit mask contains both control character and data character strings. Both the edit mask and the source fields are processed left to right, and the edited result is placed in the result field from left to right. If the number of digits in the source field is even, the four high-order bits of the source field are ignored and not checked for validity. All other source digits as well as the sign are checked for validity, and a *decimal data* (hex 0C02) exception is signaled when one is invalid. Overlapping of any of these fields gives unpredictable results.

Nine fixed value control characters can be in the edit mask, hex AA through hex AD and hex AF through hex B3. Four of these control characters specify strings of characters to be inserted into the result field under certain conditions; and the other five indicate that a digit from the source field should be checked and the appropriate action taken.

One variable value control character can be in the edit mask. This control character indicates the end of a string of characters. The value of the end-of-string character can vary with each execution of the instruction and is determined by the value of the first character in the edit mask. If the first character of the edit mask is a value less than hex 40, then that value is used as the end-of-string character. If the first character of the edit mask is a value equal to or greater than hex 40, then hex AE is used as the end-of-string character.

A significance indicator is set to the off state at the start of the execution of this instruction. It remains in this state until a nonzero source digit is encountered in the source field or until one of the four unconditional digits (hex AA through hex AD) or an unconditional string (hex B3) is encountered in the edit mask.

When significance is detected, the selected floating string is overlaid into the result field immediately before (to the left of) the first significant result character.

When the significance indicator is set to the on state, the first significant result character has been reached. The state of the significance indicator determines whether the fill character or a digit from the source field is to be inserted into the result field for conditional digits and characters in conditional strings specified in the edit mask field. The fill character is a hex 40 until it is replaced by the first character following the floating string specification control character (hex B1).

When the significance indicator is in the off state:

- A conditional digit control character in the edit mask causes the fill character to be moved to the result field.
- A character in a conditional string in the edit mask causes the fill character to be moved to the result field.

When the significance indicator is in the on state:

- A conditional digit control character in the edit mask causes a source digit to be moved to the result field.
- A character in a conditional string in the edit mask is moved to the result field.

The following control characters are found in the edit mask field.

End-of-String Character: One of these control characters (a value less than hex 40 or hex AE) indicates the end of a character string and must be present even if the string is null.

Static Field Character:

Hex AF This control character indicates the start of a static field. A static field is used to indicate that one of two mask character strings immediately following this character is to be inserted into the result field, depending upon the algebraic sign of the source field. If the sign is positive, the first string is to be inserted into the result field; if the sign is negative, the second string is to be inserted.

Static field format:

<Hex AF> <positive string>...<less than hex 40> <negative string>...<hex AE>

OR

<Hex AF> <positive string>...<hex AE> <negative string>...<hex AE>

Floating String Specification Field Character:

Hex B1 This control character indicates the start of a floating string specification field. The first character of the field is used as the fill character; following the fill character are two strings delimited by the end-of-string control character. If the algebraic sign of the source field is positive, the first string is to be overlaid into the result field; if the sign is negative, the second string is to be overlaid.

The string selected to be overlaid into the result field, called a floating string, appears immediately to the left of the first significant result character. If significance is never set, neither string is placed in the result field.

Conditional source digit positions (hex B2 control characters) must be provided in the edit mask immediately following the hex B1 field to accommodate the longer of the two floating strings; otherwise, a length conformance exception is signaled. For each of these B2 strings, the fill character is inserted into the result field, and source digits are not consumed. This ensures that the floating string never overlays bytes preceding the receiver operand.

Floating string specification field format:

<Hex B1> <fill character> <positive string>... <end-of-string character> <negative string>...<end-of-string character>

followed by

<Hex B2>...

Conditional String Character:

Hex B0 This control character indicates the start of a conditional string, which consists of any characters delimited by the end-of-string control character. Depending on the state of the significance indicator, this string or fill characters replacing it is inserted into the result field. If the significance indicator is off, a fill character for every character in the conditional string is placed in the result field. If the indicator is on, the characters in the conditional string are placed in the result field.

Conditional string format:

<Hex B0> <conditional string>...<end-of-string character>

Unconditional String Character:

Hex B3 This control character turns on the significance indicator and indicates the start of an unconditional string that consists of any characters delimited by the end-of-string control character. This string is unconditionally inserted into the result field regardless of the state of the significance indicator. If the indicator is off when a B3 control character is encountered, the appropriate floating string is overlaid into the result field before (to the left of) the B3 unconditional string (or to the left of where the unconditional string would have been if it were not null).

Unconditional string format:

<Hex B3> <unconditional string>. . .<end-of-string character>

Control Characters That Correspond to Digits in the Source Field:

Hex B2 This control character specifies that either the corresponding source field digit or the floating string (hex B1) fill character is inserted into the result field, depending on the state of the significance indicator. If the significance indicator is off, the fill character is placed in the result field; if the indicator is on, the source digit is placed. When a source digit is moved to the result field, the zone supplied is hex F. When significance (that is, a nonzero source digit) is detected, the floating string is overlaid to the left of the first significant character.

Control characters hex AA, hex AB, hex AC, and hex AD turn on the significance indicator. If the indicator is off when one of these control characters is encountered, the appropriate floating string is overlaid into the result field before (to the left of) the result digit.

Hex AA This control character specifies that the corresponding source field digit is unconditionally placed in the 4 low-order bits of the result field with the zone set to a hex F.

Hex AB This control character specifies that the corresponding source field digit is unconditionally placed in the result field. If the sign of the source field is positive, the zoned portion of the digit is set to hex F (the preferred positive sign); if the sign is negative, the zone portion is set to hex D (the preferred negative sign).

Hex AC This control character specifies that the corresponding source field digit is unconditionally placed in the result field. If the algebraic sign of the source field is positive, the zone portion of the result is set to hex F (the preferred positive sign); otherwise, the source sign field is moved to the result zone field.

Hex AD This control character specifies that the corresponding source field digit is unconditionally placed in the result field. If the algebraic sign of the source field is negative, the zone is set to hex D (the preferred negative sign); otherwise, the source field sign is moved to the zone position of the result byte.

The following table provides an overview of the results obtained with the valid edit conditions and sequences.

Table 2-1 (Page 1 of 3). Valid Edit Conditions and Results

Mask Character	Previous Significance Indicator	Source Digit	Source Sign	Result Character(s)	Resulting Significance Indicator
AF	Off/On	Any	Positive	Positive string inserted	No Change
	Off/On	Any	Negative	Negative string inserted	No Change
AA	Off	0-9	Positive	Positive floating string overlaid; hex F, source digit	On
	Off	0-9	Negative	Negative floating string overlaid; hex F, source digit	On
	On	0-9	Any	Hex F, source digit	On
AB	Off	0-9	Positive	Positive floating string overlaid; hex F, source digit	On
	Off	0-9	Negative	Negative floating string overlaid; hex D, source digit	On

Table 2-1 (Page 2 of 3). Valid Edit Conditions and Results

Mask Character	Previous Significance Indicator	Source Digit	Source Sign	Result Character(s)	Resulting Significance Indicator
	On	0-9	Positive	Hex F, source digit	On
	On	0-9	Negative	Hex D, source digit	On
AC	Off	0-9	Positive	Positive floating string overlaid; hex F, source digit	
	Off	0-9	Negative	Negative floating string overlaid; source sign and digit	On
	On	0-9	Positive	Hex F, source digit	On
	On	0-9	Negative	Source sign and digit	On
AD	Off	0-9	Positive	Positive floating string overlaid; source sign and digit	On
	Off	0-9	Negative	Negative floating string overlaid; hex D, source digit	On
	On	0-9	Positive	Source sign and digit	On
	On	0-9	Negative	Hex D, source digit	On
B0	Off	Any	Any	Insert fill character for each B0 string character	Off
	On	Any	Any	Insert B0 character string	On
B1 (including necessary B2s)	Off	Any	Any	Insert the fill character for each B2 character that corresponds to a character in the longer of the two floating strings	No Change
B2 (not for a B1 field)	Off	0	Any	Insert fill character	Off
	Off	1-9	Positive	Overlay positive floating string and insert hex F, source digit	On
	Off	1-9	Negative	Overlay negative floating string and insert hex F, source digit	On
	On	0-9	Any	Hex F, source digit	
B3	Off	Any	Positive	Overlay positive floating string and insert B3 character string	On

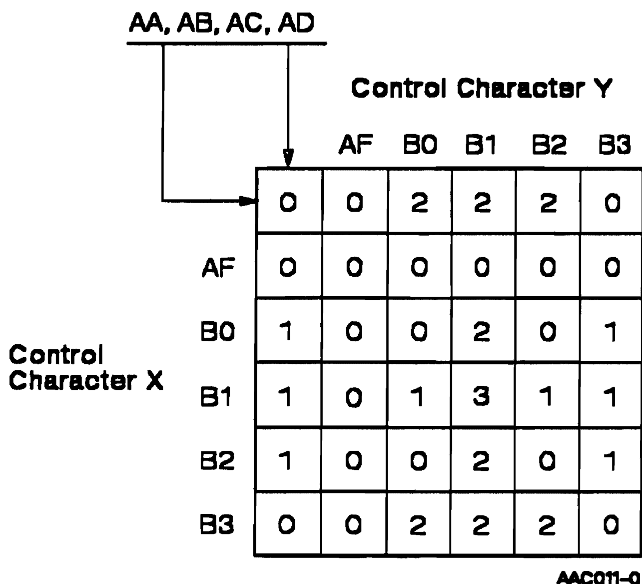
Table 2-1 (Page 3 of 3). Valid Edit Conditions and Results

Mask Character	Previous Significance Indicator	Source Digit	Source Sign	Result Character(s)	Resulting Significance Indicator
	Off	Any	Negative	Overlay negative floating string and insert B3 character string	On
	On	Any	Any	Insert B3 character string	On

Note:

1. Any character is a valid fill character, including the end-of-string character.
2. Hex AF, hex B1, hex B0, and hex B3 strings must be terminated by the end-of-string character even if they are null strings.
3. If a hex B1 field has not been encountered (specified) when the significance indicator is turned on, the floating string is considered to be a null string and is therefore not used to overlay into the result field.
4. If the positive and negative strings of a static field are of unequal length, additional static fields are necessary to ensure that the sum of the lengths of the positive strings equal the sum of the lengths of the negative strings; otherwise, a *length conformance* (0C08) exception is signaled because the receiver length does not correspond to the length implied by the edit mask and source field sign.

The following figure indicates the valid ordering of control characters in an edit mask field.



Explanation:

Condition Definition

0	Both X and Y can appear in the edit mask field in either order.
1	Y cannot precede X.
2	X cannot precede Y.
3	Both control characters (two B1's) cannot appear in an edit mask field.

Violation of any of the above rules will result in an *edit mask syntax* (hex 0C05) exception.

Figure 2-6. Edit Mask Field Control Characters

The following steps are performed when the editing is done:

- Convert Source Value to Packed Decimal
  - The numeric value in the source operand is converted to a packed decimal intermediate value before the editing is done. If the source operand is binary, then the attributes of the intermediate packed field before the edit are calculated as follows:

    Binary(2) = packed (5,0) or
    Binary(4) = packed (10,0)

- Edit
  - The editing of the source digits and mask insertion characters into the receiver operand is done from left to right.
- Insert Floating String into Receiver Field
  - If a floating string is to be inserted into the receiver field, this is done after the other editing.

Edit Digit Count Exception: An *edit digit count* (hex 0C04) exception is signaled when:

- The end of the source field is reached and there are more control characters that correspond to digits in the edit mask field.
- The end of the edit mask field is reached and there are more digit positions in the source field.

Edit Mask Syntax Exception: An *edit mask syntax* (hex 0C05) exception is signaled when an invalid edit mask control character is encountered or when a sequence rule is violated.

Length Conformance Exception: A *length conformance* (hex 0C08) exception is signaled when:

- The end of the edit mask field is reached and there are more character positions in the result field.
- The end of the result field is reached and more positions remain in the edit mask field.
- The number of B2s following a B1 field cannot accommodate the longer of the two floating strings.

Limitations: The following are limits that apply to the functions performed by this instruction.

Exceptions

Exception		Operands 1	2	3	Other
06	Addressing				
	01 Spacing addressing violation	X	X	X	
	02 Boundary alignment	X	X	X	
	03 Range	X	X	X	
	04 External data object not found	X	X	X	
	06 Optimized addressability invalid	X	X	X	
08	Argument/parameter				
	01 Parameter reference violation	X	X	X	
0C	Computation				
	02 Decimal data		X		
	04 Edit digit count		X		
	05 Edit mask syntax			X	
	08 Length conformance	X			

Exception		Operands 1	2	3	Other
10	Damage encountered				
	04 System object damage state	X	X	X	X
	44 Partial system object damage	X	X	X	X
1C	Machine-dependent exception				
	03 Machine storage limit exceeded				X
20	Machine support				
	02 Machine check				X
	03 Function check				X
22	Object access				
	01 Object not found	X	X	X	
	02 Object destroyed	X	X	X	
	03 Object suspended	X	X	X	
	08 object compressed				X
24	Pointer specification				
	01 Pointer does not exist	X	X	X	
	02 Pointer type invalid	X	X	X	
2E	Resource control limit				
	01 user profile storage limit exceeded				X
32	Scalar specification				
	01 Scalar type invalid	X	X	X	
	02 Scalar attributes invalid			X	
36	Space management				
	01 space extension/truncation				X

Exchange Bytes (EXCHBY)

Op Code (Hex)	Operand 1	Operand 2
10CE	Source 1	Source 2

Operand 1: Character variable scalar (fixed-length) or numeric variable scalar.

Operand 2: Character variable scalar (fixed-length) or numeric variable scalar.

Description: The logical character string values of the two source operands are exchanged. The value of the second source operand is placed in the first source operand and the value of the first source operand is placed in the second operand.

The operands can be either character or numeric. Any numeric operands are interpreted as logical character strings. Both operands must have the same length.

Exceptions

Exception		Operands 1	Operands 2	Other
06	Addressing			
	01 Spacing addressing violation	X	X	
	02 Boundary alignment	X	X	
	03 Range	X	X	
	06 Optimized addressability invalid	X	X	
08	Argument/parameter			
	01 Parameter reference violation	X	X	
10	Damage encountered			
	04 System object damage state	X	X	X
	44 Partial system object damage	X	X	X
1C	Machine-dependent exception			
	03 Machine storage limit exceeded			X
20	Machine support			
	02 Machine check			X
	03 Function check			X
22	Object access			
	01 Object not found	X	X	
	02 Object destroyed	X	X	
	03 Object suspended	X	X	
	08 object compressed			X
24	Pointer specification			
	01 Pointer does not exist	X	X	
	02 Pointer type invalid	X	X	
2E	Resource control limit			
	01 user profile storage limit exceeded			X

Exception		Operands 1	2	Other
36	Space management			
	01 space extension/truncation			X

Exclusive Or (XOR)

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand 3	Operand [4-5]
XOR 109B		Receiver	Source 1	Source 2	
XORI 189B	Indicator options	Receiver	Source 1	Source 2	Indicator targets
XORB 1C9B	Branch options	Receiver	Source 1	Source 2	Branch targets

Operand 1: Character variable scalar or numeric variable scalar.

Operand 2: Character scalar or numeric scalar.

Operand 3: Character scalar or numeric scalar.

Operand 4-5:

- *Branch Form*-Branch point, instruction pointer, relative instruction number, or absolute instruction number.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Short forms

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand [3-4]
XORS 119B		Receiver/Source 1	Source 2	
XORIS 199B	Indicator options	Receiver/Source 1	Source 2	Indicator targets
XORBS 1D9B	Branch options	Receiver/Source 1	Source 2	Branch targets

Operand 1: Character variable scalar or numeric variable scalar.

Operand 2: Character scalar or numeric scalar.

Operand 3-4:

- *Branch Form*-Branch point, instruction pointer, relative instruction number, or absolute instruction number.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Description: The Boolean EXCLUSIVE OR operation is performed on the string values in the source operands. The resulting string is placed in the receiver operand.

The operands may be character or numeric scalars. They are both interpreted as bit strings. Substringing is supported for both character and numeric operands.

The length of the operation is equal to the length of the longer of the two source operands. The shorter of the two operands is padded on the right. The operation begins with the two source operands left-adjusted and continues bit by bit until they are completed.

The bit values of the result are determined as follows:

Source 1 Bit	Source 2 Bit	Result Bit
0	0	0
0	1	1
1	0	1
1	1	0

The result value is then placed (left-adjusted) in the receiver operand with truncating or padding taking place on the right.

The pad value used in this instruction is a hex 00.

Substring operand references that allow for a null substring reference (a length value of zero) may be specified for operands 1, 2, and 3. The effect of specifying a null substring reference for one source operand is that the other source operand is EXCLUSIVE ORed with an equal length string of all hex 00s. When a null substring reference is specified for both source operands, the result is all zero and the instruction's resultant condition is zero. When a null substring reference is specified for the receiver, a result is not set and the instruction's resultant condition is zero regardless of the values of the source operands.

If operands overlap but do not share all of the same bytes, results of operations performed on these operands are not predictable. If overlapped operands share all of the same bytes, the results are predictable when direct addressing is used. If indirect addressing is used (that is, based operands, parameters, strings with variable lengths, and arrays with variable subscripts), the results are not always predictable.

When the receiver operand is a numeric variable scalar, it is possible that the result produced will not be a valid value for the numeric type. This can occur due to padding with hex 00, due to truncation, or due to the resultant bit string produced by the instruction. The instruction completes normally and signals no exceptions for these conditions.

Resultant Conditions

- Zero-The bit value for the bits of the scalar receiver operand is either all zero or a null substring reference is specified for the receiver.
- Not zero-The bit value for the bits of the scalar receiver operand is not all zero.

Exceptions

Exception		Operands 1	2	3	Other
06	Addressing				
	01 space addressing violation	X	X	X	
	02 boundary alignment	X	X	X	
	03 range	X	X	X	
	06 optimized addressability invalid	X	X	X	
08	Argument/parameter				
	01 parameter reference violation	X	X	X	
10	Damage encountered				
	04 system object damage state	X	X	X	X

Exception		Operands 1	2	3	Other
	44 partial system object damage	X	X	X	X
1C	Machine-dependent exception				
	03 machine storage limit exceeded				X
20	Machine support				
	02 machine check				X
	03 function check				X
22	Object access				
	01 object not found	X	X	X	
	02 object destroyed	X	X	X	
	03 object suspended	X	X	X	
	08 object compressed				X
24	Pointer specification				
	01 pointer does not exist	X	X	X	
	02 pointer type invalid	X	X	X	
2C	Program execution				
	04 invalid branch target				X
2E	Resource control limit				
	01 user profile storage limit exceeded				X
36	Space management				
	01 space extension/truncation				X

Extended Character Scan (ECSCAN)

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand 3	Operand 4	Operand [5-7]
ECSCAN 10D4		Receiver	Base	Compare operand	Mode operand	
ECSCANB 1CD4	Branch options	Receiver	Base	Compare operand	Mode operand	Branch targets
ECSCANI 18D4	Indicator options	Receiver	Base	Compare operand	Mode operand	Indicator targets

Operand 1: Binary variable scalar or binary array.

Operand 2: Character variable scalar.

Operand 3: Character scalar.

Operand 4: Character(1) scalar.

Operand 5-7:

- *Branch Form*-Branch point, instruction pointer, relative instruction number, or absolute instruction number.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Description: This instruction scans the string value of the base operand for occurrences of the string value of the compare operand and indicates the relative locations of these occurrences in the receiver operand. The character string value of the base operand is scanned for occurrences of the character string value of the compare operand under control of the mode operand and mode control characters embedded in the base string.

The base and compare operands must both be character strings. The length of the compare operand must not be greater than that of the base string. The base and compare operand are interpreted as containing a mixture of 1-byte (simple) and 2-byte (extended) character codes. The mode, simple or extended, with which the string is to be interpreted, is controlled initially by the mode operand and thereafter by mode control characters embedded in the strings. The mode control characters are as follows:

- Hex 0E = Shift out of simple character mode to extended mode.
- Hex 0F = Shift into simple character mode from extended mode. This is recognized only if it occurs in the first byte position of an extended character code.

The format of the mode operand is as follows:

• Mode operand	Char(1)
– Operand 2 initial mode indicator	Bit 0
0 = Operand starts in simple character mode. 1 = Operand starts in extended character mode.	
– Operand 3 initial mode indicator	Bit 1
0 = Operand starts in simple character mode. 1 = Operand starts in extended character mode.	
– Reserved (binary 0)	Bits 2-7

The operation begins at the left end of the base string and continues character by character, left to right. When the base string is interpreted in *simple character mode*, the operation moves through the base string 1 byte at a time. When the base string is interpreted in *extended character mode*, the operation moves through the base string 2 bytes at a time.

The compare operand value is the entire byte string specified for the compare operand. The **mode operand** determines the initial mode of the compare operand. The first character of the compare operand value is assumed to be a valid character for the initial mode of the compare operand and not a mode control character. Mode control characters in the compare operand value participate in comparisons performed during the scan function except that a mode control character as the first character of the compare operand causes unpredictable results.

The base string is scanned until the mode of the characters being processed is the same as the initial mode of the compare operand value. The operation continues comparing the characters of the base string with those of the compare operand value. The starting character of the characters being compared in the base string is always a valid character for the initial mode of the compare operand value. A mode control character encountered in the base string that changed the base string mode to match the initial mode of the compare operand value does not participate in the comparison. The length of the comparison is equal to the length of the compare operand value and the comparison is performed the same as performed by the Compare Bytes Left Adjusted instruction.

If a set of bytes that matches the compare operand value is found, the binary value for the relative location of the leftmost base string character of the set of bytes is placed in the receiver operand.

If the receiver operand is a scalar, only the first occurrence of the compare operand is noted. If the receiver operand is an array, as many occurrences as there are elements in the array are noted.

If a mode change is encountered in the base string, the base string is again scanned until the mode of the characters being processed is the same as the initial mode of the compare operand value, and then the comparisons are resumed.

The operation continues until no more occurrences of the compare operand value can be noted in the receiver operand or until the number of bytes remaining to be scanned in the base string is less than the length of the compare operand value. When the second condition occurs, the receiver value is set to zero. If the receiver operand is an array, all its remaining elements are also set to zero.

If the *escape code encountered* result condition is specified (through a branch or indicator option), verifications are performed on the base string as it is scanned. Each byte of the base string is checked for a value less than hex 40. When a value less than hex 40 is encountered, it is then determined if it is a valid mode control character.

If a byte value of less than hex 40 is not a valid mode control character, it is considered to be an escape code. The binary value for the relative location of the character (simple or extended) being interrogated is placed in the receiver operand, and the appropriate action (indicator or branch) is performed according to the specification of the escape code encountered result condition. If the receiver operand is an array, the next array element after any elements set with locations or prior occurrences of the compare operand, is set with the location of the character containing the escape code and all the remaining array elements are set to zero.

If the *escape encountered* result condition is not specified, verifications of the character codes in the base string are not performed.

Resultant Conditions

- Positive-The numeric value(s) of the receiver operand is positive.
- Zero-The numeric value(s) of the receiver operand is zero. In the case where the receiver operand is an array, the resultant condition is zero if all elements are zero.
- Escape code encountered-An escape character code value was encountered during the scanning of the base string.

Exceptions

Exception		Operands 1	2	3	4	Other
06	Addressing					
	01 space addressing violation	X	X	X	X	
	02 boundary alignment violation	X	X	X	X	
	03 range	X	X	X	X	
	06 optimized addressability invalid	X	X	X	X	
08	Argument/parameter					
	01 parameter reference violation	X	X	X	X	
0C	Computation					
	08 length conformance		X	X	X	
10	Damage encountered					
	04 System object damage state					X
	44 partial system object damage					X
1C	Machine-dependent exception					
	03 machine storage limit exceeded					X
20	Machine support					
	02 machine check					X
	03 function check					X
22	Object access					
	01 object not found	X	X	X	X	
	02 object destroyed	X	X	X	X	
	03 object suspended	X	X	X	X	
	08 object compressed					X
24	Pointer specification					
	01 pointer does not exist	X	X	X	X	
	02 pointer type invalid	X	X	X	X	
2C	Program execution					
	04 invalid branch target					X
2E	Resource control limit					
	01 user profile storage limit exceeded					X
32	Scalar specification					

Exception		Operands 1	2	3	4	Other
	01 scalar type invalid	X	X	X	X	
	03 scalar value invalid				X	
36	Space management					
	01 space extension/truncation					X

Extract Exponent (EXTREXP)

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand [3-6]
EXTREXP 1072		Receiver	Source	
EXTREXPB 1C72	Branch options	Receiver	Source	Branch targets
EXTREXPI 1872	Indicator options	Receiver	Source	Indicator targets

Operand 1: Binary variable scalar.

Operand 2: Floating-point scalar.

Operand 3-6:

- *Branch Form*-Branch point, instruction pointer, relative instruction number, or absolute instruction number.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Description: This instruction extracts the exponent portion of a floating-point scalar source operand and places it into the receiver operand as a binary variable scalar.

The operands must be the numeric types indicated because no conversions are performed.

The source floating-point field is interrogated to determine the binary floating-point value represented and either a signed exponent, for number values, or a special identifier, for infinity and NaN values, is placed in the binary variable scalar receiver operand.

The value to be assigned to the receiver, is dependent upon the floating-point value represented in the source operand as described below. It is a signed binary integer value and a numeric assignment of the value is made to the receiver.

When the source represents a normalized number, the biased exponent contained in the exponent field of the source is converted to the corresponding signed exponent by subtracting the bias of 127 for short or 1023 for long to determine the value to be returned. The resulting value ranges from -126 to +127 for short format, -1022 to +1023 for long format. When the receiver is unsigned binary, a negative exponent will result in a *size* (hex 0C0A) exception unless size exceptions are suppressed by using the *suppress binary size exception* program attribute on the Create Program (CRTPG) instruction.

When the source represents a denormalized number, the value to be returned is determined by adjusting the signed exponent of the denormalized number. The signed exponent of a denormalized number is a fixed value of -126 for the short format and -1022 for the long format. It is adjusted to the value the signed exponent would be if the source value was adjusted to a normalized number. The resulting value ranges from -127 to -149 for short format, -1023 to -1074 for long format.

When the source represents a value of zero, the value returned is zero.

When the source represents infinity, the value returned is +32767.

When the source represents a not-a-number, the value returned is -32768 for a signed binary receiver. For an unsigned binary(2) a value of 32768 is returned, and for a unsigned binary(4) a value of 4294934528 is returned.

Resultant Conditions

- Normalized-The source operand value represents a normalized binary floating-point number. The signed exponent is stored in the receiver.
- Denormalized-The source operand value represents a denormalized binary floating-point number. An adjusted signed exponent is stored in the receiver.
- Infinity-The source operand value represents infinity. The receiver is set with a value of +32767.
- NaN-The source operand value represents a not-a-number. The receiver is set with a value of -32768 when signed binary, with a value of 32768 when unsigned binary(2), and with a value of 4294934528 when unsigned binary(4).

Exceptions

Exception		Operands 1	Operands 2	Other
06	Addressing			
	01 space addressing violation	X	X	
	02 boundary alignment violation	X	X	
	03 range	X	X	
	06 optimized addressability invalid	X	X	
08	Argument/parameter			
	01 parameter reference violation	X	X	
10	Damage encountered			
	04 System object damage state			X
	44 partial system object damage			X
1C	Machine-dependent exception			
	03 machine storage limit exceeded			X
20	Machine support			
	02 machine check			X
	03 function check			X
22	Object access			
	01 object not found	X	X	
	02 object destroyed	X	X	
	03 object suspended	X	X	
	08 object compressed			X
24	Pointer specification			
	01 pointer does not exist	X	X	
	02 pointer type invalid	X	X	
2C	Program execution			
	04 invalid branch target			X
2E	Resource control limit			
	01 user profile storage limit exceeded			X
32	Scalar specification			

Exception		Operands 1	Operands 2	Other
	01 scalar type invalid	X	X	
36	Space management			
	01 space extension/truncation			X

Extract Magnitude (EXTRMAG)

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand [3-6]
EXTRMAG 1052		Receiver	Source	
EXTRMAGI 1852	Indicator options	Receiver	Source	Indicator targets
EXTRMAGB 1C52	Branch options	Receiver	Source	Branch targets

Operand 1: Numeric variable scalar.

Operand 2: Numeric scalar.

Operand 3-6:

- *Branch Form*-Branch point, instruction pointer, relative instruction number, or absolute instruction number.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Short forms

Op Code (Hex)	Extender	Operand 1	Operand [2-5]
EXTRMAGS 1152		Receiver/Source	
EXTRMAGIS 1952	Indicator options	Receiver/Source	Indicator targets
EXTRMAGBS 1D52	Branch options	Receiver/Source	Branch targets

Operand 1: Numeric variable scalar.

Operand 2-5:

- *Branch Form*-Branch point, instruction pointer, relative instruction number, or absolute instruction number.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Description: The numeric value of the source operand is converted to its absolute value and placed in the numeric variable scalar receiver operand.

The absolute value is formed from the source operand as follows:

- Signed binary
  - Extract the numeric value and form twos complement if the source operand is negative.
- Unsigned signed binary
  - Extract the numeric value.
- Packed/Zoned
  - Extract the numeric value and force the source operand's sign to positive.

- Floating-point
  - Extract the numeric value and force the significand sign to positive.

The result of the operation is copied into the receiver operand according to the rules of the Copy Numeric Value instruction. If this operand is not the same type as that used in performing the operation, the resultant value is converted to its type. If necessary, the resultant value is adjusted to the length of the receiver operand, or aligned at the assumed decimal point of the receiver operand, or both before being copied to it. If significant digits are truncated on the left end of the resultant value, a *size* (hex 0C0A) exception is signaled. An attempt to extract the magnitude of a maximum negative binary value to a binary scalar of the same size also results in a *size* (hex 0C0A) exception.

When the source floating-point operand represents not-a-number, the sign field of the source is not forced to positive and this value is not altered in the receiver.

If a decimal to binary conversion causes a *size* (hex 0C0A) exception to be signaled, the binary value contains the correct truncated result only if the decimal value contains 15 or fewer significant nonfractional digits.

For a fixed-point operation, if significant digits are truncated from the left end of the resultant value, a *size* (hex 0C0A) exception is signaled. An attempt to extract the absolute value of a maximum negative binary value into a binary scalar of the same size also results in a *size* (hex 0C0A) exception.

For floating-point operations that involve a fixed-point receiver field, if nonzero digits would be truncated from the left end of the resultant value, an *invalid floating-point conversion* (0C0C) exception is signaled.

For a floating-point receiver operand, if the exponent of the resultant value is either too large or too small to be represented in the receiver field, the *floating-point overflow* (hex 0C06) or the *floating-point underflow* (hex 0C07) exception is signaled.

Resultant Conditions

- Positive-The algebraic value of the receiver operand is positive.
- Zero-The algebraic value of the receiver operand is zero.
- Unordered-The value assigned a floating-point receiver operand is NaN.

Exceptions

Exception		Operands 1	Operands 2	Other
06	Addressing			
	01 space addressing violation	X	X	
	02 boundary alignment	X	X	
	03 range	X	X	
	06 optimized addressability invalid	X	X	
08	Argument/parameter			
	01 parameter reference violation	X	X	
0C	Computation			
	02 decimal data		X	
	06 floating-point overflow	X		
	07 floating-point underflow	X		

Exception		Operands 1	Operands 2	Other
	09 floating-point invalid operand		X	
	0A size	X		
	0C invalid floating point conversion	X		
	0D floating-point inexact result	X		
10	Damage encountered			
	04 system object damage	X	X	X
	44 partial system object damage	X	X	X
1C	Machine-dependent exception			
	03 machine storage limit exceeded			X
20	Machine support			
	02 machine check			X
	03 function check			X
22	Object access			
	01 object not found	X	X	
	02 object destroyed	X	X	
	03 object suspended	X	X	
	08 object compressed			X
24	Pointer specification			
	01 pointer does not exist	X	X	
	02 pointer type invalid	X	X	
2C	Program execution			
	04 invalid branch target			X
2E	Resource control limit			
	01 user profile storage limit exceeded			X
36	Space management			
	01 space extension/truncation			X

Multiply (MULT)

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand 3	Operand [4-7]
MULT 104B		Product	Multiplicand	Multiplier	
MULTR 124B		Product	Multiplicand	Multiplier	
MULTI 184B	Indicator options	Product	Multiplicand	Multiplier	Indicator targets
MULTIR 1A4B	Indicator options	Product	Multiplicand	Multiplier	Indicator targets
MULTB 1C4B	Branch options	Product	Multiplicand	Multiplier	Branch targets
MULTBR 1E4B	Branch options	Product	Multiplicand	Multiplier	Branch targets

Operand 1: Numeric variable scalar.

Operand 2: Numeric scalar.

Operand 3: Numeric scalar.

Operand 4-7:

- *Branch Form*-Branch point, instruction pointer, relative instruction number, or absolute instruction number.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Short forms

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand [3-6]
MULTS 104B		Product/Multiplicand	Multiplier	
MULTSR 134B		Product/Multiplicand	Multiplier	
MULTIS 194B	Indicator options	Product/Multiplicand	Multiplier	Indicator targets
MULTISR 1B4B	Indicator options	Product/Multiplicand	Multiplier	Indicator targets
MULTBS 1D4B	Branch options	Product/Multiplicand	Multiplier	Branch targets
MULTBSR 1F4B	Branch options	Product/Multiplicand	Multiplier	Branch targets

Operand 1: Numeric variable scalar.

Operand 2: Numeric scalar.

Operand 3-6:

- *Branch Form*-Branch point, instruction pointer, relative instruction number, or absolute instruction number.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Description: The Product is the result of multiplying the Multiplicand and the Multiplier.

Operands can have floating-point, packed or zoned decimal, signed or unsigned binary type.

Source operands are the Multiplicand and Multiplier. The receiver operand is the Product.

If operands are not of the same type, source operands are converted according to the following rules:

1. If any one of the operands has floating point type, source operands are converted to floating point type.
2. Otherwise, if any one of the operands has zoned or packed decimal type, source operands are converted to packed decimal.
3. Otherwise, the binary operands are converted to a like type. Note: unsigned binary(2) scalars are logically treated as signed binary(4) scalars.

Source operands are multiplied according to their type. Floating point operands are multiplied using floating point multiplication. Packed decimal operands are multiplied using packed decimal multiplication. Unsigned binary multiplication is used with unsigned source operands. Signed binary operands are multiplied using two's complement binary multiplication.

Better performance can be obtained if all operands have the same type. Signed and unsigned binary multiplication execute faster than either packed decimal or floating point multiplication.

Decimal operands used in floating-point operations cannot contain more than 15 total digit positions.

If the multiplicand operand or the multiplier operand has a value of 0, the result of the multiplication is a zero product.

For a decimal operation, no alignment of the assumed decimal point is performed for the multiplier and multiplicand operands.

The operation occurs using the specified lengths of the multiplicand and multiplier operands with no logical zero padding on the left necessary.

Floating-point multiplication uses exponent addition and significand multiplication.

For nonfloating-point computations and for significand multiplication for floating-point operations, the multiplication operation is performed according to the rules of algebra. Unsigned binary operands are treated as positive numbers for the algebra.

The result of the operation is copied into the product operand. If this operand is not the same type as that used in performing the operation, the resultant value is converted to its type. If necessary, the resultant value is adjusted to the length of the product operand, aligned at the assumed decimal point of the product operand, or both before being copied to it.

For the optional round form of the instruction, specification of a floating-point receiver operand is invalid.

For fixed-point operations in programs that request to be notified of size exceptions, if nonzero digits are truncated from the left end of the resultant value, a *size* (hex 0C0A) exception is signaled.

For floating-point operations involving a fixed-point receiver field (if nonzero digits would be truncated from the left end of the resultant value), an *invalid floating-point conversion* (hex 0C0C) exception is signaled.

For a floating-point product operand, if the exponent of the resultant value is either too large or too small to be represented in the product field, the *floating-point overflow* (hex 0C06) or the *floating-point underflow* (hex 0C07) exception is signaled.

If operands overlap but do not share all of the same bytes, results of operations performed on these operands are not predictable. If overlapped operands share all of the same bytes, the results are predictable when direct addressing is used. If indirect addressing is used (that is, based operands, parameters, strings with variable lengths, and arrays with variable subscripts), the results are not always predictable.

If a decimal to binary conversion causes a *size* (hex 0C0A) exception to be signaled, the binary value contains the correct truncated result only if the decimal value contains 15 or fewer significant nonfractional digits.

Resultant Conditions

- Positive-The algebraic value of the numeric scalar product is positive.
- Negative-The algebraic value of the numeric scalar product is negative.
- Zero-The algebraic value of the numeric scalar product is zero.
- Unordered-The value assigned a floating-point product operand is NaN.

Exceptions

Exception		Operands 1	2	3 [4, 5]	Other
06	Addressing				
	01 space addressing violation	X	X	X	
	02 boundary alignment	X	X	X	
	03 range	X	X	X	
	06 optimized addressability invalid	X	X	X	
08	Argument/parameter				
	01 parameter reference violation	X	X	X	
0C	Computation				
	02 decimal data		X	X	
	06 floating-point overflow	X			
	07 floating-point underflow	X			
	09 floating-point invalid operand		X	X	X
	0A size	X			
	0C invalid floating-point conversion	X			
	0D floating-point inexact result	X			
10	Damage encountered				
	04 system object damage state	X	X	X	X

Exception		Operands 1	2	3 [4, 5]	Other
	44 partial system object damage	X	X	X	X
1C	Machine-dependent exception				
	03 machine storage limit exceeded				X
20	Machine support				
	02 machine check				X
	03 function check				X
22	Object access				
	01 object not found	X	X	X	
	02 object destroyed	X	X	X	
	03 object suspended	X	X	X	
	08 object compressed				X
24	Pointer specification				
	01 pointer does not exist	X	X	X	
	02 pointer type invalid	X	X	X	
2C	Program execution				
	04 invalid branch target				X
2E	Resource control limit				
	01 user profile storage limit exceeded				X
36	Space management				
	01 space extension/truncation				X

Negate (NEG)

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand [3-6]
NEG 1056		Receiver	Source	
NEGI 1856	Indicator options	Receiver	Source	Indicator targets
NEGB 1C56	Branch options	Receiver	Source	Branch targets

Operand 1: Numeric variable scalar.

Operand 2: Numeric scalar.

Operand 3-6:

- *Branch Form*-Branch point, instruction pointer, relative instruction number, or absolute instruction number.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Short forms

Op Code (Hex)	Extender	Operand 1	Operand [2-5]
NEGS 1156		Receiver/Source	
NEGIS 1956	Indicator options	Receiver/Source	Indicator targets
NEGBS 1D56	Branch options	Receiver/Source	Branch targets

Operand 1: Numeric variable scalar.

Operand 2-5:

- *Branch Form*-Branch point, instruction pointer, relative instruction number, or absolute instruction number.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Description: The numeric value in the source operand is changed as if it had been multiplied by a negative one (-1). The result is placed in the receiver operand.

The sign changing of the source operand value (positive to negative and negative to positive) is performed as follows:

- Binary
  - Extract the numeric value and form the twos complement of it.
- Packed/Zoned
  - Extract the numeric value and force its sign to positive if it is negative or to negative if it is positive.
- Floating-point

- Extract the numeric value and force the significand sign to positive if it is negative or to negative if it is positive.

The result of the operation is copied into the receiver operand. If this operand is not the same type as that used in performing the operation, the resultant value is converted to its type. If necessary, the resultant value is adjusted to the length of the receiver operand, aligned at the assumed decimal point of the receiver operand, or both before being copied to it. If significant digits are truncated on the left end of the resultant value, a *size* (hex 0C0A) exception is signaled. An attempt to negate a maximum negative signed binary value to a signed binary scalar of the same size also results in a *size* (hex 0C0A) exception. When the receiver is binary the size exception may be suppressed by using the *suppress binary size exception* attribute on the Create Program (CRTPG) instruction. If a packed or zoned 0 is negated, the result is always positive 0.

When the source floating-point operand represents not-a-number, the sign field of the source is not forced to positive and this value is not altered in the receiver.

For a fixed-point operation, if significant digits are truncated from the left end of the resultant value, a *size* (hex 0C0A) exception is signaled. An attempt to negate a maximum negative binary value into a binary scalar of the same size also results in a *size* (hex 0C0A) exception.

For floating-point operations that involve a fixed-point receiver field, if nonzero digits would be truncated from the left end of the resultant value, an *invalid floating-point conversion* (hex 0C0C) exception is signaled.

For a floating-point receiver operand, if the exponent of the resultant value is either too large or too small to be represented in the receiver field, the *floating-point overflow* (hex 0C06) and the *floating-point underflow* (hex 0C07) exceptions are signaled.

If a decimal to binary conversion causes a *size* (hex 0C0A) exception to be signaled or if the size exception was suppressed, the binary value contains the correct truncated result only if the decimal value contains 15 or fewer significant nonfractional digits.

Resultant Conditions

- Positive-The algebraic value of the receiver operand is positive.
- Negative-The algebraic value of the receiver operand is negative.
- Zero-The algebraic value of the receiver operand is zero.
- Unordered-The value assigned a floating-point receiver operand is NaN.

Exceptions

Exception		Operands 1	Operands 2	Other
06	Addressing			
	01 space addressing violation	X	X	
	02 boundary alignment	X	X	
	03 range	X	X	
	06 optimized addressability invalid	X	X	
08	Argument/parameter			
	01 parameter reference violation	X	X	
0C	Computation			
	02 decimal data		X	

Exception		Operands 1	2	Other
	06 floating-point overflow	X		
	07 floating-point underflow	X		
	09 floating-point invalid operand		X	X
	0A size	X		
	0C invalid floating-point conversion	X		
	0D floating-point inexact result	X		
10	Damage encountered			
	04 system object damage	X	X	X
	44 partial system object damage	X	X	X
1C	Machine-dependent exception			
	03 machine storage limit exceeded			X
20	Machine support			
	02 machine check			X
	03 function check			X
22	Object access			
	01 object not found	X	X	
	02 object destroyed	X	X	
	03 object suspended	X	X	
	08 object compressed			X
24	Pointer specification			
	01 pointer does not exist	X	X	
	02 pointer type invalid	X	X	
2C	Program execution			
	04 invalid branch target			X
2E	Resource control limit			
	01 user profile storage limit exceeded			X
36	Space management			
	01 space extension/truncation			X

Not (NOT)

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand [3-4]
NOT 108A		Receiver	Source	
NOTI 188A	Indicator options	Receiver	Source	Indicator targets
NOTB 1C8A	Branch options	Receiver	Source	Branch targets

Operand 1: Character variable scalar or numeric variable scalar.

Operand 2: Character variable scalar or numeric variable

Operand 3-4:

- *Branch Form*-Branch point, instruction pointer, relative instruction number, or absolute instruction number.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Short forms

Op Code (Hex)	Extender	Operand 1	Operand [2-3]
NOTS 118A		Receiver/Source	
NOTIS 198A	Indicator options	Receiver/Source	Indicator targets
NOTBS 1D8A	Branch options	Receiver/Source	Branch targets

Operand 1: Character variable scalar or numeric variable scalar.

Operand 2-3:

- *Branch Form*-Branch point, instruction pointer, relative instruction number, or absolute instruction number.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Description: The Boolean NOT operation is performed on the string value in the source operand. The resulting string is placed in the receiver operand.

The operands may be character or numeric scalars. They are both interpreted as bit strings. Substringing is supported for both character and numeric operands.

The length of the operation is equal to the length of the source operand.

The bit values of the result are determined as follows:

Source Bit	Result Bit
0	1
1	0

The result value is then placed (left-adjusted) in the receiver operand with truncating or padding taking place on the right. The pad value used in this instruction is a hex 00 byte.

Substring operand references that allow for a null substring reference (a length value of zero) may be specified for operands 1 and 2. The effect of specifying a null substring reference for the source operand is that the result is all zero and the instruction's resultant condition is zero. When a null substring reference is specified for the receiver, a result is not set and the instruction's resultant condition is zero regardless of the value of the source operand.

When the receiver operand is a numeric variable scalar, it is possible that the result produced will not be a valid value for the numeric type. This can occur due to padding with hex 00, due to truncation, or due to the resultant bit string produced by the instruction. The instruction completes normally and signals no exceptions for these conditions.

Resultant Conditions

- Zero-The bit value for the bits of the scalar receiver operand is either all zero or a null substring reference is specified for the receiver.
- Not zero-The bit value for the bits of the scalar receiver operand is not all zero.

Exceptions

Exception		Operands 1	Operands 2	Other
06	Addressing			
	01 space addressing violation	X	X	
	02 boundary alignment	X	X	
	03 range	X	X	
	06 optimized addressability invalid	X	X	
08	Argument/parameter			
	01 parameter reference violation	X	X	
10	Damage encountered			
	04 system object damage state	X	X	X
	44 partial system object damage	X	X	X
1C	Machine-dependent exception			
	03 machine storage limit exceeded			X
20	Machine support			
	02 machine check			X
	03 function check			X
22	Object access			
	01 object not found	X	X	
	02 object destroyed	X	X	
	03 object suspended	X	X	
	08 object compressed			X
24	Pointer specification			
	01 pointer does not exist	X	X	
	02 pointer type invalid	X	X	

Exception		Operands 1	2	Other
2C	Program execution			
	04 invalid branch target			X
2E	Resource control limit			
	01 user profile storage limit exceeded			X
36	Space management			
	01 space extension/truncation			X

Or (OR)

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand 3	Operand [4-5]
OR 1097		Receiver	Source 1	Source 2	
ORI 1897	Indicator options	Receiver	Source 1	Source 2	Indicator targets
ORB 1C97	Branch options	Receiver	Source 1	Source 2	Branch targets

Operand 1: Character variable scalar or numeric variable scalar.

Operand 2: Character scalar or numeric scalar.

Operand 3: Character scalar or numeric scalar.

Operand 4-5:

- *Branch Form*-Branch point, instruction pointer, relative instruction number, or absolute instruction number.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Short forms

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand [3-4]
ORS 1197		Receiver/Source 1	Source 2	
ORIS 1997	Indicator options	Receiver/Source 1	Source 2	Indicator targets
ORBS 1D97	Branch options	Receiver/Source 1	Source 2	Branch targets

Operand 1: Character variable scalar or numeric variable scalar.

Operand 2: Character scalar or numeric scalar.

Operand 3-4:

- *Branch Form*-Branch point, instruction pointer, relative instruction number, or absolute instruction number.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Description: The Boolean OR operation is performed on the string values in the source operands. The resulting string is placed in the receiver operand.

The operands may be character or numeric scalars. They are both interpreted as bit strings. Substringing is supported for both character and numeric operands.

The length of the operation is equal to the length of the longer of the two source operands. The shorter of the two operands is logically padded on the right with hex 00. The excess bytes in the longer operand are assigned to the results.

The bit values of the result are determined as follows:

Source 1 Bit	Source 2 Bit	Result Bit
0	0	0
0	1	1
1	0	1
1	1	1

The result value is then placed (left-adjusted) in the receiver operand with truncating or padding taking place on the right. The pad value used in this instruction is a hex 00.

Substring operand references that allow for a null substring reference (a length value of zero) may be specified for operands 1, 2, and 3. The effect of specifying a null substring reference for one source operand is that the other source operand is ORed with an equal length string of all hex 00s. This causes the value of the other operand to be assigned to the result. When a null substring reference is specified for both source operands, the result is all zero and the instruction's resultant condition is zero. When a null substring reference is specified for the receiver, a result is not set and the instruction's resultant condition is zero regardless of the values of the source operands.

If operands overlap but do not share all of the same bytes, results of operations performed on these operands are not predictable. If overlapped operands share all of the same bytes, the results are predictable when direct addressing is used. If indirect addressing is used (that is, based operands, parameters, strings with variable lengths, and arrays with variable subscripts), the results are not always predictable.

When the receiver operand is a numeric variable scalar, it is possible that the result produced will not be a valid value for the numeric type. This can occur due to padding with hex 00, due to truncation, or due to the resultant bit string produced by the instruction. The instruction completes normally and signals no exceptions for these conditions.

Resultant Conditions

- Zero-The bit value for the bits of the scalar receiver operand is either all zero or a null substring reference is specified for the receiver.
- Not zero-The bit value for the bits of the scalar receiver operand is not all zero.

Exceptions

Exception		Operands 1	2	3	Other
06	Addressing				
	01 space addressing violation	X	X	X	
	02 boundary alignment	X	X	X	
	03 range	X	X	X	
	06 optimized addressability invalid	X	X	X	
08	Argument/parameter				
	01 parameter reference violation	X	X	X	
10	Damage encountered				
	04 system object damage state	X	X	X	X
	44 partial system object damage	X	X	X	X
1C	Machine-dependent exception				

Exception		Operands 1	2	3	Other
	03 machine storage limit exceeded				X
20	Machine support				
	02 machine check				X
	03 function check				X
22	Object access				
	01 object not found	X	X	X	
	02 object destroyed	X	X	X	
	03 object suspended	X	X	X	
	08 object compressed				X
24	Pointer specification				
	01 pointer does not exist	X	X	X	
	02 pointer type invalid	X	X	X	
2C	Program execution				
	04 invalid branch target				X
2E	Resource control limit				
	01 user profile storage limit exceeded				X
36	Space management				
	01 space extension/truncation				X

Remainder (REM)

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand 3	Operand [4-6]
REM 1073		Remainder	Dividend	Divisor	
REMI 1873	Indicator options	Remainder	Dividend	Divisor	Indicator targets
REMB 1C73	Branch options	Remainder	Dividend	Divisor	Branch targets

Operand 1: Numeric variable scalar.

Operand 2: Numeric scalar.

Operand 3: Numeric scalar.

Operand 4-6:

- *Branch Form*-Branch point, instruction pointer, relative instruction number, or absolute instruction number.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Short forms

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand [3-5]
REMS 1173		Remainder/Dividend	Divisor	
REMIS 1973	Indicator options	Remainder/Dividend	Divisor	Indicator targets
REMBS 1D73	Branch options	Remainder/Dividend	Divisor	Branch targets

Operand 1: Numeric variable scalar.

Operand 2: Numeric scalar.

Operand 3-5:

- *Branch Form*-Branch point, instruction pointer, relative instruction number, or absolute instruction number.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Description: The Remainder is the result of dividing the Dividend by the Divisor and placing the remainder in operand 1.

Operands can have packed or zoned decimal, signed or unsigned binary type.

Source operands are the Dividend and Divisor. The receiver operand is the Remainder.

If operands are not of the same type, source operands are converted according to the following rules:

1. If any one of the operands has zoned or packed decimal type, source operands are converted to packed decimal.
2. Otherwise, the binary operands are converted to a like type. Note: unsigned binary(2) scalars are logically treated as signed binary(4) scalars.

Source operands are divided according to their type. Packed decimal operands are divided using packed decimal division. Unsigned binary division is used with unsigned source operands. Signed binary operands are divided using two's complement binary division.

Better performance can be obtained if all operands have the same type. Signed and unsigned binary division execute faster than packed decimal division.

Floating-point is not supported for this instruction.

If the divisor has a numeric value of 0, a *zero divide* (hex 0C0B) exception is signaled. If the dividend has a value of 0, the result of the division is a zero value remainder.

For a decimal operation, the internal quotient value produced by the divide operation is always calculated with a precision of zero fractional digit positions. If necessary, internal alignment of the assumed decimal point for the dividend and divisor operands is performed to insure the correct precision for the resultant quotient value. These internal alignments are not subject to detection of the decimal point alignment exception. An internal quotient and the corresponding remainder value will be calculated for any combination of decimal attributes which may be specified for the instruction's operands. However, as described below, the assignment of the remainder value is limited to that portion of the remainder value which fits in the remainder operand.

If the dividend is shorter than the divisor, it is logically adjusted to the length of the divisor.

The division operation is performed according to the rules of algebra. Unsigned binary is treated as a positive number for the algebra. Before the remainder is calculated, an intermediate quotient is calculated. The attributes of this quotient are derived from the attributes of the dividend and divisor operands as follows:

Dividend	**Divisor**	**Intermediate Quotient**
IM,SIM or SBIN(2)	IM,SIM or SBIN(2)	SBIN(2)
IM,SIM or SBIN(2)	SBIN(4)	SBIN(4)
IM,SIM,SBIN(2) or UBIN(2)	DECIMAL(P2,Q2)	DECIMAL(5+Q2,0)
IM,SIM,SBIN(2) or SBIN(4)	UBIN(2) or UBIN(4)	UBIN(4)
UBIN(2) or UBIN(4)	IM,SIM,SBIN(2) or SBIN(4)	UBIN(4)
UBIN(2) or UBIN(4)	UBIN(2) or UBIN(4)	UBIN(4)
SBIN(4)	IM,SIM or SBIN(2)	SBIN(4)
SBIN(4) or UBIN(4)	DECIMAL(P2,Q2)	DECIMAL(10+Q2,0)
DECIMAL(P1,Q1)	IM,SIM,SBIN(2) or UBIN(2)	DECIMAL(P1,0)
DECIMAL(P1,Q1)	SBIN(4) or UBIN(4)	DECIMAL(P1,0)
DECIMAL(P1,Q1)	DECIMAL(P2,Q2)	DECIMAL(P1-Q1+Q,0) Where Q = Larger of Q1 or Q2

IM = IMMEDIATE
SIM = SIGNED IMMEDIATE
SBIN = SIGNED BINARY

UBIN = UNSIGNED BINARY
DECIMAL = PACKED OR ZONED

After the intermediate quotient numeric value has been determined, the numeric value of the remainder operand is calculated as follows:

Remainder = Dividend - (Quotient*Divisor)

When signed arithmetic is used, the sign of the remainder is the same as that of the dividend unless the remainder has a value of 0. When the remainder has a value of 0, the sign of the remainder is positive.

The resultant value of the calculation is copied into the remainder operand. If this operand is not the same type as that used in performing the operation, the resultant value is converted to its type. If necessary, the resultant value is adjusted to the length of the remainder operand, aligned at the assumed decimal point of the remainder operand, or both before being copied to it.

If significant digits are truncated on the left end of the resultant value, a *size* (hex 0C0A) exception is signaled for those programs that request to be notified of size exceptions.

If a decimal to binary conversion causes a *size* (hex 0C0A) exception to be signaled in programs that request to be notified of size exceptions, the binary value contains the correct truncated result only if the decimal value contains 15 or fewer significant nonfractional digits.

Resultant Conditions

- Positive-The algebraic value of the numeric scaler remainder is positive.
- Negative-The algebraic value of the numeric scaler remainder is negative.
- Zero-The algebraic value of the numeric scaler remainder is zero.

Exceptions

Exception		Operands 1	2	3	Other
06	Addressing				
	01 space addressing violation	X	X	X	
	02 boundary alignment	X	X	X	
	03 range	X	X	X	
	06 optimized addressability invalid	X	X	X	
08	Argument/parameter				
	01 parameter reference violation	X	X	X	
0C	Computation				
	02 decimal data		X	X	
	0A size	X			
	0B zero divide			X	
10	Damage encountered				
	04 system object damage state	X	X	X	X
	44 partial system object damage	X	X	X	X
1C	Machine-dependent exception				

Exception		Operands 1	2	3	Other
	03 machine storage limit exceeded				X
20	Machine support				
	02 machine check				X
	03 function check				X
22	Object access				
	01 object not found	X	X	X	
	02 object destroyed	X	X	X	
	03 object suspended	X	X	X	
	08 object compressed				X
24	Pointer specification				
	01 pointer does not exist	X	X	X	
	02 pointer type invalid	X	X	X	
2C	Program execution				
	04 invalid branch target				X
2E	Resource control limit				
	01 user profile storage limit exceeded			X	
36	Space management				
	01 space extension/truncation			X	

Scale (SCALE)

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand 3	Operand [4-7]
SCALE 1063		Receiver	Source	Scale factor	
SCALEI 1863	Indicator options	Receiver	Source	Scale factor	Indicator targets
SCALEB 1C63	Branch options	Receiver	Source	Scale factor	Branch targets

Operand 1: Numeric variable scalar.

Operand 2: Numeric scalar.

Operand 3: Binary(2) scalar.

Operand 4-7:

- *Branch Form*-Branch point, instruction pointer, relative instruction number, or absolute instruction number.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Short forms

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand [3-6]
SCALES 1163		Receiver/Source	Scale factor	
SCALEIS 1963	Indicator options	Receiver/Source	Scale factor	Indicator targets
SCALEBS 1D63	Branch options	Receiver/Source	Scale factor	Branch targets

Operand 1: Numeric variable scalar.

Operand 2: Binary(2) scalar.

Operand 3-6:

- *Branch Form*-Branch point, instruction pointer, relative instruction number, or absolute instruction number.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Description: The scale instruction performs numeric scaling of the source operand based on the scale factor and places the results in the receiver operand. The numeric operation is as follows:

```
Operand 1 = Operand 2 *(B**N)
```

where:

N is the binary integer value of the scale operand. It can be positive, negative, or 0. If N is 0, then the operation simply copies the source operand value into the receiver operand.

B is the arithmetic base for the type of numeric value in the source operand.

Base Type	B
Binary	2
Packed/Zoned	10
Floating-point	2

The scale operation is a shift of N binary, packed, or zoned digits. The shift is to the left if N is positive, to the right if N is negative. For a floating-point source operand, the scale operation is performed as if the source operand is multiplied by a floating-point value of 2**N.

If the source and receiver operands have different attributes, the scaling operation is done in an intermediate field with the same attributes as the source operand. If a fixed-point scaling operation causes nonzero digits to be truncated on the left end of the intermediate field, a *size* (hex 0C0A) exception is signaled. For a floating-point scaling operation, the *floating-point overflow* (hex 0C06) and the *floating-point underflow* (hex 0C07) exceptions can be signaled during the calculation of the intermediate result.

The resultant value of the calculation is copied into the receiver operand. If this operand is not the same type as that used in performing the operation, the resultant value is converted to its type. If necessary, the resultant value is adjusted to the length of the receiver operand, aligned at the assumed decimal point of the receiver operand, or both before being copied to it. For fixed-point operations, if nonzero digits are truncated off the left end of the resultant value, a *size* (hex 0C0A) exception is signaled.

For floating-point operations involving fixed-point receiver fields, if nonzero digits would be truncated from the left end of the resultant value, an *invalid floating-point conversion* (hex 0C0C) exception is signaled.

For floating-point receiver fields, if the exponent of the resultant value is either too large or too small to be represented in the receiver field, the *floating-point overflow* (hex 0C06) or *floating-point underflow* (hex 0C07) exception is signaled.

A *scalar value invalid* (hex 3203) exception is signaled if the value of N is beyond the range of the particular type of the source operand as specified in the following table.

Source Operand Type	Maximum Value of N
Signed Binary(2)	$-15 \leq N \leq 15$
Unsigned Binary(2)	$-16 \leq N \leq 16$
Signed Binary(4)	$-31 \leq N \leq 31$
Unsigned Binary(4)	$-32 \leq N \leq 32$
Decimal(P,Q)	$-31 \leq N \leq 31$

For a scale operation in floating-point, no limitations are placed on the values allowed for N other than the implicit limits imposed due to the floating-point overflow and underflow exceptions.

Limitations: The following are limits that apply to the functions performed by this instruction.

If a decimal to binary conversion causes a *size* (hex 0C0A) exception to be signaled, the binary value contains the correct truncated result only if the decimal value contains 15 or fewer significant nonfractional digits.

Resultant Condition

- Positive-The algebraic value of the receiver operand is positive.
- Negative-The algebraic value of the receiver operand is negative.
- Zero-The algebraic value of the receiver operand is zero.

- Unordered-The value assigned a floating-point receiver operand is NaN.

Exceptions

Exception		Operands 1	2	3	Other
06	Addressing				
	01 space addressing violation	X	X	X	
	02 boundary alignment	X	X	X	
	03 range	X	X	X	
	06 optimized addressability invalid	X	X	X	
08	Argument/parameter				
	01 parameter reference violation	X	X	X	
0C	Computation				
	02 decimal data		X		
	06 floating-point overflow	X			
	07 floating-point underflow	X			
	09 floating-point invalid operand		X		X
	0A size	X			
	0C invalid floating-point conversion	X			
	0D floating-point inexact result	X			
10	Damage encountered				
	04 system object damage state	X	X	X	X
	44 partial system object damage	X	X	X	X
1C	Machine-dependent exception				
	03 machine storage limit exceeded				X
20	Machine support				
	02 machine check				X
	03 function check				X
22	Object access				
	01 object not found	X	X	X	
	02 object destroyed	X	X	X	
	03 object suspended	X	X	X	
	08 object compressed				X
24	Pointer specification				
	01 pointer does not exist	X	X	X	
	02 pointer type invalid	X	X	X	
2C	Program execution				
	04 invalid branch target				X
2E	Resource control limit				
	01 user profile storage limit exceeded				X

Exception		Operands 1	2	3	Other
32	Scalar specification				
	03 scalar value invalid			X	
36	Space management				
	01 space extension/truncation				X

Scan (SCAN)

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand 3	Operand [4-5]
SCAN 10D3		Receiver	Base	Compare operand	
SCANB 1CD3	Branch options	Receiver	Base	Compare operand	Branch targets
SCANI 18D3	Indicator options	Receiver	Base	Compare operand	Indicator targets

Operand 1: Binary variable scalar or binary array.

Operand 2: Character variable scalar.

Operand 3: Character scalar.

Operand 4-5:

- *Branch Form*-Branch point, instruction pointer, relative instruction number, or absolute instruction number.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Description: The character string value of the base operand is scanned for occurrences of the character string value of the compare operand.

The base and compare operands must both be character strings. The length of the compare operand must not be greater than that of the base string.

The operation begins at the left end of the base string and continues character by character, from left to right, comparing the characters of the base string with those of the compare operand. The length of the comparisons are equal to the length of the compare operand value and function as if they were being compared in the Compare Bytes Left-Adjusted instruction.

If a set of bytes that match the compare operand is found, the binary value for the ordinal position of its leftmost base string character is placed in the receiver operand.

If the receiver operand is a scalar, only the first occurrence of the compare operand is noted. If it is an array, as many occurrences as there are elements in the array are noted.

The operation continues until no more occurrences of the compare operand can be noted in the receiver operand or until the number of characters (bytes) remaining to be scanned in the base string is less than the length of the compare operand.

When the second condition occurs, the receiver value is set to 0. If the receiver operand is an array, all its remaining elements are also set to 0.

The base operand and the compare operand can be variable length substring compound operands.

Substring operand references that allow for a null substring reference (a length value of zero) may be specified for operands 2 and 3. The effect of specifying a null substring reference for the compare operand or both operands is that the receiver is set to zero (no match found) and the instruction's resultant condition is null compare operand. Specifying a null substring reference for just the base operand is not allowed due to the requirement that the length of the compare operand must not be greater than that of the base string.

Resultant Conditions

- Zero-The numeric value(s) of the receiver operand is zero. When the receiver operand is an array, the resultant condition is zero if all elements are zero. One of these two conditions will result when the compare operand is not a null substring reference.
- Positive-The numeric value(s) of the receiver operand is positive.
- Null compare operand-The compare operand is a null substring reference; therefore, the receiver has been set to zero which indicates that no occurrences were found.

Exceptions

Exception		Operands 1	2	3	Other
06	Addressing				
	01 space addressing violation	X	X	X	
	02 boundary alignment	X	X	X	
	03 range	X	X	X	
	06 optimized addressability invalid	X	X	X	
08	Argument/parameter				
	01 parameter reference violation	X	X	X	
0C	Computation				
	08 length conformance		X	X	
10	Damage encountered				
	04 system object damage state	X	X	X	X
	44 partial system object damage	X	X	X	X
1C	Machine-dependent exception				
	03 machine storage limit exceeded				X
20	Machine support				
	02 machine check				X
	03 function check				X
22	Object access				
	01 object not found	X	X	X	
	02 object destroyed	X	X	X	
	03 object suspended	X	X	X	
	08 object compressed				X
24	Pointer specification				
	01 pointer does not exist	X	X	X	
	02 pointer type invalid	X	X	X	
2C	Program execution				
	04 invalid branch target				X
2E	Resource control limit				
	01 user profile storage limit exceeded				X
36	Space management				

Exception	Operands 1	2	3	Other
01 space extension/truncation				X

Scan with Control (SCANWC)

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand 3	Operand 4	Operand [5-8]
SCANWC 10E4		Base locator	Controls	Options	Escape target or null	
SCANWCB 1CE4	Branch options	Base locator	Controls	Options	Escape target or null	Branch targets
SCANWCI 18E4	Indicator options	Base locator	Controls	Options	Escape target or null	Indicator targets

Operand 1: Space pointer.

Operand 2: Character(8) variable scalar (fixed length)

Operand 3: Character(4) constant scalar (fixed length)

Operand 4: Instruction number, relative instruction number, branch point, instruction pointer, instruction definition list element, or null.

Operand 5-8:

- *Branch Form*-Branch point, instruction pointer, relative instruction number, or absolute instruction number.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Description: The base string to be scanned is specified by the base locator and controls operands. The base locator addresses first character of the base string. The controls specifies the length of the base string in the base length field.

The scan operation begins at the left end of the base string and continues character by character, left-to-right. The scan operation can be performed on a base string which contains all simple (1-byte) or all extended (2-byte) character codes or a mixture of the two. When the base string is being interpreted in simple character mode, the operation moves through the base string one byte at a time. When the base string is being interpreted in extended character mode, the operation moves through the base string 2 bytes at a time. The character string value of the base operand is scanned for occurrences of a character value satisfying the criteria specified in the control and options operands.

The scan is completed by updating the base locator and controls operands with scan status when a character value being scanned for is found, the end of the base string is encountered, or an escape code is encountered when the escape target operand is specified. The base locator is set with addressability to the character (simple or extended) which caused the instruction to complete execution. The controls operand is set with information which identifies the mode (simple or extended) of the base string character addressed by the base locator and which provides for resumption of the scan operation with minimal overhead.

The controls and options operands specify the modes to be used in interpreting characters during the scan operation. Characters can be interpreted in one of two character modes: simple (1-byte) and extended (2-byte). Additionally, the base string can be scanned in one of two scan modes, mixed (base string may contain a mixture of both character modes) and nonmixed (base string contains one mode of characters).

When the mixed scan mode is specified in the options operand, the base string is interpreted as containing a mixture of simple and extended character codes. The mode, simple or extended, with which the string is to be interpreted, is controlled initially by the base mode indicator in the controls operand and thereafter by mode control characters imbedded in the base string. The mode control characters are as follows:

- Hex 0E = Shift out (SO) of simple character mode to extended mode.
- Hex 0F = Shift in (SI) to simple character mode from extended mode. This is only recognized if it occurs in the first byte position of an extended character code.

When the nonmixed scan mode is specified in the options operand, the base string is interpreted using only the character mode specified by the base mode indicator in the controls operand. Character mode shifting can not occur because no mode control characters are recognized when scanning in nonmixed mode.

The base locator operand is a space pointer which is both input to and output from the instruction. On input, it locates the first character of the base string to be processed. On output, it locates the character of the base string which caused the instruction to complete.

The controls operand must be a character scalar which specifies additional information to be used to control the scan operation. It must be at least 8 bytes long and have the following format:

- Controls operand — Char(8)
  - Control indicators — Char(1)
  - Reserved — Char(1)
  - Comparison characters — Char(2)
  - Reserved — Char(1)
  - Base end — Char(3)
    - Instruction work area — Char(1)
    - Base length — Char(2)

Only the first 8 bytes of the controls operand are used. Any excess bytes are ignored. Reserved fields must contain binary zeros. The **control indicators** field has the following format:

- Control indicators — Char(1)
  - Base mode — Bit 0

    0 = Simple
    1 = Extended
  - Comparison character mode — Bit 1

    0 = Simple
    1 = Extended
  - Reserved (must be 0) — Bit 2-6
  - Scan state — Bit 7

    0 = Resume scan
    1 = Start scan

The **base mode** is both input to and output from the instruction. In either case, it specifies the mode of the character in the base string currently addressed by the base locator.

The **comparison character mode** is not changed by the instruction. It specifies the mode of the comparison character contained in the controls operand.

The **scan state** is both input to and output from the instruction. As input, it indicates whether the scan operation for the base string is being started or resumed. If it is being *started*, the instruction assumes that the *base length* value in the *base end* field of the controls operand specifies the length of the base string, and the instruction work area value is ignored. If it is being *resumed*, the instruction assumes the *base end* field has been set by a prior start scan execution of the instruction with an internal machine value identifying the end of the base string.

For a *start scan* execution of the instruction, the *scan state* field is reset to indicate resume scan to provide for subsequent resumption of the scan operation. Additionally, for a *start scan* execution of the instruction, the *base end* field is set with an internally optimized value which identifies the end of the base string being scanned. This value then overlays the values which were in the instruction work area and base length fields on input to the instruction. Predictable operation of the instruction on a *resume scan* execution depends upon this base end field being left intact with the value set by the start scan execution.

For a *resume scan* execution of the instruction, the *scan state* and *base end* fields are unchanged.

The **comparison character** is input to the instruction. It specifies a character code to be used in the comparisons performed during the scanning of the base string. The *comparison character mode* in the control indicators specifies the mode (simple or extended) of the comparison character. If it is a *simple* character, the first byte of the *comparison character* field is ignored and the *comparison character* is assumed to be specified in the second byte. If it is an *extended* character, the *comparison character* is specified as a 2-byte value in the *comparison character* field.

The **base end** field is both input to and output from the instruction. It contains data which identifies the end of the base string. Initially, for a start scan execution of the instruction, it contains the length of the base string in the **base length** field. Additionally, the *base end* field is used to retain information over multiple instruction executions which provides for minimizing the overhead required to resume the scan operation for a particular base string. This information is set on the initial *start scan* execution of the instruction and is used during subsequent *resume scan* executions of the instruction to determine the end of the base string to be scanned. If the end of the base string being scanned must be altered during iterative usage of this instruction, a *start scan* execution of the instruction must be performed to provide for correctly resetting the internally optimized value to be stored in the *base end* from the values specified in the base locator operand and *base length* field.

For the special case of a *start scan* execution where a length value of zero (no characters to scan) is specified in the *base length* field, the instruction results in a *not found* resultant condition. In this case, the base locator is not verified and the *scan state* indicator, the *base end* field, and the base locator are not changed.

The options operand must be a character scalar which specifies the options to be used to control the scan operation. It must be at least 4 bytes in length and has the following format:

• Options operand	Char(4)
— Options indicators	Char(1)
— Reserved	Char(3)

The options operand must be specified as a constant character scalar.

Only the first 4 bytes of the options operand are used. Any excess bytes are ignored. Reserved fields must contain binary zeros. The **option indicators** field has the following format:

• Option indicators	Char(1)
— Reserved	Bit 0
— Scan mode	Bit 1

0 = Mixed
1 = Nonmixed

- Reserved — Bits 2-3
- Comparison relation — Bits 4-6
  - Equal, (=) relation — Bit 4
  - Less than, (<) relation — Bit 5
  - Greater than, (>) relation — Bit 6

    0 = No match on relation
    1 = Match on relation
- Reserved — Bit 7

The **scan mode** specifies whether the base string contains a mixture of character modes, or contains all one mode of characters; that is, whether or not mode control characters should be recognized in the base string. *Mixed* specifies that there is a mixture of character modes and, therefore, mode control characters should be recognized. *Nonmixed* specifies that there is not a mixture of character modes and, therefore, mode control characters should not be recognized. Note that the *base mode* indicator in the controls operand specifies the character mode of the base string character addressed by the base locator.

The **comparison relation** specifies the relation or relations of the comparison character to characters of the base string which will satisfy the scan operation and cause completion of the instruction with one of the *high, low, or equal* resultant conditions. Multiple relations may be specified in conjunction with one another. Specifying all relations insures a match against any character in the base string which is of the same mode as the comparison character. Specifying no relation insures a *not found* resultant condition, in the absence of an escape due to verification, regardless of the values of the characters in the base string which match the mode of the comparison character.

An example of comparison scanning is a scan of simple mode characters for a value less than hex 40. This could be done by specifying a *comparison character* of hex 40 and a *comparison relation* of *greater* than in conjunction with a branch option for the resultant condition of *high*. This could also be done by specifying a *comparison character* of hex 3F and *comparison relations* of *equal* and *greater* than in conjunction with branch options for *equal* and *high*. The target of the branch options in either case would be the instructions to process the character less than hex 40 in value.

The escape target operand controls the verification of bytes of the base string for values less than hex 40. Verification, if requested, is always performed in conjunction with whatever comparison processing has been requested. That is, verification is performed even if no comparison relation is specified. This operand is discussed in more detail in the following material.

During the scan operation, the characters of the base string which are not of the same mode as the comparison character are skipped over until the mode of the characters being processed is the same as the mode of the comparison character. The operation then proceeds by comparing the comparison character with each of the characters of the base string. These comparisons behave as if the characters were being compared in the Compare Bytes Left Adjusted instruction.

If a base string character satisfying the criteria specified in the controls and options operands is found, the base locator is set to address the first byte of it, the *base mode* indicator is set to indicate the mode of the base string as of that character, and the instruction is completed with the appropriate resultant condition based on the *comparison relation* (*high, low*, or *equal*) of the *comparison character* to the base string character.

If a matching base string character is not found prior to encountering a mode change, the characters of the base string are again skipped over until the mode of the characters being processed is the same as the mode of the comparison character before comparisons are resumed.

If a matching base string character is not found prior to encountering the end of the base string, the base location is set to address the first byte of the character encountered at the end of the base string, the *base mode* indicator is set to indicate the mode of the base string as of that character, and the instruction is completed with the *not found* resultant condition. A mode control string results in the changing of the base string mode, but the base locator is left addressing the mode control character.

If the escape target operand is specified (operand 4 is not null), verifications are performed on the characters of the base string prior to their being skipped or compared with the comparison character. Each byte of the base string is checked for a value less than hex 40. Additionally, for a *mixed* scan mode, when such a value is encountered, it is then determined if it is a valid mode control character.

- Hex 0E (SO) when the base string is being interpreted in simple character mode.
- Hex 0F (SI) in the left byte of the character code when the base string is being interpreted in extended character mode.

If a byte value of less than hex 40 is not a valid mode control character, it is considered to be an escape code. The base locator is set to address the first byte of the base string character (simple or extended) which contains the escape code, the *base mode* indicator is set to indicate the mode of the base string as of that character, and a branch is taken to the target specified by the escape target operand. When the escape target branch is performed, the value of any optional indicator operands is meaningless.

If the escape target operand is not specified (operand 4 is null), verifications of the character codes in the base string are not performed. However, for a *mixed* scan mode, mode control values are always processed as described previously under the discussion of the mixed scan mode.

If possible, use a Space Pointer Machine Object for the base locator, operand 1. Appreciably less overhead is incurred in accessing and storing the value of the base locator if this is done.

If possible, specify through its ODT definition, the controls operand on an 8-byte multiple (doubleword) boundary relative to the start of the space containing it. Appreciably less overhead is incurred in accessing and storing the value of the controls if this is done.

For the case where a base string is to be just scanned for byte values less than hex 40, two techniques can be used.

- A direct simple mode scan for a value less than hex 40 without usage of the escape target verification feature.
- A scan for any character with usage of the escape target verification feature.

  The direct scan approach, the former, is the more efficient.

The following diagram defines the various conditions which can be encountered at the end of the base string and what the base locator addressability is in each case. The solid vertical line represents the end of the base string. The dashes represent the bytes before and after the base string end. The V is positioned over the byte addressed by the base locator in each case. These are the conditions which can be encountered when the base locator input to the instruction addresses a byte prior to the base string end. When the base length field specifies a value of zero for a start scan execution of the instruction, or the input base locator addresses a point beyond the end of the instruction, no processing is performed and the instruction is immediately completed with the *not found* resultant condition.

Addressability	Ending Condition	Instruction Response
V	(One byte code at string end)	
	• Simple character	• Appropriate resultant condition indicating found or not found
	• Shift in/out encountered	• Mode shift performed, and not found resultant condition
	• Escape code in simple character	• Branch taken
V	(Extended character split across string end)	
	• Extended character	• Not found resultant condition
	• Escape code in extended character	• Branch taken
V	(Extended character at string end)	
	• Extended character	• Appropriate resultant condition indicating found or not found
	• Escape code in extended character	• Branch taken

AAC012-0

An analysis of the diagram shows that normally, after appropriate processing for the particular *found, not found, or escape* condition, the scan can be restarted at the byte of data which would follow the base string end in the data stream being scanned. Any mode shift required by an ending mode control character will have been performed.

However, one ending condition may require subsequent resumption of the scan at the character encountered at the end of the base string. This is the case where the instruction completes with the *not found* resultant condition and the base string ends with an extended character split across string end. That is, the *base mode* indicator specifies *extended* mode, the base locator addresses the last byte of the base string, and that byte value is not a shift out, hex 0E character. In this case, complete verification of the extended character and relation comparison could not be performed. If this extended character is to be processed, it must be done through another execution of the Scan instruction where both bytes of the character can be input to the instruction within the confines of the base string.

Resultant Conditions

- Equal: A character value was found in the base string which satisfies the criteria specified in the controls and options operands in that the comparison character is of equal string value to the base string character.
- High: A character value was found in the base string which satisfies the criteria specified in the controls and options operands in that the comparison character is of higher string value to the base string character.

- Low: A character value was found in the base string which satisfies the criteria specified in the controls and options operands in that the comparison character is of lower string value to the base string character.
- Not found: A character value was not found in the base string which satisfied the criteria specified in the controls and options operands.

Exceptions

Exception		Operands 1	2	3	4	Other
06	Addressing					
	01 space addressing violation	X	X	X	X	
	02 boundary alignment	X	X	X	X	
	03 range	X	X	X	X	
	06 optimized addressability invalid	X	X	X	X	
08	Argument/parameter					
	01 parameter reference violation	X	X	X	X	
0C	Computation					
	08 length conformance		X	X		
10	Damage encountered					
	04 System object damage state					X
	44 partial system object damage					X
1C	Machine-dependent exception					
	03 machine storage limit exceeded					X
20	Machine support					
	02 machine check					X
	03 function check					X
22	Object access					
	01 object not found	X	X	X	X	
	02 object destroyed	X	X	X	X	
	03 object suspended	X	X	X	X	
	08 object compressed					X
24	Pointer specification					
	01 pointer does not exist	X	X	X	X	
	02 pointer type invalid	X	X	X	X	
2C	Program execution					
	04 invalid branch target					X
2E	Resource control limit					
	01 user profile storage limit exceeded					X
32	Scalar specification					
	01 scalar type invalid	X	X	X	X	
	03 scalar value invalid		X	X		

Exception		Operands 1	2	3	4	Other
36	Space management					
	01 space extension/truncation					X

Search (SEARCH)

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand 3	Operand 4	Operand [5-6]
SEARCH 1084		Receiver	Array	Find	Location	
SEARCHB 1C84	Branch options	Receiver	Array	Find	Location	Branch targets
SEARCHI 1884	Indicator options	Receiver	Array	Find	Location	Indicator targets

Operand 1: Binary variable scalar or binary variable array.

Operand 2: Character array or numeric array.

Operand 3: Character variable scalar or numeric variable scalar.

Operand 4: Binary scalar.

Operand 5-6:

- *Branch Form*-Branch point, instruction pointer, relative instruction number, or absolute instruction number.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Description: The portions of the array operand indicated by the location operand are searched for occurrences of the value indicated in the find operand.

The operation begins with the first element of the array operand and continues element by element, comparing those characters of each element (beginning with the character indicated in the location operand) with the characters of the find operand. The location operand contains an integer value representing the relative location of the first character in each element to be used to begin the compare.

The integer value of the location operand must range from 1 to L, where L is the length of the array operand elements; otherwise, a *scalar value invalid* (hex 3203) exception is signaled. A value of 1 indicates the leftmost character of each element.

The array and find operands can be either character or numeric. Any numeric operands are interpreted as logical character strings. The compares between these operands are performed at the length of the find operand and function as if they were being compared in the Compare Bytes Left-Adjusted instruction.

The length of the find operand must not be so large that it exceeds the length of the array operand elements when used with the location operand value. The array element length used is the length of the array scalar elements and not the length of the entire array element, which can be larger in non-contiguous arrays.

As each occurrence of the find value is encountered, the integer value of the index for this array element is placed in the receiver operand. If the receiver operand is a scalar, only the first element containing the find value is noted. If the receiver operand is an array, as many occurrences as there are elements within the receiver array are noted.

If the value of the index for an array element containing an occurrence of the find value is too large to be contained in the receiver, a *size* (hex 0C0A) exception is signaled.

The operation continues until no more occurrences of elements containing the find value can be noted in the receiver operand or until the array operand has been completely searched. When the second condition occurs, the receiver value is set to LB-1, where LB is the value of the lower bound index of the array. If LB is the most negative 32-bit integer, then LB-1 is the most positive 32-bit integer; otherwise, LB-1 is 1 less than LB. If the receiver operand is an array, all its remaining elements are also set to LB-1. The find operand can be a variable length substring compound operand.

Resultant Conditions *::* The numeric value(s) of the receiver operand is either LB-1 or in the range LB through UB, where UB is the value of the upper bound index of the array. When the receiver is LB-1, the resultant condition is *zero*. When the receiver is in the range LB through UB, the resultant condition is *positive*. When the receiver is an array, the resultant condition is *zero* if all elements are LB-1; otherwise, it is *positive*. The resultant condition is unpredictable when the ***no binary size exception*** program template option is used.

Exceptions

Exception		Operands 1	2	3	4	Other
06	Addressing					
	01 space addressing violation	X	X	X	X	
	02 boundary alignment	X	X.	X	X	
	03 range	X	X	X	X	
	06 optimized addressability invalid	X	X	X	X	
08	Argument/parameter					
	01 parameter reference violation	X	X	X	X	
0C	Computation					
	08 length conformance		X	X		
	0A size	X				
10	Damage encountered					
	04 system object damage state	X	X	X	X	X
	44 partial system object damage	X	X	X	X	X
1C	Machine-dependent exception					
	03 machine storage limit exceeded					X
20	Machine support					
	02 machine check					X
	03 function check					X
22	Object access					
	01 object not found	X	X	X	X	
	02 object destroyed	X	X	X	X	
	03 object suspended	X	X	X	X	
	08 object compressed					X
24	Pointer specification					
	01 pointer does not exist	X	X	X	X	

Exception		Operands 1	2	3	4	Other
	02 pointer type invalid	X	X	X	X	
2C	Program execution					
	04 invalid branch target					X
2E	Resource control limit					
	01 user profile storage limit exceeded					X
32	Scalar specification					
	01 scalar type invalid	X	X	X	X	
	03 scalar value invalid				X	
36	Space management					
	01 space extension/truncation					X

Set Bit in String (SETBTS)

Op Code (Hex)	Operand 1	Operand 2
101E	Source	Offset

Operand 1: Character variable scalar (fixed length) or numeric variable scalar.

Operand 2: Binary scalar.

Description: Sets the bit of the receiver operand as indicated by the bit offset operand.

The selected bit from the receiver operand is set to a value of binary 1.

The receiver operand can be a character or numeric variable. The leftmost bytes of the receiver operand are used in the operation. The receiver operand is interpreted as a bit string with the bits numbered left to right from 0 to the total number of bits in the string minus 1.

The offset operand indicates which bit of the receiver operand is to be set, with a offset of zero indicating the leftmost bit of the leftmost byte of the receiver operand. This value may be specified as a constant or any valid binary scalar variable.

If a offset value less than zero or beyond the length of the receiver is specified a *scalar value invalid* (hex 3203) exception is signaled.

Exceptions

Exception		Operands 1	Operands 2	Other
06	Addressing			
	01 Spacing addressing violation	X	X	
	02 Boundary alignment violation	X	X	
	03 Range	X	X	
	06 Optimized addressability invalid	X	X	
08	Argument/parameter			
	01 Parameter reference violation	X	X	
10	Damage encountered			
	04 System object damage state			X
	44 Partial system object damage			X
1C	Machine-dependent exception			
	03 Machine storage limit exceeded			X
20	Machine support			
	02 Machine check			X
	03 Function check			X
22	Object access			
	02 Object destroyed	X	X	
	03 Object suspended	X	X	
	08 object compressed			X

Exception		Operands 1	Operands 2	Other
24	Pointer specification			
	01 Pointer does not exist	X	X	
	02 Pointer type invalid	X	X	
2E	Resource control limit			
	01 user profile storage limit exceeded			X
32	Scalar specification			
	01 Scalar type invalid	X	X	
	03 Scalar value invalid		X	
36	Space management			
	01 space extension/truncation			X

Set Instruction Pointer (SETIP)

Op Code (Hex)	Operand 1	Operand 2
1022	Receiver	Branch target

Operand 1: Instruction pointer.

Operand 2: Instruction number, relative instruction number, or branch point.

Description: The value of the branch target (operand 2) is used to set the value of the instruction pointer specified by operand 1. The instruction number indicated by the branch target must provide the address of an instruction within the program containing the Set Instruction Pointer instruction.

Exceptions

Exception		Operands 1	2	Other
06	Addressing			
	01 space addressing violation	X		
	02 boundary alignment	X		
	03 range	X		
	06 optimized addressability invalid	X		
08	Argument/parameter			
	01 parameter reference violation	X		
10	Damage encountered			
	04 system object damage state	X	X	X
	44 partial system object damage	X	X	X
1C	Machine-dependent exception			
	03 machine storage limit exceeded			X
20	Machine support			
	02 machine check			X
	03 function check			X
22	Object access			
	01 object not found	X		
	02 object destroyed	X		
	03 object suspended	X		
	08 object compressed			X
24	Pointer specification			
	01 pointer does not exist	X		
	02 pointer type invalid	X		
2C	Program execution			
	04 branch target invalid		X	
2E	Resource control limit			
	01 user profile storage limit exceeded			X

Exception		Operands 1	2	Other
36	Space management			
	01 space extension/truncation			X

Store and Set Computational Attributes (SSCA)

Op Code (Hex)	Operand 1	Operand 2	Operand 3
107B	Receiver	Source	Controls

Operand 1: Character(5) variable scalar (fixed length).

Operand 2: Character(5) scalar (fixed length) or null.

Operand 3: Character(5) scalar (fixed length) or null.

Description: This instruction stores and optionally sets the attributes for controlling computational operations for the process this instruction is executed in.

The receiver is assigned the values that each of the computational attributes had at the start of execution of the instruction. It has the same format and bit assignment as the source.

The source specifies new values for the computational attributes for the process. The particular computational attributes that are selected for modification are determined by the controls operand. The source operand has the following format:

- Floating-point exception masks — Char(2)

  0 = Disabled (exception is masked)
  1 = Enabled (exception is unmasked)

  - Reserved (binary 0) — Bits 0-9
  - Floating-point overflow — Bit 10
  - Floating-point underflow — Bit 11
  - Floating-point zero divide — Bit 12
  - Floating-point inexact result — Bit 13
  - Floating-point invalid operand — Bit 14
  - Reserved (binary 0) — Bit 15

- Floating-point exception occurrence flags — Char(2)

  0 = Exception has not occurred
  1 = Exception has occurred

  - Reserved (binary 0) — Bits 0-9
  - Floating-point overflow — Bit 10
  - Floating-point underflow — Bit 11
  - Floating-point zero divide — Bit 12
  - Floating-point inexact result — Bit 13
  - Floating-point invalid operand — Bit 14
  - Reserved (binary 0) — Bit 15

- Modes — Char(1)
  - Reserved — Bit 0
  - Floating-point rounding mode — Bits 1-2

    00 = Round toward positive infinity
    01 = Round toward negative infinity

10= Round toward zero
11= Round to nearest (default)

— Reserved Bits 3-7

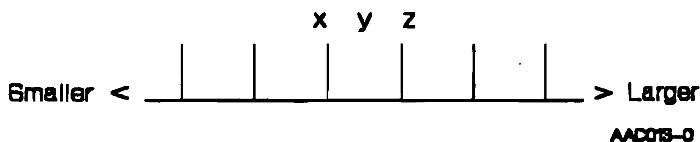
The controls operand is used to select those attributes that are to be set from the bit values of the source operand. The format of the controls is the same as that for the source. A value of one for a bit in controls indicates that the corresponding computational attribute for the process is to be set from the value of that bit of the source. A value of zero for a bit in controls indicates that the corresponding computational attribute for the process is not to be changed, and will retain the value it had prior to this instruction. For an attribute controlled by a multiple-bit field, such as the rounding modes, all of the bits in the field must be ones or all must be zeros. A mixture of ones and zeros in such a field results in a *scalar value invalid* (hex 3801) exception.

If the source and controls operands are both null, the instruction will just return the current computational attributes. If the source is specified, the computational attributes of the process are modified under control of the controls operand. If the source operand is specified and the controls operand is null, the instruction will change all of the computational attributes to the values specified in the source. If the source operand is null and the controls operand is specified, an *invalid operand type* (hex 2A06) exception is signaled on the Create Program instruction.

With the **floating-point exception masks** field, it is possible to unmask/mask the exception processing and handling for each of the five floating-point exceptions. If an exception that is unmasked occurs, then the corresponding *floating point exception occurrence* bit is set, and the exception is signaled. If an exception that is masked occurs, the exception is not signaled, but the *floating pointer exception occurrence* flag is still set to indicate the occurrence of the exception.

The **floating-point exception occurrence** flag for each exception may be set or cleared by this instruction from the source operand under control of the controls operand.

Unless specified otherwise by a particular instruction, or precluded due to implicit conversions, all floating-point operations are performed as if correct to infinite precision, and then rounded to fit in a destinations format while potentially signaling an exception that the result is inexact. To allow control of the floating-point rounding operations performed within a process, four **floating-point rounding modes** are supported. Assume y is the infinitely precise number that is to be rounded, bracketed most closely by x and z, where x is the largest representable value less than y and z is the smallest representable value greater than y. Note that x or z may be infinity. The following diagram shows this relationship of x, y, and z on a scale of numerically progressing values where the vertical bars denote values representable in a floating-point format.



Given the above, if y is not exactly representable in the receiving field format, the rounding modes change y as follows:

Round to nearest with round to even in case of a tie is the default rounding mode in effect upon the initiation of a process. For this rounding mode, y is rounded to the closer of x or z. If they are equally close, the even one (the one whose least significant bit is a zero) is chosen. For the purposes of this mode of rounding, infinity is treated as if it was even. Except for the case of y being rounded to a value of infinity, the rounded result will differ from the infinitely precise result by at most half of the least significant digit position of the chosen value. This rounding mode differs slightly from the decimal round algorithm performed for the optional round form of an instruction. This rounding mode would round a value of 0.5 to 0, where the decimal round algorithm would round that value to 1.

Round toward positive infinity indicates directed rounding upward is to occur. For this mode, y is rounded to z.

Round toward negative infinity indicates directed rounding downward is to occur. For this mode, y is rounded to x.

Round toward zero indicates truncation is to occur. For this mode, y is rounded to the smaller (in magnitude) of x or z.

Arithmetic operations upon infinity are exact. Negative infinity is less than every finite value, which is less than positive infinity.

The computational attributes are set with a default value upon process initiation. The default attributes are as follows:

- The *floating-point inexact result* exception is masked. The other floating-point exceptions are unmasked.
- All *floating point occurrence bits* have a zero value.
- *Round to the nearest* rounding mode.

These attributes can be modified by a program executing this instruction. The new attributes are then in effect for the program executing this instruction and for programs invoked subsequent to it unless changed through another execution of this instruction. External exception handlers and invocation exit routines are invoked with the same attributes as were last in effect for the program invocation they are related to. Event handlers do not really relate to another invocation in the process. As such, they are invoked with the attributes that were in effect at the point the process was interrupted to handle the event.

Upon return to the invocation of a program from subsequent program invocations, the computational attributes, other than *floating point exception occurrence* attributes, are restored to those that were in effect when the program gave up control. The *floating point exception occurrence* attributes are left intact reflecting the occurrence of any floating-point exceptions during the execution of subsequent invocations.

Internal exception handlers execute under the invocation of the program containing them. As such, the above discussion of how computational attributes are restored upon returning from an external exception handler does not apply. The execution of an internal exception handler occurs in a manner similar to the execution of an internal subroutine invoked through the Call Internal instruction. If the internal exception handler modifies the attributes, the modification remains in effect for that invocation when the exception handler completes the exception.

Exceptions

Exception		Operands 1	2	3	Other
06	Addressing				
	01 space addressing violation	X	X	X	
	02 boundary alignment	X	X	X	
	03 range	X	X	X	
	06 optimized addressability invalid	X	X	X	
08	Argument/parameter				
	01 parameter reference violation	X	X	X	

Exception		Operands 1	2	3	Other
10	Damage encountered				
	44 partial system object damage				X
1C	Machine-dependent exception				
	03 machine storage limit exceeded				X
20	Machine support				
	02 machine check				X
	03 function check				X
22	Object access				
	01 object not found	X	X	X	
	02 object destroyed	X	X	X	
	03 object suspended	X	X	X	
	08 object compressed				X
24	Pointer specification				
	01 pointer does not exist	X	X	X	
	02 pointer type invalid	X	X	X	
2E	Resource control limit				
	01 user profile storage limit exceeded				X
32	Scalar specification				
	01 scalar type invalid	X	X	X	
	03 scalar value invalid		X	X	
36	Space management				
	01 space extension/truncation				X

Subtract Logical Character (SUBLC)

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand 3	Operand [4-6]
SUBLC 1027		Difference	Minuend	Subtrahend	
SUBLCI 1827	Indicator options	Difference	Minuend	Subtrahend	Indicator targets
SUBLCB 1C27	Branch options	Difference	Minuend	Subtrahend	Branch targets

Operand 1: Character variable scalar (fixed-length).

Operand 2: Character scalar (fixed-length).

Operand 3: Character scalar (fixed-length).

Operand 4-6:

- *Branch Form*-Branch point, instruction pointer, relative instruction number, or absolute instruction number.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Short forms

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand [3-5]
SUBLCS 1127		Difference/Minuend	Subtrahend	
SUBLCIS 1927	Indicator options	Difference/Minuend	Subtrahend	Indicator targets
SUBLCBS 1D27	Branch options	Difference/Minuend	Subtrahend	Branch targets

Operand 1: Character variable scalar (fixed-length).

Operand 2: Character scalar (fixed-length).

Operand 3-5:

- *Branch Form*-Branch point, instruction pointer, relative instruction number, or absolute instruction number.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Description: The unsigned binary value of the subtrahend operand is subtracted from the unsigned binary value of the minuend operand, and the result is placed in the difference operand.

Operands 1, 2, and 3 must be the same length; otherwise, the Create Program instruction signals an *invalid length* (hex 2A0A) exception.

The subtraction operation is performed as though the ones complement of the second operand and a low-order 1-bit were added to the first operand.

The result value is then placed (left-adjusted) into the receiver operand with truncating or padding taking place on the right. The pad value used in this instruction is a byte value of hex 00.

If operands overlap but do not share all of the same bytes, results of operations performed on these operands are not predictable. If overlapped operands share all of the same bytes, the results are predictable when direct addressing is used. If indirect addressing is used (that is, based operands, parameters, strings with variable lengths, and arrays with variable subscripts), the results are not always predictable.

Resultant Conditions: The logical difference of the character scalar operands is:

- Zero with carry out of the high-order bit position
- Not-zero with carry
- Not-zero with no carry.

Exceptions

Exception		Operands 1	2	3	Other
06	Addressing				
	01 Space addressing violation	X	X	X	
	02 Boundary alignment	X	X	X	
	03 Range	X	X	X	
	06 Optimized addressability invalid	X	X	X	
08	Argument/Parameter				
	01 Parameter reference violation	X	X	X	
10	Damage Encountered				
	04 System object damage state	X	X	X	X
	44 Partial system object damage	X	X	X	X
1C	Machine-Dependent Exception				
	03 Machine storage limit exceeded				X
20	Machine Support				
	02 Machine check				X
	03 Function check				X
22	Object Access				
	01 Object not found	X	X	X	
	02 Object destroyed	X	X	X	
	03 Object suspended	X	X	X	
	08 object compressed				X
24	Pointer Specification				
	01 Pointer does not exist	X	X	X	
	02 Pointer type invalid	X	X	X	
2C	Program Execution				
	04 Invalid branch target				X
2E	Resource Control Limit				

Exception		Operands 1	2	3	Other
	01 User Profile storage limit exceeded				X
32	Scalar Specification				
	01 Scalar type invalid	X	X	X	
	02 Scalar attributes invalid	X	X	X	
36	Space Management				
	01 Space Extension/Truncation				X

Subtract Numeric (SUBN)

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand 3	Operand [4-7]
SUBN 1047		Difference	Minuend	Subtrahend	
SUBNR 1247		Difference	Minuend	Subtrahend	
SUBNB 1C47	Branch options	Difference	Minuend	Subtrahend	Branch targets
SUBNBR 1E47	Branch options	Difference	Minuend	Subtrahend	Branch targets
SUBNI 1847	Indicator options	Difference	Minuend	Subtrahend	Indicator targets
SUBNIR 1A47	Indicator options	Difference	Minuend	Subtrahend	Indicator targets

Operand 1: Numeric variable scalar.

Operand 2: Numeric scalar.

Operand 3: Numeric scalar.

Operand 4-7:

- *Branch Form*-Branch point, instruction pointer, relative instruction number, or absolute instruction number.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Short forms

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand [3-6]
SUBNS 1147		Difference/Minuend	Subtrahend	
SUBNSR 1347		Difference/Minuend	Subtrahend	
SUBNBS 1D47	Branch options	Difference/Minuend	Subtrahend	Branch targets
SUBNBSR 1F47	Branch options	Difference/Minuend	Subtrahend	Branch targets
SUBNIS 1947	Indicator options	Difference/Minuend	Subtrahend	Indicator targets
SUBNISR 1B47	Indicator options	Difference/Minuend	Subtrahend	Indicator targets

Operand 1: Numeric variable scalar.

Operand 2: Numeric scalar.

Operand 3-6:

- *Branch Form*-Branch point, instruction pointer, relative instruction number, or absolute instruction number.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Description: The Difference is the result of subtracting the Subtrahend from the Minuend.

Operands can have floating-point, packed or zoned decimal, signed or unsigned binary type.

Source operands are the Minuend and Subtrahend. The receiver operand is the Difference.

If operands have different types, source operands, Minuend and Subtrahend, are converted according to the following rules:

1. If any one of the operands has floating point type, source operands are converted to floating point type.
2. Otherwise, if any one of the operands has zoned or packed decimal type, source operands are converted to packed decimal.
3. Otherwise, the binary operands are converted to a like type. Note: unsigned binary(2) scalars are logically treated as signed binary(4) scalars.

Minuend and Subtrahend are subtracted according to their type. Floating point operands are subtracted using floating point subtraction. Packed decimal operands are subtracted using packed decimal subtraction. Unsigned binary subtraction is used with unsigned binary operands. Signed binary operands are subtracted using two's complement binary subtraction.

Better performance can be obtained if all operands have the same type. Signed and unsigned binary subtractions execute faster than either packed decimal or floating point subtractions.

Decimal operands used in floating-point operations cannot contain more than 15 total digit positions.

For a decimal operation, alignment of the assumed decimal point takes place by padding with 0's on the right end of the source operand with lesser precision.

Floating-point subtraction uses exponent comparison and significand subtraction. Alignment of the binary point is performed, if necessary, by shifting the significand of the value with the smaller exponent to the right. The exponent is increased by one for each binary digit shifted until the two exponents agree.

The operation uses the length and the precision of the source and receiver operands to calculate accurate results. Operations performed in decimal are limited to 31 decimal digits in the intermediate result.

The subtract operation is performed according to the rules of algebra.

The result of the operation is copied into the difference operand. If this operand is not the same type as that used in performing the operation, the resultant value is converted to its type. If necessary, the resultant value is adjusted to the length of the difference operand, aligned at the assumed decimal point of the difference operand, or both before being copied to it. For fixed-point operation, if significant digits are truncated on the left end of the resultant value, a *size* (hex 0C0A) exception is signaled.

For the optional round form of the instruction, specification of a floating-point receiver operand is invalid.

For floating-point operations involving a fixed-point receiver field, if nonzero digits would be truncated off the left end of the resultant value, an *invalid floating-point conversion* (hex 0C0C) exception is signaled.

For a floating-point difference operand, if the exponent of the resultant value is either too large or too small to be represented in the difference field, the *floating-point overflow* (hex 0C06) or the *floating-point underflow* (hex 0C07) exception is signaled.

If a decimal to binary conversion causes a *size* (hex 0C0A) exception to be signaled, the binary value contains the correct truncated result only if the decimal value contains 15 or fewer significant nonfractional digits.

Size exceptions can be inhibited.

Limitations: The following are limits that apply to the functions performed by this instruction.

If operands overlap but do not share all of the same bytes, results of operations performed on these operands are not predictable. If overlapped operands share all of the same bytes, the results are predictable when direct addressing is used. If indirect addressing is used (that is, based operands, parameters, strings with variable lengths, and arrays with variable subscripts), the results are not always predictable.

Resultant Conditions

- Positive-The algebraic value of the numeric scalar difference is positive.
- Negative-The algebraic value of the numeric scalar difference is negative.
- Zero-The algebraic value of the numeric scalar difference is zero.
- Unordered-The value assigned a floating-point difference operand is NaN.

Exceptions

Exception		Operands 1	2	3 [4, 5]	Other
06	Addressing				
	01 Space addressing violation	X	X	X	
	02 Boundary alignment	X	X	X	
	03 Range	X	X	X	
	06 Optimized addressability invalid	X	X	X	
08	Argument/Parameter				
	01 Parameter reference violation	X	X	X	
0C	Computation				
	02 Decimal data		X	X	
	03 Decimal point alignment		X	X	
	06 Floating-point overflow	X			
	07 Floating-point underflow	X			
	09 Floating-point invalid operand		X	X	X
	0A Size	X			
	0C Invalid floating-point conversion	X			
	0D Floating-point inexact result	X			

Exception		Operands 1	2	3 [4, 5]	Other
10	Damage Encountered				
	04 System object damage state	X	X	X	X
	44 Partial system object damage	X	X	X	X
1C	Machine-Dependent Exception				
	03 Machine storage limit exceeded				X
20	Machine Support				
	02 Machine check				X
	03 Function check				X
22	Object Access				
	01 Object not found	X	X	X	
	02 Object destroyed	X	X	X	
	03 Object suspended	X	X	X	
	08 object compressed				X
24	Pointer Specification				
	01 Pointer does not exist	X	X	X	
	02 Pointer type invalid	X	X	X	
2C	Program Execution				
	04 Invalid branch target				X
2E	Resource Control Limit				
	01 User Profile storage limit exceeded				X
36	Space Management				
	01 Space Extension/Truncation				X

Test and Replace Characters (TSTRPLC)

Op Code (Hex)	Operand 1	Operand 2
10A2	Receiver	Replacement

Operand 1: Character variable scalar.

Operand 2: Character scalar.

Description: The character string value represented by operand 1 is tested byte by byte from left to right. Any byte to the left of the leftmost byte which has a value in the range of hex F1 to hex F9 is assigned a byte value equal to the leftmost byte of operand 2. Both operands must be character strings. Only the first character of the replacement string is used in the operation.

The operation stops when the first nonzero zoned decimal digit is found or when all characters of the receiver operand have been replaced.

Exceptions

Exception		Operands 1	2	Other
06	Addressing			
	01 Space addressing violation	X	X	
	02 Boundary alignment	X	X	
	03 Range	X	X	
	06 Optimized addressability invalid	X	X	
08	Argument/Parameter			
	01 Parameter reference violation	X	X	
10	Damage Encountered			
	04 System object damage state	X	X	X
	44 Partial system object damage	X	X	X
1C	Machine-Dependent Exception			
	03 Machine storage limit exceeded			X
20	Machine Support			
	02 Machine check			X
	03 Function check			X
22	Object Access			
	01 Object not found	X	X	
	02 Object destroyed	X	X	
	03 Object suspended	X	X	
	08 object compressed			X
24	Pointer Specification			
	01 Pointer does not exist	X	X	
	02 Pointer type invalid	X	X	
2E	Resource Control Limit			

Exception		Operands 1	2	Other
	01 User Profile storage limit exceeded			X
36	Space Management			
	01 Space Extension/Truncation			X

Test Bit in String (TSTBTS)

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand 3 [4]
TSTBTSB 1C0E	Branch options	Source	Offset	Branch targets
TSTBTSI 180E	Indicator options	Source	Offset	Indicator targets

Operand 1: Character scalar (fixed length) or numeric scalar.

Operand 2: Binary scalar.

Operand 3:

- *Branch Form*-Branch point, instruction pointer, relative instruction number, or absolute instruction number.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Description: Tests the bit of the source operand as indicated by the offset operand to determine if the bit is set or not set.

Based on the test, the resulting condition is used with the extender field to either

- Transfer control conditionally to the instruction indicated in one of the branch target operands (branch form).
- Assign a value to each of the indicator operands (indicator form).

The source operand can be character or numeric. The leftmost bytes of the source operand are used in the operation. The source operand is interpreted as a bit string with the bits numbered left to right from 0 to the total number of bits in the string minus one.

The offset operand indicates which bit of the source operand is to be tested, with a offset of zero indicating the leftmost bit of the leftmost byte of the source operand.

If an offset value less than zero or beyond the length of the string is specified a *scalar value invalid* (hex 3203) exception is signaled.

Resultant Conditions

- Zero: The selected bit of the bit string source operand is zero.
- One: The selected bit of the bit string source operand is one.

Exceptions

Exception		Operands 1	2	3 [4]	Other
06	Addressing				
	01 Spacing addressing violation	X	X	X	
	02 Boundary alignment violation	X	X	X	
	03 Range	X	X	X	
	06 Optimized addressability invalid	X	X	X	
08	Argument/parameter				

Exception		Operands 1	2	3	[4]	Other
	01 Parameter reference violation	X	X	X		
10	Damage encountered					
	04 System object damage state					X
	44 Partial system object damage					X
1C	Machine-dependent exception					
	03 Machine storage limit exceeded					X
20	Machine support					
	02 Machine check					X
	03 Function check					X
22	Object access					
	02 Object destroyed	X	X	X		
	03 Object suspended	X	X	X		
	08 object compressed					X
24	Pointer specification					
	01 Pointer does not exist	X	X	X		
	02 Pointer type invalid	X	X	X		
2E	Resource control limit					
	01 user profile storage limit exceeded					X
32	Scalar specification					
	01 Scalar type invalid	X	X	X		
	03 Scalar value invalid		X			
36	Space management					
	01 space extension/truncation					X

Test Bits Under Mask (TSTBUM)

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand 3 [4, 5]
TSTBUMB 1C2A	Branch options	Source	Mask	Branch targets
TSTBUMI 182A	Indicator options	Source	Mask	Indicator targets

Operand 1: Character variable scalar or numeric variable scalar.

Operand 2: Character scalar or numeric scalar.

Operand 3 [4, 5]

- *Branch Form*-Instruction number, relative instruction number, branch point, or instruction pointer.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Description: Selected bits from the leftmost byte of the source operand are tested to determine their bit values.

Based on the test, the resulting condition is used with the extender field to:

- Transfer control conditionally to the instruction indicated in one of the branch target operands (branch form).
- Assign a value to each of the indicator operands (indicator form).

The source and the mask operands can be character or numeric. The leftmost byte of each of the operands is used in the operands. The operands are interpreted as bit strings. The testing is performed bit by bit with only those bits indicated by the mask operand being tested. A 1-bit in the mask operand specifies that the corresponding bit in the source value is to be tested. A 0-bit in the mask operand specifies that the corresponding bit in the source value is to be ignored.

Resultant Conditions: The selected bits of the bit string source operand are all zeros, all ones, or mixed ones and zeros. A mask operand of all zeros causes a zero resultant condition.

Exceptions

Exception		Operands 1	2	3 [4, 5]	Other
06	Addressing				
	01 space addressing violation	X	X	X	
	02 boundary alignment	X	X	X	
	03 range	X	X	X	
	06 optimized addressability invalid	X	X	X	
08	Argument/parameter				
	01 parameter reference violation	X	X	X	
10	Damage encountered				
	04 system object damage state	X	X	X	X
	44 partial system object damage	X	X	X	X
1C	Machine-dependent exception				

Exception		Operands 1	2	3 [4, 5]	Other
	03 machine storage limit exceeded				X
20	Machine support				
	02 machine check				X
	03 function check				X
22	Object access				
	01 object not found	X	X	X	
	02 object destroyed	X	X	X	
	03 object suspended	X	X	X	
	08 object compressed				X
24	Pointer specification				
	01 pointer does not exist	X	X	X	
	02 pointer type invalid	X	X	X	
2C	Program execution				
	04 branch target invalid			X	
2E	Resource control limit				
	01 user profile storage limit exceeded				X
36	Space management				
	01 space extension/truncation				X

Translate (XLATE)

Op Code (Hex)	Operand 1	Operand 2	Operand 3	Operand 4
1094	Receiver	Source	Position	Replacement

Operand 1: Character variable scalar.

Operand 2: Character scalar.

Operand 3: Character scalar or null.

Operand 4: Character scalar.

Description: Selected characters in the string value of the source operand are translated into a different encoding and placed in the receiver operand. The characters selected for translation and the character values they are translated to are indicated by entries in the position and replacement strings. All the operands must be character strings. The source and receiver values must be of the same length. The position and replacement operands can differ in length. If operand 3 is null, a 256-character string is used, ranging in value from hex 00 to hex FF (EBCDIC collating sequence).

The operation begins with all the operands left-adjusted and proceeds character by character, from left to right until the character string value of the receiver operand is completed.

Each character of the source operand value is compared with the individual characters in the position operand. If a character of equal value does not exist in the position string, the source character is placed unchanged in the receiver operand. If a character of equal value is found in the position string, the corresponding character in the same relative location within the replacement string is placed in the receiver operand as the source character translated value. If the replacement string is shorter than the position string and a match of a source to position string character occurs for which there is no corresponding replacement character, the source character is placed unchanged in the receiver operand. If the replacement string is longer than the position string, the rightmost excess characters of the replacement string are not used in the translation operation because they have no corresponding position string characters. If a character in the position string is duplicated, the first occurrence (leftmost) is used.

If operands overlap but do not share all of the same bytes, results of operations performed on these operands are not predictable. If overlapped operands share all of the same bytes, the results are predictable when direct addressing is used. If indirect addressing is used (that is, based operands, parameters, strings with variable lengths, and arrays with variable subscripts), the results are not always predictable.

The receiver, source, position, and replacement operands can be variable length substring compound operands.

Substring operand references that allow for a null substring reference (a length value of zero) may be specified for all of the operands on this instruction. The effect of specifying a null substring reference for either the position or replacement operands is that the source operand is copied to the receiver with no change in value. The effect of specifying a null substring reference for both the receiver and the source operands (they must have the same length) is that no result is set.

Exceptions

Exception		Operands 1	2	3	4	Other
06	Addressing					

Exception		Operands 1	2	3	4	Other
	01 Space addressing violation	X	X	X	X	
	02 Boundary alignment	X	X	X	X	
	03 Range	X	X	X	X	
	06 Optimized addressability invalid	X	X	X	X	
08	Argument/Parameter					
	01 Parameter reference violation	X	X	X	X	
0C	Computation					
	08 Length conformance	X	X			
10	Damage Encountered					
	04 System object damage state	X	X	X	X	X
	44 Partial system object damage	X	X	X	X	X
1C	Machine-Dependent Exception					
	03 Machine storage limit exceeded					X
20	Machine Support					
	02 Machine check					X
	03 Function check					X
22	Object Access					
	01 Object not found	X	X	X	X	
	02 Object destroyed	X	X	X	X	
	03 Object suspended	X	X	X	X	
	08 object compressed					X
24	Pointer Specification					
	01 Pointer does not exist	X	X	X	X	
	02 Pointer type invalid	X	X	X	X	
2E	Resource Control Limit					
	01 User Profile storage limit exceeded					X
36	Space Management					
	01 Space Extension/Truncation					X

Translate with Table (XLATEWT)

Op Code (Hex)	Operand 1	Operand 2	Operand 3
109F	Receiver	Source	Table

Operand 1: Character variable scalar.

Operand 2: Character scalar.

Operand 3: Character scalar.

Description: The source characters are translated under control of the translate table and placed in the receiver. The operation begins with the leftmost character of operand 2 and proceeds character-by-character, left-to-right.

Characters are translated as follows:

- The source character is used as an offset and added to the location of operand 3.
- The character contained in the offset location is the translated character. This character is copied to the receiver in the same relative position as the original character in the source string.

If operand 3 is less than 256 bytes long, the character in the source may specify an offset beyond the end of operand 3. If operand 2 is longer than operand 1, then only the leftmost portion of operand 2, equal to the length of operand 1, is translated. If operand 2 is shorter than operand 1, then only the leftmost portion of operand 1, equal to the length of operand 2, is changed. The remaining portion of operand 1 is unchanged.

If operand 1 overlaps with operand 2 and/or 3, the overlapped operands are updated for every character translated. The operation proceeds from left to right, one character at a time. The following example shows the results of an overlapped operands translate operation. Operands 1, 2, and 3 have the same coincident character string with a value of hex 050403020103.

Hex 050403020103-Initial value

Hex 030403020103-After the 1st character is translated

Hex 030103020103-After the 2nd character is translated

Hex 030102020103-After the 3rd character is translated

Hex 030102020103-After the 4th character is translated

Hex 030102020103-After the 5th character is translated

Hex 030102020102-After the 6th character, the final result

Note that the instruction does not use the length specified for the table operand to constrain access of the bytes addressed by the table operand.

If operand 3 is less than 256 characters long, and a source character specifies an offset beyond the end of operand 3, the result characters are obtained from byte locations in the space following operand 3. If that portion of the space is not currently allocated, a space addressing exception is signaled. If operand 3 is a constant with a length less than 256, source characters specifying offsets greater than or equal to the length of the constant are translated into unpredictable characters.

All of the operands support variable length substring compound scalars.

Substring operand references that allow for a null substring reference (a length value of zero) may be specified for all of the operands on this instruction. Specifying a null substring reference for the table operand does not affect the operation of the instruction. In this case, the bytes addressed by the table operand are still accessed as described above. This is due to the definition of the function of this instruction which does not use the length specified for the table operand to constrain access of the bytes addressed by the table operand. The effect of specifying a null substring reference for either or both of the receiver and the source operands is that no result is set.

Exceptions

Exception		Operands 1	2	3	Other
06	Addressing				
	01 Space addressing violation	X	X	X	
	02 Boundary alignment	X	X	X	
	03 Range	X	X	X	
	06 Optimized addressability invalid	X	X	X	
08	Argument/Parameter				
	01 Parameter reference violation	X	X	X	
10	Damage Encountered				
	44 Partial system object damage				X
1C	Machine-Dependent Exception				
	03 Machine storage limit exceeded				X
20	Machine Support				
	02 Machine check				X
	03 Function check				X
22	Object Access				
	02 Object destroyed	X	X	X	
	03 Object suspended	X	X	X	
	08 object compressed				X
24	Pointer Specification				
	01 Pointer does not exist	X	X	X	
	02 Pointer type invalid	X	X	X	
2E	Resource Control Limit				
	01 User Profile storage limit exceeded				X
32	Scalar Specification				
	01 Scalar type invalid	X	X	X	
36	Space Management				
	01 Space Extension/Truncation				X

Translate with Table and DBCS Skip (XLATWTDS)

Op Code (Hex)	Operand 1	Operand 2	Operand 3
1077	Target	Length	Table

Operand 1: Character variable scalar (fixed length).

Operand 2: Binary(4) scalar.

Operand 3: Character scalar (fixed length).

ILE access

```
XLATWTDS (
  var target : aggregate;
  var length : unsigned binary;
  var table  : aggregate
)
```

Description: The simple (single byte) characters in the Target are translated under control of the translate table, for the length defined by Operand 2. The extended (double byte) character portions of the target are bypassed and not translated. The operation begins with the leftmost character of operand 1 and proceeds character-by-character, left-to-right, skipping over any Double byte character (DBCS) data portions.

The target, Operand 1, should have double byte character data surrounded by a shift out control character (SO = hex 0E) and a shift in control character (SI= hex 0F). Once a SO character is encountered, the translating of single byte characters halts. The operation will then proceed double byte character-by-double byte character until a SI character is encountered. This shift in character is then used to restart the translating of single byte characters.

The length operand, Operand 2, is the number of bytes and must contain a value between 1 and 32767. For length values outside this range a *scalar value invalid* (hex 3203) exception is signaled.

Single byte characters are translated as follows:

- The target character is used as an offset and added to the location of Operand 3.
- The character contained at the offset location of Operand 3 is the translated character. This character replaces the original character in the target.

The following example shows the step-by-step results of this translate operation. The translate table for this example has the following hex value: C3D406C5D504C1C2C4C5C6C7C8C9C1C6

Hex 05040ED2D2E1E10F03 - Initial target value

Hex 04040ED2D2E1E10F03 - After the 1st character is translated

Hex 04D50ED2D2E1E10F03 - After the 2nd character is translated

Hex 04D50ED2D2E1E10F03 - SO character encountered, skip the DBCS portion

Hex 04D50ED2D2E1E10F03 - Resume translating after SI control character

Hex 04D50ED2D2E1E10FC5 - Translate 9th character

Hex 04D50ED2D2E1E10FC5 - Final target value

The translate table, Operand 3, is assumed to be 256 bytes long. If the table is less than 256 characters long, and a target character specifies an offset beyond the end of the table, the resultant characters are obtained from byte locations in the space following translate table. If that portion of the space is not currently allocated, a *space addressing* (hex 0601) exception is signaled.

This operation only translates the target string and does not validate the Double byte portions of the target. For example, if a DBCS portion of the target string is preceded by the Shift Out control character, but missing the closing Shift In character, then an *invalid extended character data* (hex 0C12) exception will NOT be signaled. However, the Copy Extended Characters Left-Adjusted With Pad (CPYECLAP) instruction can be used to validate extended character data, if necessary.

Exceptions

Exception		Operands 1	2	3	Other
06	Addressing				
	01 Spacing addressing violation	X	X	X	
	02 Boundary alignment	X	X	X	
	03 Range	X	X	X	
	06 Optimized addressability invalid	X	X	X	
08	Argument/parameter				
	01 Parameter reference violation	X	X	X	
10	Damage encountered				
	44 Partial system object damage	X	X	X	X
1C	Machine-dependent exception				
	03 Machine storage limit exceeded				X
20	Machine support				
	02 Machine check				X
	03 Function check				X
22	Object access				
	02 Object destroyed	X	X	X	
	03 Object suspended	X	X	X	
	08 object compressed				X
24	Pointer specification				
	01 Pointer does not exist	X	X	X	
	02 Pointer type invalid	X	X	X	
2E	Resource control limit				
	01 user profile storage limit exceeded				X
32	Scalar specification				
	01 Scalar type invalid	X	X	X	
	03 Scalar value invalid		X		
36	Space management				

Exception	Operands 1	2	3	Other
01 space extension/truncation				X

Trim Length (TRIML)

Op Code (Hex)	Operand 1	Operand 2	Operand 3
10A7	Receiver length	Source string	Trim character

Operand 1: Numeric variable scalar.

Operand 2: Character scalar.

Operand 3: Character(1) scalar.

Description: The operation determines the resultant length of operand 2 after the character specified by operand 3 has been trimmed from the end of operand 2. The resulting length is stored in operand 1. Operand 2 is trimmed from the end as follows: if the rightmost (last) character of operand 2 is equal to the character specified by operand 3, the length of the trimmed operand 2 string is reduced by 1. This operation continues until the rightmost character is no longer equal to operand 3 or the trimmed length is zero. If operand 3 is longer than one character, only the first (leftmost) character is used as the trim character.

Operands 2 and 3 are not changed by this instruction.

Exceptions

Exception		Operands 1	2	3	Other
06	Addressing				
	01 Space addressing violation	X	X	X	
	02 Boundary alignment	X	X	X	
	03 Range	X	X	X	
	06 Optimized addressability invalid	X	X	X	
08	Argument/Parameter				
	01 Parameter reference violation	X	X	X	
0C	Computation				
	0A Size	X			
10	Damage Encountered				
	44 Partial system object damage				X
1C	Machine-Dependent Exception				
	03 Machine storage limit exceeded				X
20	Machine Support				
	02 Machine check				X
	03 Function check				X
22	Object Access				
	01 Object not found	X	X	X	
	02 Object destroyed	X	X	X	
	03 Object suspended	X	X	X	
	08 object compressed				X

Exception		Operands 1	2	3	Other
24	Pointer Specification				
	01 Pointer does not exist	X	X	X	
	02 Pointer type invalid	X	X	X	
2E	Resource Control Limit				
	01 User Profile storage limit exceeded				X
32	Scalar Specification				
	01 Scalar type invalid	X	X	X	
36	Space Management				
	01 Space Extension/Truncation				X

Verify (VERIFY)

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand 3	Operand [4-5]
VERIFY 10D7		Receiver	Source	Class	
VERIFYB 1CD7	Branch options	Receiver	Source	Class	Branch targets
VERIFYI 18D7	Indicator options	Receiver	Source	Class	Indicator targets

Operand 1: Binary variable scalar or binary array.

Operand 2: Character scalar.

Operand 3: Character scalar.

Operand 4-5:

- *Branch Form*-Branch point, instruction pointer, relative instruction number, or absolute instruction number.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Description: Each character of the source operand character string value is checked to verify that it is among the valid characters indicated in the class operand.

The operation begins at the left end of the source string and continues character by character, from left to right. Each character of the source value is compared with the characters of the class operand. If a match for the source character exists in the class string, the next source character is verified. If a match for the source character does not exist in the class string, the binary value for the relative location of the character within the source string is placed in the receiver operand.

If the receiver operand is a scalar, only the first occurrence of an invalid character is noted. If the receiver operand is an array, as many occurrences as there are elements in the array are noted.

The operation continues until no more occurrences of invalid characters can be noted or until the end of the source string is encountered. When the second condition occurs, the current receiver value is set to 0. If the receiver operand is an array, all its remaining entries are set to 0's.

The source and class operands can be variable length substring compound operands.

Substring operand references that allow for a null substring reference (a length value of zero) may be specified for operands 2 and 3. The effect of specifying a null substring reference for the class operand when a nonnull string reference is specified for the source is that all of the characters of the source are considered invalid. In this case, the receiver is accordingly set with the offset(s) to the bytes of the source, and the instruction's resultant condition is positive. The effect of specifying a null substring reference for the source operand (no characters to verify) is that the receiver is set to zero and the instruction's resultant condition is zero regardless of what is specified for the class operand.

Resultant Conditions: The numeric value(s) of the receiver is either 0 or positive. When the receiver operand is an array, the resultant condition is 0 if all elements are 0.

Exceptions

Exception		Operands 1	2	3	Other
06	Addressing				
	01 space addressing violation	X	X	X	
	02 boundary alignment	X	X	X	
	03 range	X	X	X	
	06 optimized addressability invalid	X	X	X	
08	Argument/parameter				
	01 parameter reference violation	X	X	X	
10	Damage encountered				
	04 system object damage	X	X	X	X
	44 partial system object damage	X	X	X	X
1C	Machine-dependent exception				
	03 machine storage limit exceeded				X
20	Machine support				
	02 machine check				X
	03 function check				X
22	Object access				
	01 object not found	X	X	X	
	02 object destroyed	X	X	X	
	03 object suspended	X	X	X	
	08 object compressed				X
24	Pointer specification				
	01 pointer does not exist	X	X	X	
	02 pointer type invalid	X	X	X	
2C	Program execution				
	04 branch target invalid				X
2E	Resource control limit				
	01 user profile storage limit exceeded				X
36	Space management				
	01 space extension/truncation				X

Chapter 3. Date/Time/Timestamp Instructions

This chapter describes all instructions used for date, time, and timestamp use. These instructions are in alphabetic order. For an alphabetic summary of all the instructions, see Appendix A, "Instruction Summary"

External Data Formats . . . 3-3
Date, Time, and Timestamp Concepts . . . 3-3
Compute Date Duration (CDD) . . . 3-13
Compute Time Duration (CTD) . . . 3-16
Compute Timestamp Duration (CTSD) . . . 3-19
Convert Date (CVTD) . . . 3-22
Convert Time (CVTT) . . . 3-25
Convert Timestamp (CVTTS) . . . 3-28
Decrement Date (DECD) . . . 3-31
Decrement Time (DECT) . . . 3-35
Decrement Timestamp (DECTS) . . . 3-38
Increment Date (INCD) . . . 3-42
Increment Time (INCT) . . . 3-46
Increment Timestamp (INCTS) . . . 3-49

© Copyright IBM Corp. 1991, 1993

External Data Formats

For support of date, time, and timestamp, the external data format that is used is a character string. Associated with this character string will be a **data definitional attribute list** (DDAT) that describes all the attributes of the string.

The attributes that are associated with the character string are as follows:

- Calendar table.
- Era table.
- Format code.
- Date separator type.
- Time separator type.
- Time zone.
- Month definition.
- Year definition.
- Century.

Date, Time, and Timestamp Concepts

Data Definitions

DATE. ***Date data type.*** The internal format is a 4 byte binary value. The DDAT number must reference a DDAT with an internal DATE format code or be set to zero which implies internal format. The internal format is fixed length, trailing blanks are NOT allowed.

TIME. ***Time data type.*** The internal format is six numbers packed into a three byte value. There is no sign nibble. The DDAT number must reference a DDAT with an internal TIME format code or be set to zero which implies internal format. The internal format is fixed length, trailing blanks are NOT allowed.

TIMESTAMP. ***Timestamp data type.*** The internal format of a Timestamp is a ten byte composite value. The composite is two numbers one for date and time respectfully. The date and time numbers have the same encoding as the date and time data types, with an exception to the time number. The time number has an additional 3 bytes for a six packed digit microsecond value. The DDAT number must reference a DDAT with an internal Timestamp format code or be set to zero which implies internal format. The internal format is fixed length, trailing blanks are NOT allowed.

PseudoDATE. ***Character data type with a DDAT referenced.*** The DDAT number must reference a DDAT with a valid date format code. The character string format must match one of the supported formats, trailing blanks are allowed. The length is the number of bytes of the character string which can include trailing blanks.

PseudoTIME. ***Character data type with a DDAT referenced.*** The DDAT number must reference a DDAT with a valid time format code. The character string format must match one of the supported formats, trailing blanks are allowed. The length is the number of bytes of the character string which can include trailing blanks.

PseudoTIMESTAMP. ***Character data type with a DDAT referenced.*** The DDAT number must reference a DDAT with a valid timestamp format code. The character string format must match one of the sup-

ported formats, trailing blanks are allowed. The length is the number of bytes of the character string which can include trailing blanks.

DATE DURATION. ***Packed decimal data type with a DDAT referenced.*** The length must be 8,0. The DDAT number is set to reference a DDAT with a valid date duration format code.

TIME DURATION. ***Packed decimal data type with a DDAT referenced.*** The length must be 6,0. The DDAT number must reference a DDAT with a valid time duration format code.

TIMESTAMP DURATION. ***Packed decimal data type with a DDAT referenced.*** The length must be 20,6. The DDAT number must reference a DDAT with a valid timestamp duration format code.

DATE LABEL DURATIONS. ***Packed decimal data type with a DDAT referenced.*** The length must be 15,0. The DDAT number must reference a DDAT with a valid date label duration format code.

TIME LABEL DURATIONS. ***Packed decimal data type with a DDAT referenced.*** The length must be 15,0. The DDAT number must reference a DDAT with a valid time label duration format code.

TIMESTAMP LABEL DURATIONS. ***Packed decimal data type with a DDAT referenced.*** The length must be 15,0. The DDAT number must reference a DDAT with a valid timestamp label duration format code.

Data Conversion

The following table describes the possible conversions for date/time data.

Source Operand	Result Operand
DATE	DATE or PseudoDATE
TIME	TIME or PseudoTIME
TIMESTAMP	TIMESTAMP or PseudoTIMESTAMP
PseudoDATE	DATE or PseudoDATE
PseudoTIME	TIME or PseudoTIME
PseudoTIMESTAMP	TIMESTAMP or PseudoTIMESTAMP

Arithmetic

For all Date/Time arithmetic, Increment, Decrement, and Compute, there must be at least one non-zero DDAT number specified between operand 1, operand 2, and the result.

DDATs are equivalent when both DDAT numbers are 0 or both DDAT numbers are non-zero and the DDATs they reference are byte for byte identical.

End of month adjustment can only be specified in conjunction with Date and Timestamp arithmetic that involves the incrementing and decrementing of Dates or Timestamps.

When end of month adjustment is specified, there must be at least one non-internal format specified for operand 1, operand 2, or the result. When end of month adjustment is not specified all durations must reference a DDAT that specifies a year and month definition other than zero.

Data Definitional Attribute Template

The following describes the Data Definitional Attribute Template (DDAT).

- Data definitional attribute template (optional) Char(112-n)
  (repeated for each operand/field that requires a template, one template can be used by multiple operands/fields that have the same attributes)
  - DDAT length (ignored) UBin(2)
  - Format code UBin(2)
  - Separator definition Char(2)
    - Date separator type Char(1)
    - Time separator type Char(1)
  - Time zone definition Char(4)
    - Hour zone UBin(2)
    - Minute zone UBin(2)
  - Duration definitions Char(4)
    - Month definition UBin(2)
    - Year definition UBin(2)
  - Century definition Char(8)
    - Current century UBin(4)
    - Century division UBin(4)
  - Calendar table offset UBin(4)
  - Reserved (binary 0) Char(6)
  - Era table template Char(50-n)
  - Calendar table template Char(18-n)

The following table describes the operand type and DDAT field associations.

Operand Type	Format Code	Date Separator	Time Separator	Time Zone	Month Definition	Year Definition	Century	Era Table	Calendar Table
Date	REQ	REQ	INV	INV	INV	INV	REQ	REQ	REQ
Time	REQ	INV	REQ	REQ	INV	INV	INV	INV	INV
Timestamp	REQ	REQ	REQ	REQ	INV	INV	REQ	REQ	REQ
Pseudo Date	REQ	REQ	INV	INV	INV	INV	REQ	REQ	REQ
Pseudo Time	REQ	INV	REQ	REQ	INV	INV	INV	INV	INV
Pseudo Timestamp	REQ	REQ	REQ	REQ	INV	INV	REQ	REQ	REQ
Date Duration	REQ	INV	INV	INV	EMA	EMA	INV	INV	INV
Time Duration	REQ	INV	INV	INV	INV	INV	INV	INV	INV
Timestamp Duration	REQ	INV	INV	INV	EMA	EMA	INV	INV	INV
Year Label Duration	REQ	INV	INV	INV	EMA	EMA	INV	INV	INV
Month Label Duration	REQ	INV	INV	INV	EMA	EMA	INV	INV	INV
Day Label Duration	REQ	INV	INV	INV	EMA	EMA	INV	INV	INV
Hour Label Duration	REQ	INV	INV	INV	INV	INV	INV	INV	INV
Minute Label Duration	REQ	INV	INV	INV	INV	INV	INV	INV	INV
Second Label Duration	REQ	INV	INV	INV	INV	INV	INV	INV	INV
Microsecond Label Duration	REQ	INV	INV	INV	INV	INV	INV	INV	INV

Notes:

REQ The DDAT field is required and the value must be non-zero.

INV The DDAT field is invalid and the value must be zero.

EMA The DDAT field is required to have a zero value for end of month adjustment arithmetic. Otherwise the field value is required to be non-zero.

The **DDAT length** the length of the DDAT in bytes. This field is ignored by the instruction.

The following describes the **format code** The formats are use to define the representation or interpretation of data across the MI. They also imply the length of the data.

Format Type	Format Code	Minimum Input Length	Maximum Input Length	Minimum Output Length	Maximum Output Length
USA date	Hex 0001	8	n	10	n
USA time	Hex 0002	7	n	8	n
ISO date	Hex 0003	8	n	10	n
ISO time	Hex 0004	4	n	8	n
EUR date	Hex 0005	8	n	10	n
EUR time	Hex 0006	4	n	8	n
JIS date	Hex 0007	8	n	10	n
JIS time	Hex 0008	4	n	8	n
SAA timestamp	Hex 0009	16	n	26	n
System internal date	Hex 000A	4	4/n #	4	4/n #
System internal time	Hex 000B	3	3/n #	3	3/n #
System internal timestamp	Hex 000C	10	10/n #	10	10/n #
Labeled duration YEAR	Hex 000D	15,0	15,0	15,0	15,0
Labeled duration MONTH	Hex 000E	15,0	15,0	15,0	15,0
Labeled duration DAY	Hex 000F	15,0	15,0	15,0	15,0
Labeled duration HOUR	Hex 0010	15,0	15,0	15,0	15,0
Labeled duration MINUTE	Hex 0011	15,0	15,0	15,0	15,0
Labeled duration SECOND	Hex 0012	15,0	15,0	15,0	15,0
Labeled duration MICROSECOND	Hex 0013	15,0	15,0	15,0	15,0
Date duration	Hex 0014	8,0	8,0	8,0	8,0
Time duration	Hex 0015	6,0	6,0	6,0	6,0
Timestamp duration	Hex 0016	20,6	20,6	20,6	20,6
*MMDDYY	Hex 0017	6	n	6	n
*DDMMYY	Hex 0018	6	n	6	n
*YYMMDD	Hex 0019	6	n	6	n
*YYDDD	Hex 001A	5	n	5	n
*HHMMSS	Hex 001B	6	n	6	n
IMPI clock	Hex 001C	8	8	8	8
*YYYYDDD	Hex 001D	7	n	7	n
*YYYYMMDDhhmmss	Hex 001E	14	n	14	n
*Unknown date	Hex 001F	4	n	F	n
*Unknown time	Hex 0020	3	n	F	n
*Unknown timestamp	Hex 0021	10	n	F	n

Format Type	Format Code	Preferred Format	Search List	Character data type	Packed decimal data type
USA date	Hex 0001	Y	Y	Y	N
USA time	Hex 0002	Y	Y	Y	N
ISO date	Hex 0003	Y	Y	Y	N
ISO time	Hex 0004	Y	Y	Y	N
EUR date	Hex 0005	Y	Y	Y	N
EUR time	Hex 0006	Y	Y	Y	N
JIS date	Hex 0007	Y	Y	Y	N
JIS time	Hex 0008	Y	Y	Y	N
SAA timestamp	Hex 0009	Y	Y	Y	N
System internal date	Hex 000A	Y	N	Y	N
System internal time	Hex 000B	Y	N	Y	N
System internal timestamp	Hex 000C	Y	N	Y	N
Labeled duration YEAR	Hex 000D			N	Y
Labeled duration MONTH	Hex 000E			N	Y
Labeled duration DAY	Hex 000F			N	Y
Labeled duration HOUR	Hex 0010			N	Y
Labeled duration MINUTE	Hex 0011			N	Y
Labeled duration SECOND	Hex 0012			N	Y
Labeled duration MICROSECOND	Hex 0013			N	Y
Date duration	Hex 0014			N	Y
Time duration	Hex 0015			N	Y
Timestamp duration	Hex 0016			N	Y
*MMDDYY	Hex 0017	Y	N	Y	N
*DDMMYY	Hex 0018	Y	N	Y	N
*YYMMDD	Hex 0019	Y	N	Y	N
*YYDDD	Hex 001A	Y	N	Y	N
*HHMMSS	Hex 001B	Y	N	Y	N
IMPI clock	Hex 001C	Y	N	Y	N
*YYYYDDD	Hex 001D	Y	Y	Y	N
*YYYYMMDDhhmmss	Hex 001E	Y	Y	Y	N
*Unknown date	Hex 001F			Y	N
*Unknown time	Hex 0020			Y	N
*Unknown timestamp	Hex 0021			Y	N

Notes:

'n' - Any number which is greater than minimum length.

'Y' - Allowed.

'N' - Not allowed.

' ' - Not applicable.

'F' - Reference the formats for this value.

'#' - Date, time, and timestamp data types the length is fixed 4, 3, and 10 respectfully. For the character data type the length is variable.

For further information on the SAA formats reference the SAA CPI Database Reference. The following are the descriptions of the formats, D represents days, M represents months, Y represents years, h represents hours, m represents minutes, s represents seconds and u represents microseconds.

- USA
  - Date - MM/DD/YYYY, character string.
  - Time - hh:mm AM or PM, character string. 00:00 AM or 12:00 AM is midnight. 12:00 PM is noon. Between 'hh:mm' and 'AM' or 'PM' only one one blank is allowed.
- ISO
  - Date - YYYY-MM-DD, character string.
  - Time - hh.mm.ss, character string.
- EUR
  - Date - DD.MM.YYYY, character string.
  - Time - hh.mm.ss, character string.
- JIS
  - Date - YYYY-MM-DD, character string.
  - Time - hh:mm:ss, charracter string.
- SAA timestamp
  - Timestamp - YYYY-MM-DD-hh.mm.ss.uuuuuu, character string. Leading zeros maybe omitted from the month, day, and hour part of the timestamp, and microseconds may be truncated or entirely omitted, (.10 = .100000).
- System internal format
  - Date - 4 byte binary Scaliger number
  - Time - 3 byte value with each byte containing two packed decimal digits having the encoding of hhmmss.
  - Timestamp - 10 byte composite number, the first part being a 4 byte internal date and the second part being a 3 byte internal time, plus microseconds, a 3 byte value with each byte containing two packed decimal digits having the encoding of uuuuuu.
- Labeled durations
  - They are a 15,0 packed decimal data value. The keywords; YEARS, MONTHS, DAYS, HOURS, MINUTES, SECONDS, and MICROSECONDS, or the singular form of the keywords; YEAR, MONTH, DAY, HOUR, MINUTE, SECOND, and MICROSECOND, are not part of the data.
- Date duration
  - Packed decimal (8,0) number having the encoding of YYYYMMDD, where YYYY has the range 0 - 9999, MM has the range 0 - 99, and DD has the range 0 - 99.
- Time duration
  - Packed decimal (6,0) number having the encoding of hhmmss, where hh has the range 0 - 99, mm has the range 0 - 99, and ss has the range 0 - 99.
- Timestamp duration

  - Packed decimal (20,6) number having the encoding of YYYYMMDDhhmmssuuuuuu, where in addition to date and time, uuuuuu has the range 0 - 999999.
- *MMDDYY
  - Date - MMDDYY, character string
- *DDMMYY
  - Date - DDMMYY, character string
- *YYMMDD
  - Date - YYMMDD, character string
- *YYDDD
  - Date - YYDDD, character string
- *HHMMSS
  - Time - hhmmss, character string
- IMPI clock
  - It is an 8 byte bit clock, which is used as a timestamp.
- *YYYYDDD
  - Date - YYYYDDD, seven character string. Used on the DATE scalar function.
- *YYYYMMDDhhmmss
  - Timestamp - YYYYMMDDhhmmss fourteen character string.
- *Unknown
  - Used to reference unique SAA formats. This format is used only when the format is NOT known. This format tries to find a matching format. The formats that will be scanned for are; USA, ISO, EUR, JIS, dates and times, *YYYYDDD (date), SAA timestamp, and *YYYYMMDDhhmmss timestamp. When a valid match is not found either a data, format or value, exception will be signalled.

The following describes the **separator definitions** for date and time. The **separator type** is a one character field that contains the separator value, for example, :, ., /, or -. When the format has an implied separator, the implied separator is used. For example, an ISO date, the implied separator is '-'. When the format does not have an implied separator, the implied separator is null. For example, the format *YYMMDD has an implied separator of null. A null separator means the units of time, year, month, day, etc..., are concatenated together. A implied separator is specified by using the value of hex '00'. A null separator is specified by using the value of hex 'FF'. A null separator is invalid for the SAA formats, USA, ISO, etc... Any other separator value overrides the implied value for the format code.

The following describes the **time zone definition**. The **hour zone** value is 0 to 24. GMT zone is 0. The zone to the east of zone 0 is zone 1. The zone to the east of zone 1 is zone 2, and so on. The value 24 specifies that the time is to be stored as local time and that the concept of time zones should be ignored. The value of 24 is required for storage purposes. For example, when inserting a time field into a database data space the hour zone value must be 24. A value other than 24 can be used for retrieval purposes.

The **minute zone** value is 0 to 60. GMT zone is 0. The zone to the west of zone 0 is zone 1. The zone to the west of zone 1 is zone 2, and so on. The value 60 specifies that the time is to be stored as local time and that the concept of time zones should be ignored. The value of 60 is required for storage purposes. For example, when inserting a time field into a database data space the minute zone value must be 60. A value other than 60 can be used for retrieval purposes.

The **month definition** is an integer number of days. For example 30, would specify that each month use in a duration would have a constant value of 30 days. The a value of zero specifies that the calendar definition of the month should be used.

The **year definition** is an integer number of days. For example 365, would specify that each year use in a duration would have a constant value of 365 days. The a value of zero specifies that the calendar definition of the year should be used.

When end of month adjustment arithmetic is being performed, the *month definition* and *year definition* must have zero values. When no end of month adjustment arithmetic is being performed, the *month definition* and *year definition* must have non-zero values. Otherwise a *template value invalid* (hex 3801) exception will be signaled.

The **century field** is used to define the century when a two digit year is specified for a date. The definition is two numbers, one for current century and the other is for the century division year. The current century is a century number. The century division is a year number. The valid values for the current century are 0 - 99. The valid values for the century division are 0 - 99. The century division is included in the current century. For example, a current century of 19 and a century division of 50 would result in two digit years 00 - 49, having the values 2000 - 2049 and 50 - 99 having values 1950 - 1999.

The **calendar table offset** is the number of bytes from the start of the DDAT to the start of the calendar table (described below).

Era Table

The **era table** immediately follows the fixed portion of the DDAT.

The following describes the era table data.

- Number of table elements — UBin(2)
- Era element (repeatable) — Char(48)
  - Origin date — UBin(4)
  - Era name — Char(32)
  - Reserved (binary 0) — Char(12)

The *era table* is a list of elements that state what era should be used across the time line. The start of usage of a particular era is specified by the origin date. The end of usage of a particular era is terminated by the next table element. The last table element era is used until the end of the time line. The origin date is specified in the internal format. The era name is a character field. The maximum number of table entries allowed is 256.

The SAA *era table* has one entry. The SAA origin date is January 1, 0001, Gregorian for the start of the time line. The internal format would be 1721424. The SAA name is AD, anno Domini.

The SAA *era table* can have only one element and that element must have an effective date that falls in the time line specified in the SAA *calendar table*.

Calendar Table

The following describes the **calendar table** data.

• Number of table elements	UBin(2)
• Calendar change element (repeatable)	Char(16)
— Effective date	UBin(4)
— Calendar type	UBin(2)
— Reserved (binary 0)	Char(10)

The *calendar table* is a list of elements that state what calendar type/algorithm should be used across the time line. The start of usage of a particular calendar is specified by the effective date. The effective date is specified using the internal format. This first table element represent the beginning of the time line. The end of usage of a particular calendar is terminated by the next table element. The last table element calendar must be null this indicates the end of the time line. The maximum number of table entries allowed is 256.

The SAA *calendar table* has 2 entries. The first entry has a calendar type of Gregorian. The effective date is January 1, 0001, Gregorian for the start of the time line. The internal format would be 1721424. The second entry has a calendar type of null. The effective date is January 1, 10000, Gregorian for the end of the time line. The internal format would be 5373485.

Multiple calendar table entries are only valid with DDATs specifying the internal date *format code.* The rest of *format codes* can only have two entries in the calendar table. The second entry must have a NULL *calendar type.*

The following describes the encoding of the **calendar types**.

Calendar type	Calendar type value
Null	Hex 0000
Gregorian	Hex 0001
Julian	Hex 0002
Muslim	Hex 0003
Hebrew	Hex 0004

Compute Date Duration (CDD)

Op Code (Hex)	Operand 1	Operand 2	Operand 3	Operand 4
0424	Date duration	Date 1	Date 2	Instruction template

Operand 1: Packed decimal variable scalar.

Operand 2: Character scalar.

Operand 3: Character scalar.

Operand 4: Space pointer.

ILE access

```
CDD (
  var date_duration        : packed decimal;
  var date1                : aggregate;
  var date2                : aggregate;
      instruction_template : space pointer
)
```

Description: The date specified by operand 3 is subtracted from the date specified by operand 2 and the value of the results a duration is place in operand 1. Operand 4 defines the data definitional attributes for operands 1 through 3.

A negative value will be returned when the first operand is less than the second operand.

The following describes the instruction template.

- Instruction template — Char(*)
  - Instruction template size — UBin(4)
  - Operand 1 data definitional attribute template number — UBin(2)
  - Operand 2 data definitional attribute template number — UBin(2)
  - Operand 3 data definitional attribute template number — UBin(2)
  - Reserved (binary 0) — Char(2)
  - Operand 2 length — UBin(2)
  - Operand 3 length — UBin(2)
  - Reserved (binary 0) — Char(26)
  - Data definitional attribute template list — Char(*)
    - Size of the DDAT list — UBin(4)
    - Number of DDATs — UBin(2)
    - Reserved (binary 0) — Char(10)
    - DDAT offset (repeated) — UBin(4)
    - Data definitional attribute template (repeated) — Char(*)

A **data definitional attribute template (DDAT) number** is a number that corresponds to the relative position of a template in the **data definitional attribute template list**. For example the number 1 references the first template. The valid values for this field are 1, 2, and 3.

The DDAT for operand 1 must be valid for a date duration. The DDATs for operands 2 and 3 must be valid for a date and identical. Otherwise, a *template value invalid* (hex 3801) exception will be signaled.

Operand 2 length and **operand 3 length** are specified in number of bytes.

The **size of the DDAT list** is specified in bytes.

The **number of DDATs** is the count of DDATs specified for this instruction template. The maximum number of DDATs that can be specified is 3.

The **DDAT offset** is the number of bytes from the start of the DDAT list to the start of the specific DDAT. There should be as many *DDAT offsets* as there are DDATs specified.

A data definitional attribute template defines the presentation of the data. Each template describes the definitional attributes of the operands. The length of the character operands will be defined by the template. For a further description of the data definitional attribute template, see "Data Definitional Attribute Template" on page 3-5

Authorization Required

- None.

Lock Enforcement

- None.

Exceptions

Exception		Operands 1	2	3	4	Other
06	Addressing					
	01 Space addressing violation	X	X	X	X	
	02 Boundary alignment	X	X	X	X	
	03 Range	X	X	X	X	
	06 Optimized addressability invalid	X	X	X	X	
08	Argument/Parameter					
	01 Parameter reference violation	X	X	X	X	
0C	Computation					
	15 Data boundary overflow		X	X		
	16 Data format		X	X		
	17 Data value		X	X		
	18 Date boundary underflow		X	X		
10	Damage Encountered					
	04 System object damage state	X	X	X	X	X
	44 Partial system object damage	X	X	X	X	X
20	Machine Support					
	02 Machine check					X
	03 Function check					X

Exception		Operands 1	2	3	4	Other
22	Object Access					
	01 Object not found	X	X	X	X	
	02 Object destroyed	X	X	X	X	
	03 Object suspended	X	X	X	X	
24	Pointer Specification					
	01 Pointer does not exist	X	X	X	X	
	02 Pointer type invalid	X	X	X	X	
2E	Resource Control Limit					
	01 User Profile storage limit exceeded					X
32	Scalar Specification					
	02 Scalar attributes invalid		X		X	
	03 Scalar value invalid		X		X	
36	Space Management					
	01 Space Extension/Truncation					X
38	Template Specification					
	01 Template value invalid				X	

Compute Time Duration (CTD)

Op Code (Hex)	Operand 1	Operand 2	Operand 3	Operand 4
0454	Time duration	Time 1	Time 2	Instruction template

Operand 1: Packed decimal variable scalar.

Operand 2: Character scalar.

Operand 3: Character scalar.

Operand 4: Space pointer.

ILE access

```
CTD (
 var time_duration        : packed decimal;
 var time1                : aggregate;
 var time2                : aggregate;
     instruction_template : space pointer
)
```

Description: The time specified by operand 3 is subtracted from the time specified by operand 2 and the value of the results a duration is place in operand 1. Operand 4 defines the data definitional attributes for operands 1 through 3.

A negative value will be returned when the first operand is less than the second operand.

The following describes the instruction template.

- Instruction template — Char(*)
  - Instruction template size — UBin(4)
  - Operand 1 data definitional attribute template number — UBin(2)
  - Operand 2 data definitional attribute template number — UBin(2)
  - Operand 3 data definitional attribute template number — UBin(2)
  - Reserved (binary 0) — Char(2)
  - Operand 2 length — UBin(2)
  - Operand 3 length — UBin(2)
  - Reserved (binary 0) — Char(26)
  - Data definitional attribute template list — Char(*)
    - Size of the DDAT list — UBin(4)
    - Number of DDATs — UBin(2)
    - Reserved (binary 0) — Char(10)
    - DDAT offset (repeated) — UBin(4)
    - Data definitional attribute template (repeated) — Char(*)

A **data definitional attribute template (DDAT) number** is a number that corresponds to the relative position of a template in the *data definitional attribute template list*. For example the number 1 references the first template. The valid values for this field are 1, 2, and 3.

The DDAT for operand 1 must be valid for a time duration. The DDATs for operands 2 and 3 must be valid for a time and identical. Otherwise, a *template value invalid* (hex 3801) exception will be signaled.

Operand 2 length and **operand 3 length** are specified in number of bytes.

The **size of the DDAT list** is specified in bytes.

The **number of DDATs** is the count of DDATs specified for this instruction template. The maximum number of DDATs that can be specified is 3.

The **DDAT offset** is the number of bytes from the start of the DDAT list to the start of the specific DDAT. There should be as many *DDAT offsets* as there are DDATs specified.

A data definitional attribute template defines the presentation of the data. Each template describes the definitional attributes of the operands. The length of the character operands will be defined by the template. For a further description of the data definitional attribute template, see "Data Definitional Attribute Template" on page 3-5

Authorization Required

- None.

Lock Enforcement

- None.

Exceptions

Exception		Operands 1	2	3	4	Other
06	Addressing					
	01 Space addressing violation	X	X	X	X	
	02 Boundary alignment	X	X	X	X	
	03 Range	X	X	X	X	
	06 Optimized addressability invalid	X	X	X	X	
08	Argument/Parameter					
	01 Parameter reference violation	X	X	X	X	
0C	Computation					
	16 Data format		X	X		
	17 Data value		X	X		
10	Damage Encountered					
	04 System object damage state	X	X	X	X	X
	44 Partial system object damage	X	X	X	X	X
20	Machine Support					
	02 Machine check					X
	03 Function check					X
22	Object Access					
	01 Object not found	X	X	X	X	

Exception		Operands 1	2	3	4	Other
	02 Object destroyed	X	X	X	X	
	03 Object suspended	X	X	X	X	
24	Pointer Specification					
	01 Pointer does not exist	X	X	X	X	
	02 Pointer type invalid	X	X	X	X	
2E	Resource Control Limit					
	01 User Profile storage limit exceeded					X
32	Scalar Specification					
	02 Scalar attributes invalid		X		X	
	03 Scalar value invalid		X		X	
36	Space Management					
	01 Space Extension/Truncation					X
38	Template Specification					
	01 Template value invalid				X	

Compute Timestamp Duration (CTSD)

Op Code (Hex)	Operand 1	Operand 2	Operand 3	Operand 4
043C	Timestamp duration	Timestamp 1	Timestamp 2	Instruction template

Operand 1: Packed decimal variable scalar.

Operand 2: Character scalar.

Operand 3: Character scalar.

Operand 4: Space pointer.

ILE access

```
CTSD (
 var timestamp_duration    : packed decimal;
 var timestamp1            : aggregate;
 var timestamp2            : aggregate;
     instruction_template  : space pointer
)
```

Description: The timestamp specified by operand 3 is subtracted from the timestamp specified by operand 2 and the value of the results a duration is place in operand 1. Operand 4 defines the data definitional attributes for operands 1 through 3.

A negative value will be returned when the first operand is less than the second operand.

The following describes the instruction template.

• Instruction template	Char(*)
— Instruction template size	Bin(4)
— Operand 1 data definitional attribute template number	UBin(2)
— Operand 2 data definitional attribute template number	UBin(2)
— Operand 3 data definitional attribute template number	UBin(2)
— Reserved (binary 0)	Char(2)
— Operand 2 length	UBin(2)
— Operand 3 length	UBin(2)
— Reserved (binary 0)	Char(26)
— Data definitional attribute template list	Char(*)
- Size of the DDAT list	UBin(4)
- Number of DDATs	UBin(2)
- Reserved (binary 0)	Char(10)
- DDAT offset (repeated)	UBin(4)
- Data definitional attribute template (repeated)	Char(*)

A **data definitional attribute template (DDAT) number** is a number that corresponds to the relative position of a template in the *data definitional attribute template list*. For example the number 1 references the first template. The valid values for this field are 1, 2, and 3.

The DDAT for operand 1 must be valid for a timestamp duration. The DDATs for operands 2 and 3 must be valid for a timestamp and identical. Otherwise, a *emplate value invalid* (hex 3801) exception will be signaled.

Operand 2 length and **operand 3 length** are specified in number of bytes.

The **size of the DDAT list** is specified in bytes.

The **number of DDATs** is the count of DDATs specified for this instruction template. The maximum number of DDATs that can be specified is 3.

The **DDAT offset** is the number of bytes from the start of the DDAT list to the start of the specific DDAT. There should be as many *DDAT offsets* as there are DDATs specified.

A data definitional attribute template defines the presentation of the data. Each template describes the definitional attributes of the operands. The length of the character operands will be defined by the template. For a further description of the data definitional attribute template, see :hdref=DDAT.

Authorization Required

- None.

Lock Enforcement

- None.

Exceptions

Exception		Operands 1	2	3	4	Other
06	Addressing					
	01 Space addressing violation	X	X	X	X	
	02 Boundary alignment	X	X	X	X	
	03 Range	X	X	X	X	
	06 Optimized addressability invalid	X	X	X	X	
08	Argument/Parameter					
	01 Parameter reference violation	X	X	X	X	
0C	Computation					
	15 Data boundary overflow		X	X		
	16 Data format		X	X		
	17 Data value		X	X		
	18 Date boundary underflow		X	X		
10	Damage Encountered					
	04 System object damage state	X	X	X	X	X
	44 Partial system object damage	X	X	X	X	X
20	Machine Support					

Exception		Operands 1	2	3	4	Other
	02 Machine check					X
	03 Function check					X
22	Object Access					
	01 Object not found	X	X	X	X	
	02 Object destroyed	X	X	X	X	
	03 Object suspended	X	X	X	X	
24	Pointer Specification					
	01 Pointer does not exist	X	X	X	X	
	02 Pointer type invalid	X	X	X	X	
2E	Resource Control Limit					
	01 User Profile storage limit exceeded					X
32	Scalar Specification					
	02 Scalar attributes invalid		X		X	
	03 Scalar value invalid		X		X	
36	Space Management					
	01 Space Extension/Truncation					X
38	Template Specification					
	01 Template value invalid				X	

Convert Date (CVTD)

Op Code (Hex)	Operand 1	Operand 2	Operand 3
040F	Result date	Source date	Instruction template

Operand 1: Character variable scalar.

Operand 2: Character scalar.

Operand 3: Space pointer.

ILE access

```
CVTD (
 var result_date          : aggregate;
 var source_date          : aggregate;
     instruction_template : space pointer
)
```

Description: The date specified in operand 2 is converted to another calendar external or internal presentation and placed in operand 1. Operand 3 defines the data definitional attributes for operands 1 and 2.

The following describes the instruction template.

- Instruction template — Char(*)
  - Instruction template size — Bin(4)
  - Operand 1 data definitional attribute template number — UBin(2)
  - Operand 2 data definitional attribute template number — UBin(2)
  - Reserved (binary 0) — Char(2)
  - Operand 1 length — UBin(2)
  - Operand 2 length — UBin(2)
  - Reserved (binary 0) — Char(2)
  - Preferred/Found date format — UBin(2)
  - Preferred/Found date separator — Char(1)
  - Reserved (binary 0) — Char(23)
  - Data definitional attribute template list — Char(*)
    - Size of the DDAT list — UBin(4)
    - Number of DDATs — UBin(2)
    - Reserved (binary 0) — Char(10)
    - DDAT offset (repeated) — UBin(4)
    - Data definitional attribute template (repeated) — Char(*)

A **data definitional attribute template number** is a number that corresponds to the relative position of a template in the **data definitional attribute template list**. For example the number 1 references the first template. The valid values for this field are 1 and 2.

The DDATs for operands 1 and 2 must be valid for a date. Otherwise, a *template value invalid* (hex 3801) exception will be signaled.

Operand 1 length and **operand 2 length** are specified in number of bytes.

If the *data definitional attribute template numbers* for operands 1 and 2 are the same, only data validation is performed. The validation will check for format and data value correctness.

A format of unknown date, time, or timestamp will indicate that operand 2 will be scanned for a valid format. For a list of formats that can be scanned, see "Data Definitional Attribute Template" on page 3-5. With an unknown format, the **preferred/found format** and **preferred/found separator** can be specified to select an additional non-scannable format. This *preferred format* and *preferred separator* will be used first to find a matching format before scanning operand 2. When the *preferred format* and *preferred separator* have a hex value of zero, only the scan occurs.

When a format of unknown date, time, or timestamp is specified, the *preferred/found format* and *preferred/found separator* fields will be set to the format and separator found.

The **size of the DDAT list** is specified in bytes.

The **number of DDATs** is the count of DDATs specified for this instruction template. The maximum number of DDATs that can be specified is 2.

The **DDAT offset** is the number of bytes from the start of the DDAT list to the start of the specific DDAT. There should be as many *DDAT offsets* as there are DDATs specified.

A data definitional attribute template defines the presentation of the data. Each template describes the definitional attributes of the operands. The length of the character operands will be defined by the templates. For a further description of the data definitional attribute template, see "Data Definitional Attribute Template" on page 3-5.

Authorization Required

- None.

Lock Enforcement

- None.

Exceptions

Exception		Operands 1	2	3	Other
06	Addressing				
	01 Space addressing violation	X	X	X	
	02 Boundary alignment	X	X	X	
	03 Range	X	X	X	
	06 Optimized addressability invalid	X	X	X	
08	Argument/Parameter				
	01 Parameter reference violation	X	X	X	
0C	Computation				
	15 Data boundary overflow		X	X	
	16 Data format		X		

Exception		Operands 1	2	3	Other
	17 Data value		X		
	18 Date boundary underflow		X	X	
10	Damage Encountered				
	04 System object damage state	X	X	X	X
	44 Partial system object damage	X	X	X	X
20	Machine Support				
	02 Machine check				X
	03 Function check				X
22	Object Access				
	01 Object not found	X	X	X	
	02 Object destroyed	X	X	X	
	03 Object suspended	X	X	X	
24	Pointer Specification				
	01 Pointer does not exist	X	X	X	
	02 Pointer type invalid	X	X	X	
2E	Resource Control Limit				
	01 User Profile storage limit exceeded				X
32	Scalar Specification				
	02 Scalar attributes invalid		X	X	
	03 Scalar value invalid		X	X	
36	Space Management				
	01 Space Extension/Truncation				X
38	Template Specification				
	01 Template value invalid			X	

Convert Time (CVTT)

Op Code (Hex)	Operand 1	Operand 2	Operand 3
041F	Result time	Source time	Instruction template

Operand 1: Character variable scalar.

Operand 2: Character scalar.

Operand 3: Space pointer.

ILE access

```
CVTT (
 var result_time          : aggregate
 var source_time          : aggregate
     instruction_template : space pointer
)
```

Description: The time specified in operand 2 is converted to another external or internal presentation and placed in operand 1. Operand 3 defines the data definitional attributes for operands 1 and 2.

The following describes the instruction template.

- Instruction template — Char(*)
  - Instruction template size — Bin(4)
  - Operand 1 data definitional attribute template number — UBin(2)
  - Operand 2 data definitional attribute template number — UBin(2)
  - Reserved (binary 0) — Char(2)
  - Operand 1 length — UBin(2)
  - Operand 2 length — UBin(2)
  - Reserved (binary 0) — Char(2)
  - Preferred/Found time format — UBin(2)
  - Reserved (binary 0) — Char(1)
  - Preferred/Found time separator — Char(1)
  - Reserved (binary 0) — Char(22)
  - Data definitional attribute template list — Char(*)
    - Size of the DDAT list — UBin(4)
    - Number of DDATs — UBin(2)
    - Reserved (binary 0) — Char(10)
    - DDAT offset (repeated) — UBin(4)
    - Data definitional attribute template (repeated) — Char(*)

A **data definitional attribute template number** is a number that corresponds to the relative position of a template in the **data definitional attribute template list**. For example the number 1 references the first template. The valid values for this field are 1 and 2.

The DDATs for operands 1 and 2 must be valid for a time. Otherwise, a *template value invalid* (hex 3801) exception will be signaled.

Operand 1 length and **operand 2 length** are specified in number of bytes.

If the *data definitional attribute template numbers* for operands 1 and 2 are the same, only data validation is performed. The validation will check for format and data value correctness.

A format of unknown date, time, or timestamp will indicate that operand 2 will be scanned for a valid format. For a list of formats that can be scanned, see "Data Definitional Attribute Template" on page 3-5. With an unknown format, the **preferred/found format** and **preferred/found separator** can be specified to select an additional non-scannable format. This *preferred format* and *preferred separator* will be used first to find a matching format before scanning operand 2. When the *preferred format* and *preferred separator* have a hex value of zero, only the scan occurs.

When a format of unknown date, time, or timestamp is specified, the *preferred/found format* and *preferred/found separator* fields will be set to the format and separator found.

The **size of the DDAT list** is specified in bytes.

The **number of DDATs** is the count of DDATs specified for this instruction template. The maximum number of DDATs that can be specified is 2.

The **DDAT offset** is the number of bytes from the start of the DDAT list to the start of the specific DDAT. There should be as many DDAT offsets as there are DDATs specified.

A data definitional attribute template defines the presentation of the data. Each template describes the definitional attributes of the operands. The length of the character operands will be defined by the templates. For a further description of the data definitional attribute template, see "Data Definitional Attribute Template" on page 3-5.

Authorization Required

- None.

Lock Enforcement

- None.

Exceptions

Exception		Operands 1	2	3	Other
06	Addressing				
	01 Space addressing violation	X	X	X	
	02 Boundary alignment	X	X	X	
	03 Range	X	X	X	
	06 Optimized addressability invalid	X	X	X	
08	Argument/Parameter				
	01 Parameter reference violation	X	X	X	
0C	Computation				
	16 Data format		X		
	17 Data value		X		

Exception		Operands 1	2	3	Other
10	Damage Encountered				
	04 System object damage state	X	X	X	X
	44 Partial system object damage	X	X	X	X
20	Machine Support				
	02 Machine check				X
	03 Function check				X
22	Object Access				
	01 Object not found	X	X	X	
	02 Object destroyed	X	X	X	
	03 Object suspended	X	X	X	
24	Pointer Specification				
	01 Pointer does not exist	X	X	X	
	02 Pointer type invalid	X	X	X	
2E	Resource Control Limit				
	01 User Profile storage limit exceeded				X
32	Scalar Specification				
	02 Scalar attributes invalid		X	X	
	03 Scalar value invalid		X	X	
36	Space Management				
	01 Space Extension/Truncation				X
38	Template Specification				
	01 Template value invalid			X	

Convert Timestamp (CVTTS)

Op Code (Hex)	Operand 1	Operand 2	Operand 3
043F	Converted timestamp	Input timestamp	Instruction template

Operand 1: Character variable scalar.

Operand 2: Character scalar.

Operand 3: Space pointer.

ILE access

```
CVTTS (
 var result_timestamp     : aggregate
 var source_timestamp     : aggregate
     instruction_template : space pointer
)
```

Description: The timestamp specified in operand 2 is converted to another external or internal presentation and placed in operand 1. Operand 3 defines the data definitional attributes for operands 1 and 2.

The following describes the instruction template.

- Instruction template — Char(*)
  - Instruction template size — Bin(4)
  - Operand 1 data definitional attribute template number — UBin(2)
  - Operand 2 data definitional attribute template number — UBin(2)
  - Reserved (binary 0) — Char(2)
  - Operand 1 length — UBin(2)
  - Operand 2 length — UBin(2)
  - Reserved (binary 0) — Char(2)
  - Preferred/Found timestamp format — UBin(2)
  - Preferred/Found date separator — Char(1)
  - Preferred/Found time separator — Char(1)
  - Reserved (binary 0) — Char(22)
  - Data definitional attribute template list — Char(*)
    - Size of the DDAT list — UBin(4)
    - Number of DDATs — UBin(2)
    - Reserved (binary 0) — Char(10)
    - DDAT offset (repeated) — UBin(4)
    - Data definitional attribute template (repeated) — Char(*)

A **data definitional attribute template number** is a number that corresponds to the relative position of a template in the **data definitional attribute template list**. For example the number 1 references the first template. The valid values for this field are 1 and 2.

The DDATs for operands 1 and 2 must be valid for a timestamp. Otherwise, a *template value invalid* (hex 3801) exception will be issued.

Operand 1 length and **operand 2 length** are specified in number of bytes.

If the *data definitional attribute template numbers* for operands 1 and 2 are the same, only data validation is performed. The validation will check for format and data value correctness.

A format of unknown date, time, or timestamp will indicate that

A format of unknown date, time, or timestamp will indicate that operand 2 will be scanned for a valid format. For a list of formats that can be scanned, see "Data Definitional Attribute Template" on page 3-5. With an unknown format, the **preferred/found format** and **preferred/found separator** can be specified to select an additional non-scannable format. This *preferred format* and *preferred separator* will be used first to find a matching format before scanning operand 2. When the *preferred format* and *preferred separator* have a hex value of zero, only the scan occurs.

When a format of unknown date, time, or timestamp is specified, the *preferred/found format* and *preferred/found separator* fields will be set to the format and separator found.

The **size of the DDAT list** is specified in bytes.

The **number of DDATs** is the count of DDATs specified for this instruction template. The maximum number of DDATs that can be specified is 2.

The **DDAT offset** is the number of bytes from the start of the DDAT list to the start of the specific DDAT. There should be as many DDAT offsets as there are DDATs specified.

A data definitional attribute template defines the presentation of the data. Each template describes the definitional attributes of the operands. The length of the character operands will be defined by the templates. For a further description of the data definitional attribute template, see "Data Definitional Attribute Template" on page 3-5.

Authorization Required

- None.

Lock Enforcement

- None.

Exceptions

Exception		Operands 1	2	3	Other
06	Addressing				
	01 Space addressing violation	X	X	X	
	02 Boundary alignment	X	X	X	
	03 Range	X	X	X	
	06 Optimized addressability invalid	X	X	X	
08	Argument/Parameter				
	01 Parameter reference violation	X	X	X	
0C	Computation				

Exception		Operands 1	2	3	Other
	15 Data boundary overflow		X		
	16 Data format		X		
	17 Data value		X		
	18 Date boundary underflow		X		
10	Damage Encountered				
	04 System object damage state	X	X	X	X
	44 Partial system object damage	X	X	X	X
20	Machine Support				
	02 Machine check				X
	03 Function check				X
22	Object Access				
	01 Object not found	X	X	X	
	02 Object destroyed	X	X	X	
	03 Object suspended	X	X	X	
24	Pointer Specification				
	01 Pointer does not exist	X	X	X	
	02 Pointer type invalid	X	X	X	
2E	Resource Control Limit				
	01 User Profile storage limit exceeded				X
32	Scalar Specification				
	02 Scalar attributes invalid		X	X	
	03 Scalar value invalid		X	X	
36	Space Management				
	01 Space Extension/Truncation				X
38	Template Specification				
	01 Template value invalid			X	

Decrement Date (DECD)

Op Code (Hex)	Operand 1	Operand 2	Operand 3	Operand 4
0414	Result date	Source date	Duration	Instruction template

Operand 1: Character variable scalar.

Operand 2: Character scalar.

Operand 3: Packed decimal scalar.

Operand 4: Space pointer.

ILE access

```
DECD (
 var result_date          : aggregate;
 var source_date          : aggregate;
 var duration             : packed decimal;
     instruction_template : space pointer
)
```

Description: The date specified by operand 2 is decremented by the date duration specified by operand 3. The resulting date from the operation is placed in operand 1. Operand 4 defines the data definitional attributes for operands 1 through 3.

The following describes the instruction template.

• Instruction template	Char(*)
— Instruction template size	Bin(4)
— Operand 1 data definitional attribute template number	UBin(2)
— Operand 2 data definitional attribute template number	UBin(2)
— Operand 3 data definitional attribute template number	UBin(2)
— Operand 1 length	UBin(2)
— Operand 2 length	UBin(2)
— Operand 3 length	UBin(2)
- Fractional number of digits	Char(1)
- Total number of digits	Char(1)
— Input indicators	Char(2)
- End of month adjustment	Bit 0
0 = No adjustment	
1 = Adjustment	
- Tolerate data decimal errors	Bit 1
0 = No toleration	
1 = Tolerate	
- Reserved (binary 0)	Bit 2-15
— Output indicators	Char(2)

- End of month adjustment	Bit 0
0 = No adjustment	
1 = Adjustment	
- Reserved (binary 0)	Bit 1-15
— Reserved (binary 0)	Char(22)
— Data definitional attribute template list	Char(*)
- Size of the DDAT list	UBin(4)
- Number of DDATs	UBin(2)
- Reserved (binary 0)	Char(10)
- DDAT offset (repeated)	UBin(4)
- Data definitional attribute template (repeated)	Char(*)

A **data definitional attribute template (DDAT) number** is a number that corresponds to the relative position of a template in the **data definitional attribute template list**. For example the number 1 references the first template. The valid values for this field are 1, 2, and 3.

The DDATs for operands 1 and 2 must be valid for a date and identical. The DDAT for operand 3 must be valid for a date duration. Otherwise, a *template value invalid* (hex 3801) exception will be issued.

Operand 1 length and **operand 2 length** are specified in number of bytes.

The **input indicator, end of month adjustment**, is used to allow or disallow the occurrence of an end of month adjustment.

The input indicator, **tolerate decimal data errors**, is used to determine whether errors found in the packed data for the duration will generate exceptions or will be ignored. When the errors are to be *tolerated*, the following rules will apply:

1. An invalid sign nibble found in the packed data value will be changed to a hex F.
2. Any invalid decimal digits found in the packed data value will be forced to zero.
3. If all digits of a packed data value become zero, and no decimal overflow condition exists, the sign will be set to hex F. If all digits are zero and a decimal overflow condition exists, then the sign will not be changed, but its representation will be changed to the preferred sign code.

The **output indicator, end of month adjustment**, is used to indicate an end of month adjustment, when end of month adjustments are allowed.

End of month adjustment is the following concept. For SAA, the result of subtracting a 1 month duration from the Gregorian date 03/31/1989 is 02/28/1989. The days portion is adjusted to fit the month, 31 is changed to 28. When this happens, the *end of month adjustment output indicator* is set to *on*.

When *end of month adjustments are not allowed*, the month and year definitions in the data definition attribute template must have values greater than zero, otherwise a *template value invalid* (hex 3801) exception will be signalled. The result of subtracting a 1 month duration from the Gregorian date 03/31/1989 is 03/01/1989, when the definition of a month is 30 days.

The **size of the DDAT list** is specified in bytes.

The **number of DDATs** is the count of DDATs specified for this instruction template. The maximum number of DDATs that can be specified is 3.

The **DDAT offset** is the number of bytes from the start of the DDAT list to the start of the specific DDAT. There should be as many *DDAT offsets* as there are DDATs specified.

A data definitional attribute template defines the presentation of the data. Each template describes definitional attributes of the operands. The length of the date and date duration character operands will be defined by the template. For a further description of the data definitional attribute template, see "Data Definitional Attribute Template" on page 3-5.

Authorization Required

- None.

Lock Enforcement

- None.

Exceptions

Exception		Operands 1	2	3	4	Other
06	Addressing					
	01 Space addressing violation	X	X	X	X	
	02 Boundary alignment	X	X	X	X	
	03 Range	X	X	X	X	
	06 Optimized addressability invalid	X	X	X	X	
08	Argument/Parameter					
	01 Parameter reference violation	X	X	X	X	
0C	Computation					
	02 Decimal data			X		
	15 Date boundary overflow	X	X			
	16 Data format		X			
	17 Data value		X			
	18 Date boundary underflow	X	X			
10	Damage Encountered					
	04 System object damage state	X	X	X	X	X
	44 Partial system object damage	X	X	X	X	X
20	Machine Support					
	02 Machine check					X
	03 Function check					X
22	Object Access					
	01 Object not found	X	X	X	X	
	02 Object destroyed	X	X	X	X	
	03 Object suspended	X	X	X	X	
24	Pointer Specification					
	01 Pointer does not exist	X	X	X	X	
	02 Pointer type invalid	X	X	X	X	

Exception		Operands 1	2	3	4	Other
2E	Resource Control Limit					
	01 User Profile storage limit exceeded					X
32	Scalar Specification					
	02 Scalar attributes invalid		X		X	
	03 Scalar value invalid		X		X	
36	Space Management					
	01 Space Extension/Truncation					X
38	Template Specification					
	01 Template value invalid				X	

Decrement Time (DECT)

Op Code (Hex)	Operand 1	Operand 2	Operand 3	Operand 4
0444	Result time	Source time	Duration	Instruction template

Operand 1: Character variable scalar.

Operand 2: Character scalar.

Operand 3: Packed decimal scalar.

Operand 4: Space pointer.

ILE access

```
DECT (
 var result_time          : aggregate;
 var source_time          : aggregate;
 var duration             : packed decimal;
     instruction_template : space pointer
)
```

Description: The time specified by operand 2 is decremented by the time duration specified by operand 3. The resulting time from the operation is placed in operand 1. Operand 4 defines the data definitional attributes for operands 1 through 3.

The following describes the instruction template.

• Instruction template	Char(*)
— Instruction template size	Bin(4)
— Operand 1 data definitional attribute template number	UBin(2)
— Operand 2 data definitional attribute template number	UBin(2)
— Operand 3 data definitional attribute template number	UBin(2)
— Operand 1 length	UBin(2)
— Operand 2 length	UBin(2)
— Operand 3 length	UBin(2)
- Fractional number of digits	Char(1)
- Total number of digits	Char(1)
— Input indicators	Char(2)
- Reserved (binary 0)	Bit 0
- Tolerate data decimal errors 0 = No toleration 1 = Tolerate	Bit 1
- Reserved (binary 0)	Bit 2-15
— Reserved (binary 0)	Char(24)
— Data definitional attribute template list	Char(*)
- Size of the DDAT list	UBin(4)

- Number of DDATs	UBin(2)
- Reserved (binary 0)	Char(10)
- DDAT offset (repeated)	UBin(4)
- Data definitional attribute template (repeated)	Char(*)

A **data definitional attribute template (DDAT) number** is a number that corresponds to the relative position of a template in the **data definitional attribute template list**. For example the number 1 references the first template. The valid values for this field are 1, 2, and 3.

The DDATs for operands 1 and 2 must be valid for a time and identical. The DDAT for operand 3 must be valid for a time duration. Otherwise, a *template value invalid* (hex 3801) exception will be issued.

Operand 1 length and **operand 2 length** are specified in number of bytes.

The **input indicator, tolerate decimal data errors**, is used to determine whether errors found in the packed data for the duration will generate exceptions or will be ignored. When the errors are to be *tolerated*, the following rules will apply:

1. An invalid sign nibble found in the packed data value will be changed to a hex F.
2. Any invalid decimal digits found in the packed data value will be forced to zero.
3. If all digits of a packed data value become zero, and no decimal overflow condition exists, the sign will be set to hex F. If all digits are zero and a decimal overflow condition exists, then the sign will not be changed, but its representation will be changed to the preferred sign code.

The **size of the DDAT list** is specified in bytes.

The **number of DDATs** is the count of DDATs specified for this instruction template. The maximum number of DDATs that can be specified is 3.

The **DDAT offset** is the number of bytes from the start of the DDAT list to the start of the specific DDAT. There should be as many *DDAT offsets* as there are DDATs specified.

A data definitional attribute template defines the presentation of the data. Each template describes the definitional attributes of the operands. The length of the time and time duration character operands will be defined by the templates. For a further description of the data definitional attribute template, see "Data Definitional Attribute Template" on page 3-5.

Authorization Required

- None.

Lock Enforcement

- None.

Exceptions

Exception		Operands 1	2	3	4	Other
06	Addressing					
	01 Space addressing violation	X	X	X	X	
	02 Boundary alignment	X	X	X	X	
	03 Range	X	X	X	X	

Exception		Operands 1	2	3	4	Other
	06 Optimized addressability invalid	X	X	X	X	
08	Argument/Parameter					
	01 Parameter reference violation	X	X	X	X	
0C	Computation					
	02 Decimal data			X		
	16 Data format		X			
	17 Data value		X			
10	Damage Encountered					
	04 System object damage state	X	X	X	X	X
	44 Partial system object damage	X	X	X	X	X
20	Machine Support					
	02 Machine check					X
	03 Function check					X
22	Object Access					
	01 Object not found	X	X	X	X	
	02 Object destroyed	X	X	X	X	
	03 Object suspended	X	X	X	X	
24	Pointer Specification					
	01 Pointer does not exist	X	X	X	X	
	02 Pointer type invalid	X	X	X	X	
2E	Resource Control Limit					
	01 User Profile storage limit exceeded					X
32	Scalar Specification					
	02 Scalar attributes invalid		X		X	
	03 Scalar value invalid		X		X	
36	Space Management					
	01 Space Extension/Truncation					X
38	Template Specification					
	01 Template value invalid				X	

Decrement Timestamp (DECTS)

Op Code (Hex)	Operand 1	Operand 2	Operand 3	Operand 4
042C	Result timestamp	Source timestamp	Duration	Instruction template

Operand 1: Character variable scalar.

Operand 2: Character scalar.

Operand 3: Packed decimal scalar.

Operand 4: Space pointer.

ILE access

```
DECTS (
 var result_timestamp     : aggregate;
 var source_timestamp     : aggregate;
 var duration             : packed decimal;
     instruction_template : space pointer
)
```

Description: The timestamp specified by operand 2 is decremented by the date, time, or timestamp duration specified by operand 3. The resulting timestamp from the operation is placed in operand 1. Operand 4 defines the data definitional attributes for operands 1 through 3.

The following describes the instruction template.

- Instruction template — Char(*)
  - Instruction template size — Bin(4)
  - Operand 1 data definitional attribute template number — UBin(2)
  - Operand 2 data definitional attribute template number — UBin(2)
  - Operand 3 data definitional attribute template number — UBin(2)
  - Operand 1 length — UBin(2)
  - Operand 2 length — UBin(2)
  - Operand 3 length — UBin(2)
    - Fractional number of digits — Char(1)
    - Total number of digits — Char(1)
  - Input indicators — Char(2)
    - End of month adjustment — Bit 0
      - 0 = No adjustment
      - 1 = Adjustment
    - Tolerate data decimal errors — Bit 1
      - 0 = No toleration
      - 1 = Tolerate
    - Reserved (binary 0) — Bit 2-15
  - Output indicators — Char(2)

- End of month adjustment	Bit 0
0 = No adjustment	
1 = Adjustment	
- Reserved (binary 0)	Bit 1-15
— Reserved (binary 0)	Char(22)
— Data definitional attribute template list	Char(*)
- Size of the DDAT list	UBin(4)
- Number of DDATs	UBin(2)
- Reserved (binary 0)	Char(10)
- DDAT offset (repeated)	UBin(4)
- Data definitional attribute template (repeated)	Char(*)

A **data definitional attribute template (DDAT) number** is a number that corresponds to the relative position of a template in the **data definitional attribute template list**. For example the number 1 references the first template. The valid values for this field are 1, 2, and 3.

The DDATs for operands 1 and 2 must be valid for a timestamp and identical. The DDAT for operand 3 must be valid for a timestamp duration. Otherwise, a *template value invalid* (hex 3801) exception will be issued.

Operand 1 length and **operand 2 length** are specified in number of bytes.

The **input indicator, end of month adjustment**, is used to allow or disallow the occurrence of an end of month adjustment.

The input indicator, **tolerate decimal data errors**, is used to determine whether errors found in the packed data for the duration will generate exceptions or will be ignored. When the errors are to be *tolerated*, the following rules will apply:

1. An invalid sign nibble found in the packed data value will be changed to a hex F.
2. Any invalid decimal digits found in the packed data value will be forced to zero.
3. If all digits of a packed data value become zero, and no decimal overflow condition exists, the sign will be set to hex F. If all digits are zero and a decimal overflow condition exists, then the sign will not be changed, but its representation will be changed to the preferred sign code.

The **output indicator, end of month adjustment**, is used to indicate an end of month adjustment, when end of month adjustments are allowed.

End of month adjustment is the following concept. For SAA, the result of subtracting a 1 month duration from the date 03/31/1989 is 02/28/1989. The days portion is adjusted to fit the month, 31 is changed to 28. When this happens, the *end of month adjustment output indicator* is set to *on*.

When *end of month adjustments are not allowed*, the month and year definitions in the data definition attribute template must have values greater than zero, otherwise a *template value invalid* (hex 3801) exception will be signaled. The result of subtracting a 1 month duration from the Gregorian date 03/31/1989 is 03/01/1989, when the definition of a month is 30 days.

The **size of the DDAT list** is specified in bytes.

The **number of DDATs** is the count of DDATs specified for this instruction template. The maximum number of DDATs that can be specified is 3.

The <u>DDAT offset</u>
is the number of bytes from the start of the DDAT list to the start of the specific DDAT. There should be as many *DDAT offsets* as there are DDATs specified.

A data definitional attribute template defines the presentation of the data. Each template describes the definitional attributes of the operands. The length of the timestamp and duration character operands will be defined by the template. For a further description of the data definitional attribute template, see "Data Definitional Attribute Template" on page 3-5.

Authorization Required

- None.

Lock Enforcement

- None.

Exceptions

Exception		Operands 1	2	3	4	Other
06	Addressing					
	01 Space addressing violation	X	X	X	X	
	02 Boundary alignment	X	X	X	X	
	03 Range	X	X	X	X	
	06 Optimized addressability invalid	X	X	X	X	
08	Argument/Parameter					
	01 Parameter reference violation	X	X	X	X	
0C	Computation					
	02 Decimal data			X		
	15 Date boundary overflow	X	X			
	16 Data format		X			
	17 Data value		X			
	18 Date boundary underflow	X	X			
10	Damage Encountered					
	04 System object damage state	X	X	X	X	X
	44 Partial system object damage	X	X	X	X	X
20	Machine Support					
	02 Machine check					X
	03 Function check					X
22	Object Access					
	01 Object not found	X	X	X	X	
	02 Object destroyed	X	X	X	X	
	03 Object suspended	X	X	X	X	
24	Pointer Specification					
	01 Pointer does not exist	X	X	X	X	

Exception		Operands 1	2	3	4	Other
	02 Pointer type invalid	X	X	X	X	
2E	Resource Control Limit					
	01 User Profile storage limit exceeded					X
32	Scalar Specification					
	02 Scalar attributes invalid		X		X	
	03 Scalar value invalid		X		X	
36	Space Management					
	01 Space Extension/Truncation					X
38	Template Specification					
	01 Template value invalid				X	

Increment Date (INCD)

Op Code (Hex)	Operand 1	Operand 2	Operand 3	Operand 4
0404	Result date	Source date	Duration	Instruction template

Operand 1: Character variable scalar.

Operand 2: Character scalar.

Operand 3: Packed decimal scalar.

Operand 4: Space pointer.

ILE access

```
INCD (
 var result_date          : aggregate;
 var source_date          : aggregate;
 var duration             : packed decimal;
     instruction_template : space pointer
)
```

Description: The date specified by operand 2 is incremented by the date duration specified by operand 3. The resulting date from the operation is placed in operand 1. Operand 4 defines the data definitional attributes for operands 1 through 3.

The following describes the instruction template.

Field	Type
• Instruction template	Char(*)
— Instruction template size	Bin(4)
— Operand 1 data definitional attribute template number	UBin(2)
— Operand 2 data definitional attribute template number	UBin(2)
— Operand 3 data definitional attribute template number	UBin(2)
— Operand 1 length	UBin(2)
— Operand 2 length	UBin(2)
— Operand 3 length	UBin(2)
- Fractional number of digits	Char(1)
- Total number of digits	Char(1)
— Input indicators	Char(2)
- End of month adjustment	Bit 0
0 = No adjustment	
1 = Adjustment	
- Tolerate data decimal errors	Bit 1
0 = No toleration	
1 = Tolerate	
- Reserved (binary 0)	Bit 2-15
— Output indicators	Char(2)

- End of month adjustment	Bit 0
0 = No adjustment	
1 = Adjustment	
- Reserved (binary 0)	Bit 1-15
— Reserved (binary 0)	Char(22)
— Data definitional attribute template list	Char(*)
- Size of the DDAT list	UBin(4)
- Number of DDATs	UBin(2)
- Reserved (binary 0)	Char(10)
- DDAT offset (repeated)	UBin(4)
- Data definitional attribute template (repeated)	Char(*)

A **data definitional attribute template (DDAT) number** is a number that corresponds to the relative position of a template in the **data definitional attribute template list**. For example the number 1 references the first template. The valid values for this field are 1, 2, and 3.

The DDATs for operands 1 and 2 must be valid for a date and identical. The DDAT for operand 3 must be valid for a date duration. Otherwise, a *template value invalid* (hex 3801) exception will be issued.

Operand 1 length and **operand 2 length** are specified in number of bytes.

The **input indicator, end of month adjustment**, is used to allow or disallow the occurrence of an end of month adjustment.

The input indicator, **tolerate decimal data errors**, is used to determine whether errors found in the packed data for the duration will generate exceptions or will be ignored. When the errors are to be *tolerated*, the following rules will apply:

1. An invalid sign nibble found in the packed data value will be changed to a hex F.
2. Any invalid decimal digits found in the packed data value will be forced to zero.
3. If all digits of a packed data value become zero, and no decimal overflow condition exists, the sign will be set to hex F. If all digits are zero and a decimal overflow condition exists, then the sign will not be changed, but its representation will be changed to the preferred sign code.

The **output indicator, end of month adjustment**, is used to indicate an end of month adjustment, when end of month adjustments are allowed.

End of month adjustment is the following concept. For SAA, the result of adding a 1 month duration to the date 01/31/1989 is 02/28/1989. The days portion is adjusted to fit the month, 31 is changed to 28. When this happens,the *end of month adjustment output indicator* is set to *on*.

When *end of month adjustments are not allowed*, the month and year definitions in the data definition attribute template must have values greater than zero, otherwise a *template value invalid* (hex 3801) exception will be signaled. The result of adding a 1 month duration to the Gregorian date 01/31/1989 is 03/02/1989, when the definition of a month is 30 days.

The **size of the DDAT list** is specified in bytes.

The **number of DDATs** is the count of DDATs specified for this instruction template. The maximum number of DDATs that can be specified is 3.

The **DDAT offset** is the number of bytes from the start of the DDAT list to the start of the specific DDAT. There should be as many *DDAT offsets* as there are DDATs specified.

A data definitional attribute template defines the presentation of the data. Each template describes definitional attributes of the operands. The length of the date and date duration character operands will be defined by the template. For a further description of the data definitional attribute template, see "Data Definitional Attribute Template" on page 3-5.

Authorization Required

- None.

Lock Enforcement

- None.

Exceptions

Exception		Operands 1	2	3	4	Other
06	Addressing					
	01 Space addressing violation	X	X	X	X	
	02 Boundary alignment	X	X	X	X	
	03 Range	X	X	X	X	
	06 Optimized addressability invalid	X	X	X	X	
08	Argument/Parameter					
	01 Parameter reference violation	X	X	X	X	
0C	Computation					
	02 Decimal data			X		
	15 Date boundary overflow	X	X			
	16 Data format		X			
	17 Data value		X			
	18 Date boundary underflow	X	X			
10	Damage Encountered					
	04 System object damage state	X	X	X	X	X
	44 Partial system object damage	X	X	X	X	X
20	Machine Support					
	02 Machine check					X
	03 Function check					X
22	Object Access					
	01 Object not found	X	X	X	X	
	02 Object destroyed	X	X	X	X	
	03 Object suspended	X	X	X	X	
24	Pointer Specification					
	01 Pointer does not exist	X	X	X	X	
	02 Pointer type invalid	X	X	X	X	

Exception		Operands				
		1	2	3	4	Other
2E	Resource Control Limit					
	01 User Profile storage limit exceeded					X
32	Scalar Specification					
	02 Scalar attributes invalid		X		X	
	03 Scalar value invalid		X		X	
36	Space Management					
	01 Space Extension/Truncation					X
38	Template Specification					
	01 Template value invalid				X	

Increment Time (INCT)

Op Code (Hex)	Operand 1	Operand 2	Operand 3	Operand 4
0434	Result time	Source time	Duration	Instruction template

Operand 1: Character variable scalar.

Operand 2: Character scalar.

Operand 3: Packed decimal scalar.

Operand 4: Space pointer.

ILE access

```
INCT (
 var result_time          : aggregate;
 var source_time          : aggregate;
 var duration             : packed decimal;
     instruction_template : space pointer
)
```

Description: The time specified by operand 2 is incremented by the time duration specified by operand 3. The resulting time from the operation is placed in operand 1. Operand 4 defines the data definitional attributes for operands 1 through 3.

The following describes the instruction template.

- Instruction template — Char(*)
  - Instruction template size — Bin(4)
  - Operand 1 data definitional attribute template number — UBin(2)
  - Operand 2 data definitional attribute template number — UBin(2)
  - Operand 3 data definitional attribute template number — UBin(2)
  - Operand 1 length — UBin(2)
  - Operand 2 length — UBin(2)
  - Operand 3 length — UBin(2)
    - Fractional number of digits — Char(1)
    - Total number of digits — Char(1)
  - Input indicators — Char(2)
    - Reserved (binary 0) — Bit 0
    - Tolerate data decimal errors — Bit 1
      - 0 = No toleration
      - 1 = Tolerate
    - Reserved (binary 0) — Bit 2-15
  - Reserved (binary 0) — Char(24)
  - Data definitional attribute template list — Char(*)
    - Size of the DDAT list — UBin(4)

- Number of DDATs	UBin(2)
- Reserved (binary 0)	Char(10)
- DDAT offset (repeated)	UBin(4)
- Data definitional attribute template (repeated)	Char(*)

A **data definitional attribute template number (DDAT)** is a number that corresponds to the relative position of a template in the **data definitional attribute template list**. For example the number 1 references the first template. The valid values for this field are 1, 2, and 3.

The DDATs for operands 1 and 2 must be valid for a time and identical. The DDAT for operand 3 must be valid for a time duration. Otherwise, a *template value invalid* (hex 3801) exception will be issued.

Operand 1 length and **operand 2 length** are specified in number of bytes.

The input indicator, **tolerate decimal data errors**, is used to determine whether errors found in the packed data for the duration will generate exceptions or will be ignored. When the errors are to be *tolerated*, the following rules will apply:

1. An invalid sign nibble found in the packed data value will be changed to a hex F.
2. Any invalid decimal digits found in the packed data value will be forced to zero.
3. If all digits of a packed data value become zero, and no decimal overflow condition exists, the sign will be set to hex F. If all digits are zero and a decimal overflow condition exists, then the sign will not be changed, but its representation will be changed to the preferred sign code.

The **size of the DDAT list** is specified in bytes.

The **number of DDATs** is the count of DDATs specified for this instruction template. The maximum number of DDATs that can be specified is 3.

The **DDAT offset** is the number of bytes from the start of the DDAT list to the start of the specific DDAT. There should be as many *DDAT offsets* as there are DDATs specified.

A data definitional attribute template defines the presentation of the data. Each template describes the definitional attributes of the operands. The length of the time and time duration character operands will be defined by the templates. For a further description of the data definitional attribute template, see "Data Definitional Attribute Template" on page 3-5.

Authorization Required

- None.

Lock Enforcement

- None.

Exceptions

Exception		Operands 1	2	3	4	Other
06	Addressing					
	01 Space addressing violation	X	X	X	X	
	02 Boundary alignment	X	X	X	X	
	03 Range	X	X	X	X	

Exception		Operands 1	2	3	4	Other
	06 Optimized addressability invalid	X	X	X	X	
08	Argument/Parameter					
	01 Parameter reference violation	X	X	X	X	
0C	Computation					
	02 Decimal data			X		
	16 Data format		X			
	17 Data value		X			
10	Damage Encountered					
	04 System object damage state	X	X	X	X	X
	44 Partial system object damage	X	X	X	X	X
20	Machine Support					
	02 Machine check					X
	03 Function check					X
22	Object Access					
	01 Object not found	X	X	X	X	
	02 Object destroyed	X	X	X	X	
	03 Object suspended	X	X	X	X	
24	Pointer Specification					
	01 Pointer does not exist	X	X	X	X	
	02 Pointer type invalid	X	X	X	X	
2E	Resource Control Limit					
	01 User Profile storage limit exceeded					X
32	Scalar Specification					
	02 Scalar attributes invalid		X		X	
	03 Scalar value invalid		X		X	
36	Space Management					
	01 Space Extension/Truncation					X
38	Template Specification					
	01 Template value invalid				X	

Increment Timestamp (INCTS)

Op Code (Hex)	Operand 1	Operand 2	Operand 3	Operand 4
040C	Result timestamp	Source timestamp	Duration	Instruction template

Operand 1: Character variable scalar.

Operand 2: Character scalar.

Operand 3: Packed decimal scalar.

Operand 4: Space pointer.

ILE access

```
INCTS (
 var result_timestamp     : aggregate;
 var source_timestamp     : aggregate;
 var duration             : packed decimal;
     instruction_template : space pointer
)
```

Description: The timestamp specified by operand 2 is incremented by the date, time, or timestamp duration specified by operand 3. The resulting timestamp from the operation is placed in operand 1. Operand 4 defines the data definitional attributes for operands 1 through 3.

The following describes the instruction template.

• Instruction template	Char(*)
— Instruction template size	Bin(4)
— Operand 1 data definitional attribute template number	UBin(2)
— Operand 2 data definitional attribute template number	UBin(2)
— Operand 3 data definitional attribute template number	UBin(2)
— Operand 1 length	UBin(2)
— Operand 2 length	UBin(2)
— Operand 3 length	UBin(2)
- Fractional number of digits	Char(1)
- Total number of digits	Char(1)
— Input indicators	Char(2)
- End of month adjustment	Bit 0
0 = No adjustment	
1 = Adjustment	
- Tolerate data decimal errors	Bit 1
0 = No toleration	
1 = Tolerate	
- Reserved (binary 0)	Bit 2-15
— Output indicators	Char(2)

- End of month adjustment	Bit 0
0 = No adjustment	
1 = Adjustment	
- Reserved (binary 0)	Bit 1-15
— Reserved (binary 0)	Char(22)
— Data definitional attribute template list	Char(*)
- Size of the DDAT list	UBin(4)
- Number of DDATs	UBin(2)
- Reserved (binary 0)	Char(10)
- DDAT offset (repeated)	UBin(4)
- Data definitional attribute template (repeated)	Char(*)

A **data definitional attribute template (DDAT) number** is a number that corresponds to the relative position of a template in the **data definitional attribute template list**. For example the number 1 references the first template. The valid values for this field are 1, 2, and 3.

The DDATs for operands 1 and 2 must be valid for a timestamp and identical. The DDAT for operand 3 must be valid for a timestamp duration. Otherwise, a *template value invalid* (hex 3801) exception will be issued.

Operand 1 length and **operand 2 length** are specified in number of bytes.

The **input indicator, end of month adjustment**, is used to allow or disallow the occurrence of an end of month adjustment.

The input indicator, **tolerate decimal data errors**, is used to determine whether errors found in the packed data for the duration will generate exceptions or will be ignored. When the errors are to be *tolerated*, the following rules will apply:

1. An invalid sign nibble found in the packed data value will be changed to a hex F.
2. Any invalid decimal digits found in the packed data value will be forced to zero.
3. If all digits of a packed data value become zero, and no decimal overflow condition exists, the sign will be set to hex F. If all digits are zero and a decimal overflow condition exists, then the sign will not be changed, but its representation will be changed to the preferred sign code.

The **output indicator, end of month adjustment**, is used to indicate an end of month adjustment, when end of month adjustments are allowed.

End of month adjustment is the following concept. For SAA, the result of adding a 1 month duration to the date 01/31/1989 is 02/28/1989. The days portion is adjusted to fit the month, 31 is changed to 28. When this happens, the *end of month adjustment output indicator* is set to *on*.

When *end of month adjustments are not allowed*, the month and year definitions in the data definition attribute template must have values greater than zero, otherwise a *template value invalid* (hex 3801) exception will be signaled. The result of adding a 1 month duration to the Gregorian date 01/31/1989 is 03/02/1989, when the definition of a month is 30 days.

The **size of the DDAT list** is specified in bytes.

The **number of DDATs** is the count of DDATs specified for this instruction template. The maximum number of DDATs that can be specified is 3.

The **DDAT offset** is the number of bytes from the start of the DDAT list to the start of the specific DDAT. There should be as many *DDAT offsets* as there are DDATs specified.

A data definitional attribute template defines the presentation of the data. Each template describes the definitional attributes of the operands. The length of the timestamp and duration character operands will be defined by the template. For a further description of the data definitional attribute template, see "Data Definitional Attribute Template" on page 3-5.

Authorization Required

- None.

Lock Enforcement

- None.

Exceptions

Exception		Operands 1	2	3	4	Other
06	Addressing					
	01 Space addressing violation	X	X	X	X	
	02 Boundary alignment	X	X	X	X	
	03 Range	X	X	X	X	
	06 Optimized addressability invalid	X	X	X	X	
08	Argument/Parameter					
	01 Parameter reference violation	X	X	X	X	
0C	Computation					
	02 Decimal data			X		
	15 Date boundary overflow	X	X			
	16 Data format		X			
	17 Data value		X			
	18 Date boundary underflow	X	X			
10	Damage Encountered					
	04 System object damage state	X	X	X	X	X
	44 Partial system object damage	X	X	X	X	X
20	Machine Support					
	02 Machine check					X
	03 Function check					X
22	Object Access					
	01 Object not found	X	X	X	X	
	02 Object destroyed	X	X	X	X	
	03 Object suspended	X	X	X	X	
24	Pointer Specification					
	01 Pointer does not exist	X	X	X	X	
	02 Pointer type invalid	X	X	X	X	

Exception		Operands 1	2	3	4	Other
2E	Resource Control Limit					
	01 User Profile storage limit exceeded					X
32	Scalar Specification					
	02 Scalar attributes invalid		X		X	
	03 Scalar value invalid		X		X	
36	Space Management					
	01 Space Extension/Truncation					X
38	Template Specification					
	01 Template value invalid				X	

Chapter 4. Pointer/Name Resolution Addressing Instructions

This chapter describes the instructions used for pointer and name resolution functions. These instructions are in alphabetic order. See Appendix A, "Instruction Summary," for an alphabetic summary of all the instructions.

Compare Pointer for Object Addressability (CMPPTRA) 4-3
Compare Pointer for Space Addressability (CMPPSPAD) 4-5
| Compare Pointers for Equality (CMPPTRE) 4-7
Compare Pointer Type (CMPPTRT) 4-9
Copy Bytes with Pointers (CPYBWP) 4-12
Resolve Data Pointer (RSLVDP) 4-14
Resolve System Pointer (RSLVSP) 4-17
Set Space Pointer from Pointer (SETSPPFP) 4-22
Set System Pointer from Pointer (SETSPFP) 4-24

© Copyright IBM Corp. 1991, 1993

Compare Pointer for Object Addressability (CMPPTRA)

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand 3 [4]
CMPPTRAB 1CD2	Branch options	Compare operand 1	Compare operand 2	Branch targets
CMPPTRAI 18D2	Indicator options	Compare operand 1	Compare operand 2	Indicator targets

Operand 1: Data pointer, space pointer, system pointer, or instruction pointer.

Operand 2: Data pointer, space pointer, system pointer, or instruction pointer.

Operand 3 [4]:

- *Branch Form*—Instruction number, relative instruction number, branch point, or instruction pointer.
- *Indicator Form*—Numeric variable scalar or character variable scalar.

Description: The object addressed by operand 1 is compared with the object addressed by operand 2 to determine if both operands are addressing the same object. Based on the comparison, the resulting condition is used with the extender to transfer control (branch form) or to assign a value to each of the indicator operands (indicator form).

If operand 1 is a data pointer, a space pointer, or a system pointer, operand 2 may be any pointer type except for instruction pointer in any combination. An *equal* condition occurs if the pointers are addressing the same object. For space pointers and data pointers, only the space they are addressing is considered in the comparison. That is, the space offset portion of the pointer is ignored.

For system pointer compare operands, an *equal* condition occurs if the system pointer is compared with a space pointer or data pointer that addresses the space that is associated with the object that is addressed by the system pointer. For example, a space pointer that addresses a byte in a space associated with a system object compares equal with a system pointer that addresses the system object.

For instruction pointer comparisons, both operands must be instruction pointers; otherwise, an *invalid pointer type* (hex 2402) exception is signaled. An *equal* condition occurs when both instruction pointers are addressing the same instruction in the same program. A *not equal* condition occurs if the instruction pointers are not addressing the same instruction in the same program.

A *pointer does not exist* (hex 2401) exception is signaled if a pointer does not exist in either of the operands.

Resultant Conditions

- Equal
- Not equal

Authorization Required

- Retrieve
  - Contexts referenced for address resolution

Lock Enforcement

- Materialize
  - Contexts referenced for address resolution

Exceptions

Exception		Operands 1	2	3	4	Other
06	Addressing					
	01 Space addressing violation	X	X	X	X	
	02 Boundary alignment	X	X	X	X	
	03 Range	X	X	X	X	
	06 Optimized addressability invalid	X	X	X	X	
08	Argument/parameter					
	01 Parameter reference violation	X	X	X	X	
0A	Authorization					
	01 Unauthorized for operation	X	X			
10	Damage encountered					
	04 System object damage state	X	X	X	X	X
	05 authority verification terminated due to damaged object					X
	44 Partial system object damage	X	X	X	X	X
1A	Lock state					
	01 Invalid lock state	X	X			
1C	Machine-dependent exception					
	03 Machine storage limit exceeded					X
20	Machine support					
	02 Machine check					X
	03 Function check					X
22	Object access					
	01 Object not found	X	X	X	X	
	02 Object destroyed	X	X	X	X	
	03 Object suspended	X	X	X	X	
	07 authority verification terminated due to destroyed object					X
	08 object compressed					X
24	Pointer specification					
	01 Pointer does not exist	X	X	X	X	
	02 Pointer type invalid	X	X	X	X	
2E	Resource control limit					
	01 user profile storage limit exceeded					X
36	Space management					
	01 space extension/truncation					X

Compare Pointer for Space Addressability (CMPPSPAD)

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand 3 [4-6]
CMPPSPADB 1CE6	Branch options	Compare Operand 1	Compare Operand 2	Branch targets
CMPPSPADI 18E6	Indicator options	Compare Operand 1	Compare Operand 2	Indicator targets

Operand 1: Space pointer or data pointer.

Operand 2: Numeric variable scalar, character variable scalar, numeric variable array, character variable array, space pointer, or data pointer.

Operand 3 [4-6]:

- *Branch Form*-Instruction number, relative instruction number, branch point, or instruction pointer.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Description: The space addressability contained in the pointer specified by operand 1 is compared with the space addressability defined by operand 2.

The value of the operand 1 pointer is compared based on the following:

- If operand 2 is a scalar data object (element or array), the space addressability of that data object is compared with the space addressability contained in the operand 1 pointer.
- If operand 2 is a pointer, it must be a space pointer or data pointer, and the space addressability contained in the pointer is compared with the space addressability contained in the operand 1 pointer.

Based on the results of the comparison, the resulting condition is used with the extender to transfer control (branch form) or to assign a value to each of the indicator operands (indicator form). If the operands are not in the same space, the resultant condition is *unequal*. If the operands are in the same space and the offset into the space of operand 1 is larger or smaller than the offset of operand 2, the resultant condition is *high* or *low*, respectively. An *equal* condition occurs only if the operands are in the same space at the same offset. Therefore, the resultant conditions (*high, low, equal*, and *unequal*) are mutually exclusive. Consequently, if you specify that an action be taken upon the nonexistence of a condition, this results in the action being taken upon the occurrence of any of the other three possible conditions. For example, a branch not high would result in the branch being taken on a *low, equal*, or *unequal* condition.

The *object destroyed* (hex 2202) exception, *optimized addressability invalid* (hex 0606) exception, *parameter reference violation* (hex 0801) exception, and *pointer does not exist* (hex 2401) exception are not signaled when operand 1 or operand 2 is a space pointer machine object or when operand 2 is a scalar based on a space pointer machine object. This occurs when the space pointer machine object contains an internal machine value that indicates one of these error conditions exists. If the corresponding exception is not signaled, the resulting condition of the comparison operation is not defined other than that it will be one of the four valid resultant conditions for this instruction.

When the Override Program Attributes (OVRPGATR) instruction is used to override this instruction, the *pointer does not exist*, (hex 2401) exception is not signaled when operand 1 or operand 2 is a space pointer (i.e. either a space pointer data object or a space pointer machine object). Furthermore, some comparisons involving space pointers are defined even when one or both of the compare operands is a pointer subject to the pointer does not exist condition. Specifically, if both compare operands are subject to the pointer does not exist condition, the resultant condition is *equal*. When one space pointer is set and one is subject to the pointer does not exist condition, the resultant condition is

unequal, but undefined with respect to comparisons which include specification of the *high* or *low* conditions.

Resultant Conditions

- High
- Low
- Equal
- Unequal

Exceptions

Exception		Operands 1	2	3 [4-6]	Other
06	Addressing				
	01 space addressing violation	X	X	X	
	02 boundary alignment	X	X		
	03 range	X	X		
	04 external data object not found	X	X		
	06 optimized addressability invalid	X	X		
08	Argument/parameter				
	01 parameter reference violation	X	X		
10	Damage encountered				
	04 system object damage state	X	X	X	X
	44 partial system object damage	X	X	X	X
1C	Machine-dependent exception				
	03 machine storage limit exceeded				X
20	Machine support				
	02 machine check				X
	03 function check				X
22	Object access				
	01 object not found	X	X	X	
	02 object destroyed	X	X	X	
	03 object suspended	X	X	X	
	08 object compressed				X
24	Pointer specification				
	01 pointer does not exist	X	X	X	
	02 pointer type invalid	X	X	X	
2E	Resource control limit				
	01 user profile storage limit exceeded				X
36	Space management				
	01 space extension/truncation				X

Compare Pointers for Equality (CMPPTRE)

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand 3 [4]
CMPPTREB 1C12	Branch options	Compare operand 1	Compare operand 2	Branch targets
CMPPTREI 1812	Indicator options	Compare operand 1	Compare operand 2	Indicator targets

Operand 1: Data pointer, space pointer, system pointer, instruction pointer, invocation pointer, procedure pointer, label pointer, or suspend pointer

Operand 2: Data pointer, space pointer, system pointer, instruction pointer, invocation pointer, procedure pointer, label pointer, or suspend pointer

Operand 3 [4]:

- *Branch Form* — Instruction number, relative instruction number, branch point, or instruction pointer.
- *Indicator Form* — Numeric variable scalar or character variable scalar.

Description: The pointer specified by operand 1 is compared with the pointer specified by operand 2 to determine if both operands are of the same type and contain equal values. Based on the comparison, the resulting condition is used with the extender to transfer control (branch form) or to assign a value to each of the indicator operands (indicator form).

Pointers may be specified for operands 1 and 2 in any combination. An *equal* condition occurs if the pointers are of the same type and contain the same value, or if neither pointer has been set. If one pointer is set and the other is not, a *not equal* condition occurs.

System pointers and data pointers are not resolved by this instruction. The comparison result is undefined when an unresolved pointer is supplied for one or both operands.

Note that any authorities stored in a resolved system pointer are part of the pointer. Thus system pointers pointing to the same object, but with different levels of authority, will compare as *not equal*.

Since any pointer type may be specified for this instruction, the *invalid pointer type* (hex 2402) exception is not signaled except for pointers used as a base for the operands. Similarly, since the instruction accepts unset pointers, the *pointer does not exist* (hex 2401) exception is not signaled except for pointers used as a base for the operands.

Resultant Conditions

- Equal
- Not equal

Exceptions

Exception		Operands 1	2	3	4	Other
06	Addressing					
	01 Space addressing violation	X	X	X	X	
	02 Boundary alignment	X	X	X	X	
	03 Range	X	X	X	X	
	06 Optimized addressability invalid	X	X	X	X	

Exception		Operands 1	2	3	4	Other
08	Argument/parameter					
	01 Parameter reference violation	X	X	X	X	
0A	Authorization					
	01 Unauthorized for operation	X	X			
10	Damage encountered					
	04 System object damage state	X	X	X	X	X
	05 authority verification terminated due to damaged object					X
	44 Partial system object damage	X	X	X	X	X
1A	Lock state					
	01 Invalid lock state	X	X			
1C	Machine-dependent exception					
	03 Machine storage limit exceeded					X
20	Machine support					
	02 Machine check					X
	03 Function check					X
22	Object access					
	01 Object not found	X	X	X	X	
	02 Object destroyed	X	X	X	X	
	03 Object suspended	X	X	X	X	
	07 authority verification terminated due to destroyed object					X
	08 object compressed					X
24	Pointer specification					
	01 Pointer does not exist	X	X	X	X	
	02 Pointer type invalid	X	X	X	X	
2E	Resource control limit					
	01 user profile storage limit exceeded					X
36	Space management					
	01 space extension/truncation					X

Compare Pointer Type (CMPPTRT)

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand 3 [4]
CMPPTRTB 1CE2	Branch options	Compare operand 1	Compare operand 2	Branch targets
CMPPTRTI 18E2	Indicator options	Compare operand 1	Compare operand 2	Indicator targets

Operand 1: Data pointer, space pointer, system pointer, instruction pointer, invocation pointer, procedure pointer, label pointer, or suspend pointer

Operand 2: Character(1) scalar or null.

Operand 3 [4]:

- *Branch Form*—Instruction number, relative instruction number, branch point, or instruction pointer.
- *Indicator Form*—Numeric variable scalar or character variable scalar.

Extender

Description: The instruction compares the pointer type currently in operand 1 with the character scalar identified by operand 2. Based on the comparison, the resulting condition is used with the extender to transfer control (branch form) or to assign a value to each of the indicator operands (indicator form).

Operand 1 can specify a space pointer machine object only when operand 2 is null.

If operand 2 is null or if operand 2 specifies a comparison value of hex 00, an equal condition occurs if a pointer does not exist in the storage area identified by operand 1.

Following are the allowable values for operand 2:

Hex 00 — A pointer does not exist at this location
Hex 01 — System pointer
Hex 02 — Space pointer
Hex 03 — Data pointer
Hex 04 — Instruction pointer
Hex 05 — Invocation pointer
Hex 06 — Procedure pointer
Hex 07 — Label pointer
Hex 08 — Suspend pointer

Resultant Conditions

- Equal
- Not equal

Authorization Required

- Retrieve
  - Contexts referenced for address resolution

Lock Enforcement

- Materialize
  - Contexts referenced for address resolution

Exceptions

Exception		Operands 1	2	3	4	Other
06	Addressing					
	01 Space addressing violation	X	X	X	X	
	02 Boundary alignment	X	X	X	X	
	03 Range	X	X	X	X	
	06 Optimized addressability invalid	X	X	X	X	
08	Argument/parameter					
	01 Parameter reference violation	X	X	X	X	
0A	Authorization					
	01 Unauthorized for operation	X				
10	Damage encountered					
	04 System object damage state	X	X	X	X	X
	05 authority verification terminated due to damaged object					X
	44 Partial system object damage	X	X	X	X	X
1A	Lock state					
	01 Invalid lock state	X				
1C	Machine-dependent exception					
	03 Machine storage limit exceeded					X
20	Machine support					
	02 Machine check					X
	03 Function check					X
22	Object access					
	01 Object not found	X	X	X	X	
	02 Object destroyed	X	X	X	X	
	03 Object suspended	X	X	X	X	
	07 authority verification terminated due to destroyed object					X
	08 object compressed					X
24	Pointer specification					
	01 Pointer does not exist	X	X	X	X	
	02 Pointer type invalid	X	X	X	X	
2E	Resource control limit					
	01 user profile storage limit exceeded					X
32	Scalar specification					
	03 Scalar value invalid			X		

Exception		Operands 1	2	3	4	Other
36	Space management					
	01 space extension/truncation					X

Copy Bytes with Pointers (CPYBWP)

Op Code (Hex)	Operand 1	Operand 2
0132	Receiver	Source

| *Operand 1*: Character variable scalar, space pointer, data pointer, system pointer, instruction pointer, | invocation pointer, procedure pointer, label pointer, or suspend pointer

Operand 2: Character variable scalar, space pointer, data pointer, system pointer, instruction pointer, | invocation pointer, procedure pointer, label pointer, suspend pointer, or null

Description: This instruction copies either the pointer value or the byte string specified for the source operand into the receiver operand depending upon whether or not a space pointer machine object is specified as one of the operands.

If either operand is a character variable scalar, it can have a length as great as 16776191 bytes.

Operations involving space pointer machine objects perform a pointer value copy operation for only space pointer values or the pointer does not exist state. Due to this, a space pointer machine object may only be specified as an operand in conjunction with another pointer or a null second operand. The pointer does not exist state is copied from the source to the receiver pointer without signaling the *pointer does not exist* (hex 2401) exception. Source pointer data objects must either be not set or contain a space pointer value when being copied into a receiver space pointer machine object. Receiver pointer data objects will be set with either the system default pointer does not exist value or the space pointer value from a source space pointer machine object.

Normal pointer alignment checking is performed on a pointer data object specified as an operand in conjunction with a space pointer machine object.

Operations not involving space pointer machine objects, those involving just data objects as operands, perform a byte string copy of the data for the specified operands.

The value of the byte string specified by operand 2 is copied to the byte string specified by operand 1 (no padding done).

The byte string identified by operand 2 can contain the storage forms of both scalars and pointers. Normal pointer alignment checking is not done.

\hen the Override Program Attributes (OVRPGATR) instruction is not used to override CPYBWP, the only alignment requirement is that the space addressability alignment of the two operands must be to the same position relative to a 16-byte multiple boundary. A *boundary alignment* (hex 0602) exception is signaled if the alignment is incorrect. The pointer attributes of any complete pointers in the source are preserved if they can be completely copied into the receiver. Partial pointer storage forms are copied into the receiver as scalar data. Scalars in the source are copied to the receiver as scalars.

When the OVRPGATR instruction is used to override this instruction, the alignment requirement is removed. If the space addressability alignment of the two operands is the same relative to 16-byte multiple boundary, then this instruction will work the same as stated above. If the space addressability alignment is different, then this instruction will work like a Copy Bytes Left Adjusted (CPYBLA) and the pointer attributes of any complete pointers in the source are not preserved in the receiver.

If a pointer data object operand contains a data pointer value upon execution of the instruction, the pointer storage form is copied rather than the scalar described by the data pointer value. The character variable scalar reference allowed on either operand cannot be described through a data pointer value.

The length of the operation is equal to the length of the shorter of the two operands. The copying begins with the two operands left-adjusted and proceeds until completion of the shorter operand.

Operand 1 can specify a space pointer machine object only when operand 2 is a space pointer or null.

If operand 2 is null, operand 1 must define a pointer reference; otherwise, an *invalid operand type* (hex 2A06) exception is signaled by the Create Program instruction. When operand 2 is null, the pointer identified by operand 1 is set to the system default pointer does not exist value.

Exceptions

Exception		Operands 1	Operands 2	Other
06	Addressing			
	01 Space addressing violation	X	X	
	02 Boundary alignment	X	X	
	03 Range	X	X	
	06 Optimized addressability invalid	X	X	
08	Argument/parameter			
	01 Parameter reference violation	X	X	
10	Damage encountered			
	04 System object damage state	X	X	X
	44 Partial system object damage	X	X	X
1C	Machine-dependent exception			
	03 machine storage limit exceeded			X
20	Machine support			
	02 Machine check			X
	03 Function check			X
22	Object access			
	01 Object not found	X	X	
	02 Object destroyed	X	X	
	03 Object suspended	X	X	
	08 object compressed			X
24	Pointer specification			
	01 Pointer does not exist	X	X	
	02 Pointer type invalid	X	X	
2E	Resource control limit			
	01 user profile storage limit exceeded			X
36	Space management			
	01 space extension/truncation			X

Resolve Data Pointer (RSLVDP)

Op Code (Hex)	Operand 1	Operand 2	Operand 3
0163	Pointer for addressability to data object	Data object identification	Program

Operand 1: Data pointer.

Operand 2: Character(32) scalar (fixed-length) or null.

Operand 3: System pointer or null.

Description: A data pointer with addressability to and the attributes of an external scalar data element is returned in the storage area identified by operand 1. The following describes the instruction's function when operand 2 is null:

- If operand 1 does not contain a data pointer, an exception is signaled.
- If the data pointer specified by operand 1 is not resolved and has an initial value declaration, the instruction resolves the data pointer to the external scalar that the initial value describes. The initial value defines the external scalar to be located and, optionally, defines the program in which it is to be located. If the program name is specified in the initial value, only that program's activation entry is searched for the external scalar. If no program is specified, programs associated with the activation entries in the current activation group in which the program is executing, are searched in reverse order of the activation entries, and operand 3 is ignored. The current activation group for non-bound programs is the default activation group whose state is the same as the state of the process at the time the instruction is run.
- If the data pointer is currently resolved and defines an existing scalar, the instruction causes no operation, and no exception is signaled.

The following describes the instruction's function when operand 2 is not null:

- A data pointer that is resolved to the external scalar identified by operand 2 is returned in operand 1. Operand 2 is a 32-byte value that provides the name of the external scalar to be located.
- Operand 3 specifies a system pointer that identifies the program whose activation is to be searched for the external scalar definition. If operand 3 is null, the instruction searches all activations in the activaction group from which the instruction is executed, starting with the most recent activation and continuing to the oldest. The activation under which the instruction is issued also participates in the search. If operand 3 is not null, the instruction searches the activation of the program addressed by the system pointer.

Authorization Required

- Retrieve
  - Contexts referenced for address resolution

Lock Enforcement

- Materialize
  - Contexts referenced for address resolution

Exceptions

Exception		Operands 1	2	3	Other
06	Addressing				
	01 Space addressing violation	X	X	X	
	02 Boundary alignment	X	X	X	
	03 Range	X	X	X	
	04 External data object not found	X			
	06 Optimized addressability invalid	X	X	X	
08	Argument/parameter				
	01 Parameter reference violation	X	X	X	
0A	Authorization				
	01 Unauthorized for operation	X		X	
10	Damage encountered				
	04 System object damage state	X	X	X	X
	05 authority verification terminated due to damaged object				X
	44 Partial system object damage	X	X	X	X
1A	Lock state				
	01 Invalid lock state			X	
1C	Machine-dependent exception				
	03 Machine storage limit exceeded				X
20	Machine support				
	02 Machine check				X
	03 Function check				X
22	Object access				
	01 Object not found	X	X	X	
	02 Object destroyed	X	X	X	
	03 Object suspended	X	X	X	
	07 authority verification terminated due to destroyed object				X
	08 object compressed				X
24	Pointer specification				
	01 Pointer does not exist	X	X	X	
	02 Pointer type invalid	X	X	X	
	04 Pointer not resolved				X
2E	Resource control limit				
	01 user profile storage limit exceeded				X
32	Scalar specification				
	01 Scalar type invalid	X	X	X	
	02 Scalar attributes invalid		X		
	03 Scalar value invalid		X		
36	Space management				

Exception	Operands 1	2	3	Other
01 space extension/truncation				X

Resolve System Pointer (RSLVSP)

Op Code (Hex)	Operand 1	Operand 2	Operand 3	Operand 4
0164	Pointer for addressability to object	Object identification and required authorization	Context through which objects is to be located	Authority to be set[1]

Operand 1: System pointer.

Operand 2: Character(34) scalar (fixed-length) or null.

Operand 3: System pointer or null.

Operand 4: Character(2) scalar (fixed-length) or null.

ILE access

```
RSLVSP (
 var ptr_to_object      : system pointer;
 var objid_and_authreq  : aggregate; OR
                          null operand;
 var context            : system pointer; OR
                          null operand;
 var authority_to_set   : aggregate OR
                          null operand
)
```

Description: This instruction locates an object identified by a symbolic address and stores the object's addressability and authority[1] in a system pointer. A resolved system pointer is returned in operand 1 with addressability to a system object and the requested authority currently available to the process for the object.

Note: The ownership flag is never set in the system pointer.

Operand 2 specifies the symbolic identification of the object to be located. Operand 3 identifies the context to be searched in order to locate the object. Operand 4 identifies the authority states to be set in the pointer. First, the instruction locates an object based on operands 2 and 3. Then, the instruction sets the appropriate authority states in the system pointer.

The following describes the instruction's function when operand 2 is null:

- If operand 1 does not contain a system pointer, an exception is signaled.
- If the system pointer specified by operand 1 is not resolved but has an initial value declaration, the instruction resolves the system pointer to the object that the initial value describes. The initial value defines the following:
  - Object to be located (by *type code*, *subtype code*, and *object name*)
  - *Context* to be searched to locate the object (optional)
  - Minimum *required authorization* required for the object

  If a *context* is specified, only that context is referenced to locate the object, and operand 3 is ignored. If no context is specified, the context(s) located by the process name resolution list is

[1] Programs executing in user-domain may not assign authority in the resulting system pointer. The value in operand 4 is ignored and no exception is raised.

used to locate the object, and operand 3 is ignored. If the object is of a type that can only be addressed through the machine context, then only the machine context is searched, and the context (if any) identified in the initial value or identified in operand 3 is ignored.

If the minimum *required authorization* in the initial value is not set (binary 0), the instruction resolves the operand 1 system pointer to the first object encountered with the designated type code, subtype code, and object name without regard to the authorization available to the process for the object. If one or more authorization (or ownership) states are required (signified by binary 1's), the context(s) is searched until an object is encountered with the designated type, subtype, and name and for which the process currently has all required authorization states.

- If the system pointer specified by operand 1 is currently resolved to address an existing object, the instruction does not modify the addressability contained in the pointer and causes only the authority attribute in the pointer to be modified based on operand 4.

If operand 2 is not null, the operand 1 system pointer is resolved to the object identified by operand 2 in the context(s) specified by operand 3. The format of operand 2 is as follows:

• Object specification	Char(32)
— Type code	Char(1)
— Subtype code	Char(1)
— Object name	Char(30)
• Required authorization (1 = required)	Char(2)
— Object control	Bit 0
— Object management	Bit 1
— Authorized pointer	Bit 2
— Space authority	Bit 3
— Retrieve	Bit 4
— Insert	Bit 5
— Delete	Bit 6
— Update	Bit 7
— Ownership	Bit 8
— Excluded	Bit 9
— Authority List Management	Bit 10
— Reserved (binary 0)	Bit 11-15

The allowed type codes are as follows:

Hex 01 = Access group
Hex 02 = Program
Hex 03 = Module
Hex 04 = Context
Hex 06 = Byte string space
Hex 07 = Journal space
Hex 08 = User profile
Hex 09 = Journal port
Hex 0A = Queue
Hex 0B = Data space
Hex 0C = Data space index
Hex 0D = Cursor
Hex 0E = Index

Hex 0F = Commit block
Hex 10 = Logical unit description
Hex 11 = Network description
Hex 12 = Controller description
Hex 13 = Dump space
Hex 14 = Class of Service Description
Hex 15 = Mode description
Hex 16 = Network interface description
Hex 17 = Connection list
Hex 18 = Queue space
Hex 19 = Space
Hex 1A = Process control space
Hex 1B = Authorization list
Hex 1C = Dictionary

All other codes are reserved. If other codes are specified, they cause a *scalar value invalid* (hex 3203) exception to be signaled.

Operand 3 identifies the context in which to locate the object identified by operand 2. If operand 3 is null, then the contexts identified in the process name resolution list are searched in the order in which they appear in the list. If operand 3 is not null, the system pointer specified must address a context, and only this context is used to locate the object. If the object is of a type that can only be addressed through the machine context, then only the machine context is searched, and operand 3 and the process name resolution list are ignored.

If the **required authorization** field in operand 2 is not set (all values set to 0), the instruction resolves the operand 1 system pointer to the first object encountered with the designated **type code**, **subtype code**, and **object name** without regard to the authorization currently available to the process. If one or more authorization (or ownership) states are required (signified by binary 1's), the context is searched until an object is encountered with the designated *type code, subtype code, object name*, and the user profiles governing the instruction's execution that have all the required authorization states.

Once addressability has been set in the pointer, operand 4 is used to determine which, if any, of the object authority states is to be set into the pointer. Only the object authority states correlating with bits 0 through 7, that is, *object control* through *update*, can be set into the pointer. This restriction applies whether the authority mask controlling which authorities to set in the pointer comes from operand 4, operand 2, or the initial value for the system pointer.

If operand 4 is null, the object authority states required to locate the object are set in the pointer. This required object authority is as specified in operand 2 or in the initial value for operand 1 if operand 2 is null. If the process does not currently have authorized pointer authority for the object, no authority is stored in the system pointer, and no exception is signaled.

If operands 2 and 4 are null and operand 1 is a resolved system pointer, the authority states in the pointer are not modified.

If operand 4 is not null, it specifies the object authority states to be set in the resolved system pointer. The format of operand 4 is as follows:

• Requested authorization (1 = set authority)	Char(2)
– Object control	Bit 0
– Object management	Bit 1
– Authorized pointer	Bit 2
– Space authority	Bit 2

- Retrieve — Bit 4
- Insert — Bit 5
- Delete — Bit 6
- Update — Bit 7
- Reserved (binary 0) — Bits 8-15

The authority states set in the resolved system pointer are based on the following:

- The authority already stored in the pointer can be increased only when the process has authorized pointer authority to the referenced object. If this authority is not available and the pointer was resolved by this instruction, the authority in the operand 1 system pointer is set to the not set state, and no exception is signaled. If operand 2 is null, if operand 1 is a resolved system pointer containing authority, and if authorized pointer authority is not available to the process, additional authorities cannot be stored in the pointer.
- If the process does not currently have all the authority states requested in operand 4, only the requested and available states are set in the pointer, and no exception is signaled.
- Note that the authority stored in the operand 1 system pointer is a source of authority applies to this instruction when operand 2 is null and operand 1 is a resolved system pointer with authority stored in it.

Authorization Required

- Retrieve
  - Contexts referenced for address resolution (including operand 3)

Lock Enforcement

- Materialization
  - Contexts referenced for address resolution (including operand 3)

Exceptions

Exception		Operands 1	2	3	4	Other
06	Addressing					
	01 Space addressing violation	X	X	X	X	
	02 Boundary alignment	X	X	X	X	
	03 Range	X	X	X	X	
	06 Optimized addressability invalid	X	X	X	X	
08	Argument/parameter					
	01 Parameter reference violation	X	X	X	X	
0A	Authorization					
	01 Unauthorized for operation	X		X		
10	Damage encountered					
	02 Machine context damage state					X
	04 System object damage state	X	X	X	X	X
	05 authority verification terminated due to damaged object					X
	44 Partial system object damage	X	X	X	X	X

Exception		Operands 1	2	3	4	Other
1A	Lock state					
	01 Invalid lock state	X		X		
20	Machine support					
	02 Machine check					X
	03 Function check					X
22	Object access					
	01 Object not found	X	X	X	X	
	02 Object destroyed	X	X	X	X	
	03 Object suspended	X	X	X	X	
	07 authority verification terminated due to destroyed object					X
	08 object compressed					X
24	Pointer specification					
	01 Pointer does not exist	X	X	X	X	
	02 Pointer type invalid	X	X	X	X	
	04 Pointer not resolved					X
2E	Resource control limit					
	01 user profile storage limit exceeded					X
32	Scalar specification					
	02 Scalar attributes invalid		X		X	
	03 Scalar value invalid		X		X	
36	Space management					
	01 space extension/truncation					X

Set Space Pointer from Pointer (SETSPPFP)

Op Code (Hex)	Operand 1	Operand 2
0022	Receiver	Source Pointer

Operand 1: Space pointer.

Operand 2: Data pointer, system pointer, or space pointer.

Description: A space pointer is returned in operand 1 with the addressability to a space object from the pointer specified by operand 2.

The meaning of the pointers allowed for operand 2 is as follows:

Pointer	Meaning
Data pointer or space pointer	The space pointer returned in operand 1 is set to address of the leftmost byte of the byte string addressed by the source pointer for operand 2.
System pointer	The space pointer returned in operand 1 is set to address the first byte of the space contained in the system object addressed by the system pointer for operand 2. The space object addressed is either the created system space or an associated space for the system object addressed by the system pointer. If the operand 2 system pointer addresses a system object with no associated space, the *invalid space reference* (hex 0605) exception is signaled.

The *object destroyed* (hex 2202) exception, *optimized addressability invalid* (hex 0606) exception, *parameter reference violation* (hex 0801) exception, and *pointer does not exist* (hex 2401) exception are not signaled when operand 1 and operand 2 are space pointer machine objects. This occurs when operand 2 contains an internal machine value that indicates one of these error conditions exists. If the corresponding exception is not signaled, operand 1 is set with an internal machine value that preserves the exception condition that existed for operand 2. The appropriate exception condition will be signaled for either pointer when a subsequent attempt is made to reference the space data that the pointer addresses.

Authorization Required

- Space authority
  - Operand 2 (if a system pointer)
- Retrieve
  - Contexts referenced for address resolution

Lock Enforcement

- Materialize
  - Contexts referenced for address resolution

Exceptions

Exception		Operands 1	2	Other
06	Addressing			
	01 space addressing violation	X	X	
	02 boundary alignment	X	X	
	03 range	X	X	

Exception		Operands 1	Operands 2	Other
	04 external data object not found		X	
	05 invalid space reference		X	
	06 optimized addressability invalid	X	X	
08	Argument/parameter			
	01 parameter reference violation			X
0A	Authorization			
	01 unauthorized for operation		X	
10	Damage encountered			
	04 system object damage state	X	X	X
	05 authority verification terminated due to damaged object			X
	44 partial system object damage	X	X	X
1A	Lock state			
	01 invalid lock state		X	
1C	Machine-dependent exception			
	03 machine storage limit exceeded			X
20	Machine support			
	02 machine check			X
	03 function check			X
22	Object access			
	01 object not found	X	X	
	02 object destroyed	X	X	
	03 object suspended	X	X	
	07 authority verification terminated due to destroyed object			X
	08 object compressed			X
24	Pointer specification			
	01 pointer does not exist	X	X	
	02 pointer type invalid	X	X	
	03 pointer addressing invalid object		X	
2E	Resource control limit			
	01 user profile storage limit exceeded			X
36	Space management			
	01 space extension/truncation			X

Set System Pointer from Pointer (SETSPFP)

Op Code (Hex)	Operand 1	Operand 2
0032	Receiver	Source pointer

Operand 1: System pointer.

Operand 2: System pointer, space pointer, data pointer, or instruction pointer.

Description: This instruction returns a system pointer to the system object address by the supplied pointer.

If operand 2 is a system pointer, then a system pointer addressing the same object is returned in operand 1 containing the same authority as the input pointer.

If operand 2 is a space pointer or a data pointer, then a system pointer addressing the system object that contains the associated space addressed by operand 2 is returned in operand 1.

If operand 2 is an instruction pointer, then a system pointer addressing the program system object that contains the instruction addressed by operand 2 is returned in operand 1.

If operand 2 is an unresolved system pointer or data pointer, the pointer is resolved first.

Authorization Required

- Retrieve
  - Contexts referenced for address resolution

Lock Enforcement

- Materialization
  - Contexts referenced for address resolution

Exceptions

Exception		Operands 1	Operands 2	Other
06	Addressing			
	01 space addressing violation	X	X	
	02 boundary alignment	X	X	
	03 range	X	X	
	04 external data object not found	X	X	
	06 optimized addressability invalid	X	X	
08	Argument/parameter			
	01 parameter reference violation			X
0A	Authorization			
	01 unauthorized for operation		X	
10	Damage encountered			
	02 machine context damage			X
	04 system object damage state	X	X	X

Exception		Operands 1	Operands 2	Other
	05 authority verification terminated due to damaged object			X
	44 partial system object damage	X	X	X
1A	Lock state			
	01 invalid lock state		X	
1C	Machine-dependent exception			
	03 machine storage limit exceeded			X
20	Machine support			
	02 machine check			X
	03 function check			X
22	Object access			
	01 object not found		X	
	02 object destroyed	X	X	
	03 object suspended	X	X	
	07 authority verification terminated due to destroyed object			X
	08 object compressed			X
24	Pointer specification			
	01 pointer does not exist	X	X	
	02 pointer type invalid	X	X	
2E	Resource control limit			
	01 user profile storage limit exceeded			X
32	Scalar specification			
	01 scalar type invalid	X	X	
36	Space management			
	01 space extension/truncation			X

Chapter 5. Space Addressing Instructions

This chapter describes the instructions used for space addressing. These instructions are in alphabetic order. For an alphabetic summary of all the instructions, see Appendix A, "Instruction Summary"

Add Space Pointer (ADDSPP) 5-3
Compare Space Addressability (CMPSPAD) 5-5
Set Data Pointer (SETDP) 5-7
Set Data Pointer Addressability (SETDPADR) 5-9
Set Data Pointer Attributes (SETDPAT) 5-11
Set Space Pointer (SETSPP) 5-14
Set Space Pointer with Displacement (SETSPPD) 5-16
Set Space Pointer Offset (SETSPPO) 5-18
Store Space Pointer Offset (STSPPO) 5-20
Subtract Space Pointer Offset (SUBSPP) 5-22
Subtract Space Pointers For Offset (SUBSPPFO) 5-24

© Copyright IBM Corp. 1991, 1993

Add Space Pointer (ADDSPP)

Op Code (Hex)	Operand 1	Operand 2	Operand 3
0083	Receiver Pointer	Source Pointer	Increment

Operand 1: Space pointer.

Operand 2: Space pointer.

Operand 3: Binary scalar.

Description: This instruction adds a signed or unsigned binary value to the offset of a space pointer. The value of the binary scalar represented by operand 3 is added to the space address contained in the space pointer specified by operand 2, and the result is stored in the space pointer identified by operand 1. I.e.

```
Operand 1 = Operand 2 + Operand 3
```

Operand 3 can have a positive or negative value. The space object that the pointer is addressing is not changed by the instruction.

Operand 2 must contain a space pointer when the execution of the instruction is initiated; otherwise, an *invalid pointer type* (hex 2402) exception is signaled. When the addressability in a space pointer is modified, the instruction signals a *space addressing* (hex 0601) exception only when the space address to be stored in the pointer has a negative offset value or when the offset addresses beyond the largest space allocatable in the object. This maximum offset value is dependent on the size and packaging of the object containing the space and is independent of the actual size of the space allocated. If the exception is signaled by this instruction for this reason, the pointer is not modified by the instruction. Attempts to use a pointer whose offset value lies between the currently allocated extent of the space and the maximum allocatable extent of the space cause the *space addressing* (hex 0601) exception to be signaled.

The *object destroyed* (hex 2202) exception, *optimized addressability invalid* (hex 0606) exception, *parameter reference violation* (hex 0801) exception, and *pointer does not exist* (hex 2401) exception are not signaled when operand 1 and operand 2 are space pointer machine objects. This occurs when operand 2 contains an internal machine value that indicates one of these error conditions exists. If the corresponding exception is not signaled, operand 1 is set with an internal machine value that preserves the exception condition that existed for operand 2. The appropriate exception condition will be signaled for either pointer when a subsequent attempt is made to reference the space data that the pointer addresses.

Exceptions

Exception		Operands 1	2	3[4-6]	Other
06	Addressing				
	01 space addressing violation	X	X	X	
	02 boundary alignment	X	X	X	
	03 range	X	X	X	
	06 optimized addressability invalid	X	X	X	
08	Argument/parameter				

Exception		Operands 1	2	3[4-6]	Other
	01 parameter reference violation	X	X	X	
10	Damage encountered				
	04 system object damage state	X	X	X	X
	44 partial system object damage	X	X	X	X
1C	Machine-dependent exception				
	03 machine storage limit exceeded				X
20	Machine support				
	02 machine check				X
	03 function check				X
22	Object access				
	01 object not found	X	X	X	
	02 object destroyed	X	X	X	
	03 object suspended	X	X	X	
	08 object compressed				X
24	Pointer specification				
	01 pointer does not exist	X	X	X	
	02 pointer type invalid	X	X	X	
2E	Resource control limit				
	01 user profile storage limit exceeded				X
36	Space management				
	01 space extension/truncation				X

Compare Space Addressability (CMPSPAD)

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand 3 [4-6]
CMPSPADB 1CF2	Branch options	Compare operand 1	Compare operand 2	Branch targets
CMPSPADI 18F2	Indicator options	Compare operand 1	Compare operand 2	Indicator targets

Operand 1: Numeric variable scalar, character variable scalar, numeric variable array, character variable array, pointer data object, pointer data object array.

Operand 2: Numeric variable scalar, character variable scalar, numeric variable array, character variable array, pointer data object, pointer data object array.

Operand 3 [4-6]:

- *Branch Form*-Instruction number, relative instruction number, branch point, or instruction pointer.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Description: The space addressability of the object specified by operand 1 is compared with the space addressability of the object specified by operand 2.

Based on the results of the comparison, the resulting condition is used with the extender to transfer control (branch form) or to assign a value to each of the indicator operands (indicator form). If the operands are not in the same space, the resultant condition is *unequal*. If the operands are in the same space and the offset of operand 1 is larger or smaller than the offset of operand 2, the resultant condition is *high* or *low*, respectively. Equal occurs only if the operands are in the same space at the same offset. Therefore, the resultant conditions (*high, low, equal*, and *unequal*) are mutually exclusive. Consequently, if you specify that an action be taken upon the nonexistence of a condition, this results in the action being taken upon the occurrence of any of the other three possible conditions. For example, a branch not high would result in the branch being taken on a *low, equal*, or *unequal* condition.

If a pointer data object operand contains a data pointer value upon execution of the instruction, the addressability is compared to the pointer data object rather than to the scalar described by the data pointer value. The variable scalar references allowed on operands 1 and 2 cannot be described through a data pointer value.

The *object destroyed* (hex 2202) exception, *optimized addressability invalid* (hex 0606) exception, *parameter reference violation* (hex 0801) exception, and *pointer does not exist* (hex 2401) exception are not signaled when operand 1 or operand 2 is based on a space pointer machine object. This occurs when the space pointer machine object contains an internal machine value that indicates one of these error conditions exists. If the corresponding exception is not signaled, the resulting condition of the comparison operation is not defined other than that it will be one of the four valid resultant conditions for this instruction.

Resultant Conditions

- High
- Low
- Equal
- Unequal

Exceptions

Exception		Operands 1	2	3 [4-6]	Other
06	Addressing				
	01 space addressing violation	X	X	X	
	02 boundary alignment	X	X	X	
	03 range	X	X	X	
	06 optimized addressability invalid	X	X	X	
08	Argument/parameter				
	01 parameter reference violation	X	X	X	
10	Damage encountered				
	04 system object damage state	X	X	X	X
	44 partial system object damage	X	X	X	X
1C	Machine-dependent exception				
	03 machine storage limit exceeded				X
20	Machine support				
	02 machine check				X
	03 function check				X
22	Object access				
	01 object not found	X	X	X	
	02 object destroyed	X	X	X	
	03 object suspended	X	X	X	
	08 object compressed				X
24	Pointer specification				
	01 pointer does not exist	X	X	X	
	02 pointer type invalid	X	X	X	
2E	Resource control limit				
	01 user profile storage limit exceeded				X
36	Space management				
	01 space extension/truncation				X

Set Data Pointer (SETDP)

Op Code (Hex)	Operand 1	Operand 2
0096	Receiver	Source

Operand 1: Data pointer.

Operand 2: Numeric variable scalar, character variable scalar, numeric variable array, or character variable array.

Description: A data pointer is created and returned in the storage area specified by operand 1 and has the attributes and space addressability of the object specified by operand 2. Addressability is set to the low-order (leftmost) byte of the object specified by operand 2. The attributes given to the data pointer include scalar type and scalar length.

If operand 2 is a substring compound operand, the length attribute is set equal to the length of the substring. If operand 2 is a subscript compound operand, the attributes and addressability of the single array element specified are assigned to the data pointer. If operand 2 is an array, the attributes and addressability of the first element of the array are assigned to the data pointer. A data pointer can only be set to describe an element of a data array, not a data array in its entirety.

When the addressability in the data pointer is modified, the instruction signals the *space addressing* (hex 0601) exception when one of the following conditions occurs:

- When the space address to be stored in the pointer would have a negative offset value.
- When the offset would address an area beyond the largest space allocatable in the object. This maximum offset value is dependent on the size and packaging of the object containing the space and is independent of the actual size of the space allocated.

If the exception is signaled by this instruction for one of these reasons, the pointer is not modified by the instruction.

Attempts to use a pointer whose offset value lies between the currently allocated extent of the space and the maximum allocatable extent cause the *space addressing* (hex 0601) exception to be signaled.

Exceptions

Exception		Operands 1	Operands 2	Other
06	Addressing			
	01 space addressing violation	X	X	
	02 boundary alignment	X	X	
	03 range	X	X	
	06 optimized addressability invalid	X	X	
08	Argument/parameter			
	01 parameter reference violation	X	X	
10	Damage encountered			
	04 system object damage state	X	X	X
	44 partial system object damage	X	X	X
1C	Machine-dependent exception			
	03 machine storage limit exceeded			X

Exception		Operands 1	Operands 2	Other
20	Machine support			
	02 machine check			X
	03 function check			X
22	Object access			
	01 object not found		X	
	02 object destroyed	X	X	
	03 object suspended	X	X	
	08 object compressed			X
24	Pointer specification			
	01 pointer does not exist	X	X	
	02 pointer type invalid	X	X	
2E	Resource control limit			
	01 user profile storage limit exceeded			X
36	Space management			
	01 space extension/truncation			X

Set Data Pointer Addressability (SETDPADR)

Op Code (Hex)	Operand 1	Operand 2
0046	Receiver	Source

Operand 1: Data pointer.

Operand 2: Numeric variable scalar, character variable scalar, numeric variable array, or character variable array.

Description: The space addressability of the object specified for operand 2 is assigned to the data pointer specified by operand 1. If operand 1 contains a resolved data pointer, the data pointer's scalar attribute component is not changed by the instruction. If operand 1 contains an initialized but unresolved data pointer, the data pointer is resolved in order to establish the scalar attribute component of the pointer. If operand 1 contains other than a resolved data pointer, the instruction creates and returns a data pointer in operand 1 with the addressability of the object specified for operand 2, and the instruction establishes the attributes as a character(1) scalar.

When the addressability is set into a data pointer, the *space addressing* (hex 0601) exception is signaled by the instruction only when the space address to be stored in the pointer has a negative offset value or if the offset addresses beyond the largest space allocatable in the object. This maximum offset value is dependent on the size and packaging of the object containing the space and is independent of the actual size of the space allocated. If the exception is signaled for this reason, the pointer is not modified by the instruction. Attempts to use a pointer whose offset value lies between the currently allocated extent of the space and the maximum allocatable extent of the space cause the *space addressing* (hex 0601) exception to be signaled.

Exceptions

Exception		Operands 1	2	Other
06	Addressing			
	01 space addressing violation	X	X	
	02 boundary alignment	X	X	
	03 range	X	X	
	04 external data object not found	X		
	06 optimized addressability invalid	X	X	
08	Argument/parameter			
	01 parameter reference violation	X	X	
10	Damage encountered			
	04 system object damage state	X	X	X
	44 partial system object damage	X	X	X
1C	Machine-dependent exception			
	03 machine storage limit exceeded			X
20	Machine support			
	02 machine check			X
	03 function check			X
22	Object access			

Exception		Operands 1	Operands 2	Other
	01 object not found	X	X	
	02 object destroyed	X	X	
	03 object suspended	X	X	
	08 object compressed			X
24	Pointer specification			
	01 pointer does not exist	X	X	
	02 pointer type invalid	X	X	
2E	Resource control limit			
	01 user profile storage limit exceeded			X
36	Space management			
	01 space extension/truncation			X

Set Data Pointer Attributes (SETDPAT)

Op Code (Hex)	Operand 1	Operand 2
004A	Receiver	Attributes

Operand 1: Data pointer.

Operand 2: Character(7) scalar.

Description: The value of the character scalar specified by operand 2 is interpreted as an encoded representation of an attribute set that is assigned to the attribute portion of the data pointer specified by operand 1. The addressability portion of the data pointer is not modified. If operand 1 contains an initialized but unresolved data pointer, the data pointer is resolved in order to establish the addressability in the pointer. The attributes specified by the instruction are then assigned to the data pointer. If operand 1 does not contain a data pointer at the initiation of the instruction's execution, an exception is signaled.

The format of the attribute set is as follows:

- Data pointer attributes — Char(7)
  - Scalar type — Char(1)

    Hex 00 = Signed binary
    Hex 01 = Floating-point
    Hex 02 = Zoned decimal
    Hex 03 = Packed decimal
    Hex 04 = Character
    Hex 06 = Onlyns
    Hex 07 = Onlys
    Hex 08 = Either
    Hex 09 = Open
    Hex 0A = Unsigned binary
  - Scalar length — Bin(2)

    If binary or character:
    - Length (only 2 or 4 for binary)

    If floating-point:
    - Length (only 4 or 8 for floating-point)

    If zoned decimal or packed decimal:
    - Fractional digits (F) — Bits 0-7
    - Total digits (T) — Bits 8-15
      (where $1 \leq T \leq 31$, $0 \leq F \leq T$)

    If character:
    - Length (L, where $1 \leq L \leq 32767$)

    If Onlyns:
    - Length (L, where $1 \leq L \leq 16,383$)

      L is the number of double-byte characters.

    If Onlys:
    - Length (L, where $2 \leq L \leq 32,766$)

- L is the number of bytes
- L is even
- L includes any SO and SI characters

If Either:

- Length (L, where $1 \leq L \leq 32,766$)
  - L is the number of bytes
  - L includes any SO and SI characters.

If Open:

- Length (L, where $2 \leq L \leq 32,766$)
  - L is the number of bytes
  - L includes any SO and SI characters.

— Reserved (binary 0) Bin(4)

Support for usage of a Data Pointer describing an Onlyns, Onlys, Either, or Open scalar value is limited to the Copy Extended Characters Left Adjusted With Pad instruction. Usage of such a data pointer defined value on any other instruction is not supported and results in the signaling of the *scalar type invalid* (hex 3201) exception.

Exceptions

Exception		Operands 1	2	Other
06	Addressing			
	01 space addressing violation	X	X	
	02 boundary alignment	X	X	
	03 range	X	X	
	04 external data object not found	X		
	06 optimized addressability invalid	X	X	
08	Argument/parameter			
	01 parameter reference violation	X	X	
10	Damage encountered			
	04 system object damage state	X	X	X
	44 partial system object damage	X	X	X
1C	Machine-dependent exception			
	03 machine storage limit exceeded			X
20	Machine support			
	02 machine check			X
	03 function check			X
22	Object access			
	01 object not found	X	X	
	02 object destroyed	X	X	
	03 object suspended	X	X	
	08 object compressed			X
24	Pointer specification			

Exception		Operands 1	2	Other
	01 pointer does not exist	X	X	
	02 pointer type invalid	X	X	
2E	Resource control limit			
	01 user profile storage limit exceeded			X
32	Scalar specifications			
	01 scalar type invalid		X	
	02 scalar attributes invalid		X	
	03 scalar value invalid		X	
36	Space management			
	01 space extension/truncation			X

Set Space Pointer (SETSPP)

Op Code (Hex)	Operand 1	Operand 2
0082	Receiver	Source

Operand 1: Space pointer.

Operand 2: Numeric variable scalar, character variable scalar, numeric variable array, character variable array, or pointer data object.

Description: A space pointer is returned in operand 1 and is set to address the lowest order (leftmost) byte of the byte string identified by operand 2.

When the addressability is set in a space pointer, the instruction signals the *space addressing* (hex 0601) exception when the offset addresses beyond the largest space allocatable in the object or when the space address to be stored in the pointer has a nonpositive offset value. This offset value is dependent on the size and packaging of the object containing the space and is independent of the actual size of the space allocated. If the exception is signaled for this reason, the pointer is not modified by the instruction. Attempts to use a pointer whose offset value lies between the currently allocated extent of the space and the maximum allocatable extent of the space cause the *space addressing* (hex 0601) exception to be signaled.

If a pointer data object specified for operand 2 contains a data pointer value upon execution of the instruction, the addressability is set to the pointer storage form rather than to the scalar described by the data pointer value. The variable scalar references allowed on operand 2 cannot be described through a data pointer value.

The *object destroyed* (hex 2202) exception, the *optimized addressability invalid* (hex 0606) exception, the *parameter reference violation* (hex 0801) exception, and the *pointer does not exist* (hex 2401) exception are not signaled when operand 1 is a space pointer machine object and operand 2 is based on a space pointer machine object. This occurs when the basing space pointer machine object for operand 2 contains an internal machine value that indicates one of these error conditions exists. If the corresponding exception is not signaled, operand 1 is set with an internal machine value that preserves the exception condition which existed for operand 2. The appropriate exception condition is signaled for either pointer upon a subsequent attempt to reference the space data the pointer addresses.

Exceptions

Exception		Operands 1	2	Other
06	Addressing			
	01 space addressing violation	X	X	
	02 boundary alignment	X	X	
	03 range	X	X	
	06 optimized addressability invalid	X	X	
08	Argument/parameter			
	01 parameter reference violation	X	X	
10	Damage encountered			
	04 system object damage state	X	X	X
	44 partial system object damage	X	X	X
1C	Machine-dependent exception			

Exception		Operands 1	2	Other
	03 machine storage limit exceeded			X
20	Machine support			
	02 machine check			X
	03 function check			X
22	Object access			
	01 object not found	X		
	02 object destroyed	X	X	
	03 object suspended	X	X	
	08 object compressed			X
24	Pointer specification			
	01 pointer does not exist	X	X	
	02 pointer type invalid	X	X	
2E	Resource control limit			
	01 user profile storage limit exceeded			X
32	Scalar specification			
	01 scalar type invalid	X	X	
36	Space management			
	01 space extension/truncation			X

Set Space Pointer with Displacement (SETSPPD)

Op Code (Hex)	Operand 1	Operand 2	Operand 3
0093	Receiver	Source	Displacement

Operand 1: Space pointer.

Operand 2: Numeric variable scalar, character variable scalar, numeric variable array, character variable array, or pointer data object.

Operand 3: Binary scalar.

Description: A space pointer is returned in operand 1 and is set to the space addressability of the lowest (leftmost) byte of the object specified for operand 2 as modified algebraically by an integer displacement specified by operand 3. Operand 3 can have a positive or negative value. I.e.

```
Operand 1 = Address_of(Operand 2) + Operand 3
```

When the addressability is set in a space pointer, the instruction signals the *space addressing* (hex 0601) exception when the space address to be stored in the pointer has a negative offset value or when the offset addresses beyond the largest space allocatable in the object. This maximum offset value is dependent on the size and packaging of the object containing the space and is independent of the actual size of the space allocated. If the exception is signaled for this reason, the pointer is not modified by the instruction. Attempts to use a pointer whose offset value lies between the currently allocated extent of the space and the maximum allocatable extent of the space cause the *space addressing* (hex 0601) exception to be signaled.

If a pointer data object specified for operand 2 contains a data pointer value upon execution of the instruction, the addressability is set to the pointer storage form rather than to the scalar described by the data pointer value. The variable scalar references allowed on operand 2 cannot be described through a data pointer value.

The *object destroyed* (hex 2202) exception, the *optimized addressability invalid* (hex 0606) exception, the *parameter reference violation* (hex 0801) exception, and the *pointer does not exist* (hex 2401) exception are not signaled when operand 1 is a space pointer machine object and operand 2 is based on a space pointer machine object. This occurs when the basing space pointer machine object for operand 2 contains an internal machine value that indicates one of these error conditions exists. If the corresponding exception is not signaled, operand 1 is set with an internal machine value that preserves the exception condition which existed for operand 2. The appropriate exception condition is signaled for either pointer upon a subsequent attempt is made to reference the space data the pointer addresses.

Exceptions

Exception		Operands 1	2	Other
06	Addressing			
	01 space addressing violation	X	X	X
	02 boundary alignment	X	X	X
	03 range	X	X	X
	06 optimized addressability invalid	X	X	X
08	Argument/parameter			

Exception		Operands			
		1	2	Other	
	01 parameter reference violation	X	X	X	
10	Damage encountered				
	04 system object damage state	X	X	X	X
	44 partial system object damage	X	X	X	X
1C	Machine-dependent exception				
	03 machine storage limit exceeded				X
20	Machine support				
	02 machine check				X
	03 function check				X
22	Object access				
	01 object not found	X	X	X	
	02 object destroyed	X	X	X	
	03 object suspended	X	X	X	
	08 object compressed				X
24	Pointer specification				
	01 pointer does not exist	X	X	X	
	02 pointer type invalid	X	X	X	
2E	Resource control limit				
	01 user profile storage limit exceeded			X	
36	Space management				
	01 space extension/truncation			X	

Set Space Pointer Offset (SETSPPO)

Op Code (Hex)	Operand 1	Operand 2
0092	Receiver	Source 1

Operand 1: Space pointer.

Operand 2: Binary scalar.

Description: The value of the binary scalar specified by operand 2 is assigned to the offset portion of the space pointer identified by operand 1. The space pointer continues to address the same space object.

Operand 1 must contain a space pointer; otherwise, an *invalid pointer type* (hex 2402) exception is signaled.

When the addressability in the space pointer is modified, the instruction signals a *space addressing* (hex 0601) exception when one of the following conditions occurs:

- The space address to be stored in the pointer has a negative offset value.
- The offset addresses beyond the largest space allocatable in the object. This maximum offset value is dependent on the size and packaging of the object containing the space and is independent of the actual size of the space allocated.

If the exception is signaled by this instruction for this reason, the pointer is not modified by the instruction.

Attempts to use a pointer whose offset value lies between the currently allocated extent of the space and the maximum allocatable extent cause the *space addressing* (hex 0601) exception to be signaled.

Exceptions

Exception		Operands 1	2	Other
06	Addressing			
	01 space addressing violation	X	X	
	02 boundary alignment	X	X	
	03 range	X	X	
	06 optimized addressability invalid	X	X	
08	Argument/parameter			
	01 parameter reference violation	X	X	
10	Damage encountered			
	04 system object damage state	X	X	X
	44 partial system object damage	X	X	X
1C	Machine-dependent exception			
	03 machine storage limit exceeded			X
20	Machine support			
	02 machine check			X
	03 function check			X

Exception		Operands 1	2	Other
22	Object access			
	01 object not found	X	X	
	02 object destroyed	X	X	
	03 object suspended	X	X	
	08 object compressed			X
24	Pointer specification			
	01 pointer does not exist	X	X	
	02 pointer type invalid	X	X	
2E	Resource control limit			
	01 user profile storage limit exceeded			X
32	Scalar specification			
	01 scalar type invalid	X	X	
36	Space management			
	01 space extension/truncation			X

Store Space Pointer Offset (STSPPO)

Op Code (Hex)	Operand 1	Operand 2
00A2	Receiver	Source

Operand 1: Binary variable scalar.

Operand 2: Space pointer.

Description: The offset value of the space pointer referenced by operand 2 is stored in the binary variable scalar defined by operand 1.

If operand 2 does not contain a space pointer at the initiation of the instruction's execution, an *invalid pointer type* (hex 2401) exception is signaled.

If binary *size* (hex 0C0A) exceptions are to be signaled either because the program creation attribute indicated to do so or because a translator directive indicated to do so, they will be signalled under the following conditions. If the offset value is greater than 32 767 and operand 1 is a signed binary (2) scalar, a *size* (hex 0C0A) exception is signaled. If the offset value is greater than 65 535 and operand 1 is an unsigned binary (2) scalar, a *size* (hex 0C0A) exception is signaled.

Exceptions

Exception		Operands 1	Operands 2	Other
06	Addressing			
	01 space addressing violation	X	X	
	02 boundary alignment	X	X	
	03 range	X	X	
	06 optimized addressability invalid	X	X	
08	Argument/parameter			
	01 parameter reference violation	X	X	
0C	Computations			
	0A size	X		
10	Damage encountered			
	04 system object damage state	X	X	X
	44 partial system object damage	X	X	X
1C	Machine-dependent exception			
	03 machine storage limit exceeded			X
20	Machine support			
	02 machine check			X
	03 function check			X
22	Object access			
	01 object not found	X	X	
	02 object destroyed	X	X	
	03 object suspended	X	X	

Exception		Operands 1	Operands 2	Other
	08 object compressed			X
24	Pointer specification			
	01 pointer does not exist	X	X	
	02 pointer type invalid	X	X	
2E	Resource control limit			
	01 user profile storage limit exceeded			X
36	Space management			
	01 space extension/truncation			X

Subtract Space Pointer Offset (SUBSPP)

Op Code (Hex)	Operand 1	Operand 2	Operand 3
0087	Receiver pointer	Source pointer	Decrement

Operand 1: Space pointer.

Operand 2: Space pointer.

Operand 3: Binary scalar.

Description: The value of the binary scalar specified by operand 3 is subtracted from the space address contained in the space pointer specified by operand 2; the result is stored in the space pointer identified by operand 1. I.e.

```
Operand 1 = Operand 2 - Operand 3
```

Operand 3 can have a positive or negative value. The space object that the pointer is addressing is not changed by the instruction. If operand 2 does not contain a space pointer at the initiation of the instruction's execution, an *invalid pointer type* (hex 2402) exception is signaled.

When the addressability in the space pointer is modified, the instruction signals a *space addressing* (hex 0601) exception when one of the following conditions occurs:

- The space address to be stored in the pointer has a negative offset value.
- The offset addresses beyond the largest space allocatable in the object. This maximum offset value is dependent on the size and packaging of the object containing the space and is independent of the actual size of the space allocated.

If the exception is signaled by this instruction for this reason, the pointer is not modified by the instruction.

Attempts to use a pointer whose offset value lies between the currently allocated extent of the space and the maximum allocatable extent cause the space addressing exception to be signaled.

The *object destroyed* (hex 2202) exception, *optimized addressability invalid* (hex 0606) exception, *parameter reference violation* (hex 0801) exception, and *pointer does not exist* (hex 2401) exception are not signaled when operand 1 and operand 2 are space pointer machine objects. This occurs when operand 2 contains an internal machine value that indicates one of these error conditions exists. If the corresponding exception is not signaled, operand 1 is set with an internal machine value that preserves the exception condition that existed for operand 2. The appropriate exception condition will be signaled for either pointer when a subsequent attempt is made to reference the space data that the pointer addresses.

Exceptions

Exception		Operands 1	2	3	Other
06	Addressing				
	01 space addressing violation	X	X	X	
	02 boundary alignment	X	X	X	
	03 range	X	X	X	
	06 optimized addressability invalid	X	X	X	

Exception		Operands 1	2	3	Other
08	Argument/parameter				
	01 parameter reference violation	X	X	X	
10	Damage encountered				
	04 system object damage state	X	X	X	X
	44 partial system object damage	X	X	X	X
1C	Machine-dependent exception				
	03 machine storage limit exceeded				X
20	Machine support				
	02 machine check				X
	03 function check				X
22	Object access				
	01 object not found	X	X	X	
	02 object destroyed	X	X	X	
	03 object suspended	X	X	X	
	08 object compressed				X
24	Pointer specification				
	01 pointer does not exist	X	X	X	
	02 pointer type invalid	X	X	X	
2E	Resource control limit				
	01 user profile storage limit exceeded	X	X	X	
36	Space management				
	01 space extension/truncation	X	X	X	

Subtract Space Pointers For Offset (SUBSPPFO)

Op Code (Hex)	Operand 1	Operand 2	Operand 3
0033	Offset Difference	Minuend pointer	Subtrahend pointer

Operand 1: Binary(4) variable scalar.

Operand 2: Space pointer.

Operand 3: Space pointer.

Description: The offset portion of the space address contained in the operand 3 space pointer is subtracted from the offset of the space address contained in the space pointer specified by operand 2; the result is stored in the 4 byte binary scalar identified by operand 1. I.e.

```
Operand 1 = Address_of(Operand 2) - Address_of(Operand 3)
```

The offsets for operands 2 and 3 are strictly unsigned values, while the operand 1 result can have a positive or negative value.

No check is made to determine that the space pointers point to the same space. In addition, the existence of the pointers is not checked except for pointers used as a base for the operands. When the space pointers point to different spaces, or one or both of the pointer operands is subject to the pointer does not exist condition, the resulting value is undefined, but no exception is signaled. However, if both operand 2 and operand 3 are subject to the pointer does not exist condition, the result value is zero.

If either operand 2 or operand 3 contains a pointer which is not a space pointer at the initiation of the instruction's execution, an *invalid pointer type* (hex 2402) exception is signaled.

A *size* (hex 0C0A) exception occurs when the operand 1 field is unsigned binary, the resulting value of the subtraction is negative, and the program attribute to *signal size exceptions* is in effect.

The *object destroyed* (hex 2202) exception, *optimized addressability invalid* (hex 0606) exception, *parameter reference violation* (hex 0801) exception, and *pointer does not exist* (hex 2401) exception are not signaled when operand 2 and operand 3 are space pointer machine objects. This occurs when operand 2 or operand 3 contains an internal machine value that indicates one of these error conditions exists. If the corresponding exception is not signaled, operand 1 is undefined, but no exception is signaled.

Exceptions

Exception		Operands 1	2	3	Other
06	Addressing				
	01 space addressing violation	X	X	X	
	02 boundary alignment	X	X	X	
	03 range	X	X	X	
	06 optimized addressability invalid	X	X	X	
08	Argument/parameter				
	01 parameter reference violation	X	X	X	

Exception		Operands 1	2	3	Other
10	Damage encountered				
	04 system object damage state	X	X	X	X
	44 partial system object damage	X	X	X	X
0C	Computation				
	0A Size	X			
1C	Machine-dependent exception				
	03 machine storage limit exceeded				X
20	Machine support				
	02 machine check				X
	03 function check				X
22	Object access				
	01 object not found	X	X	X	
	02 object destroyed	X	X	X	
	03 object suspended	X	X	X	
	08 object compressed				X
24	Pointer specification				
	01 pointer does not exist	X	X	X	
	02 pointer type invalid	X	X	X	
2E	Resource control limit				
	01 user profile storage limit exceeded	X	X	X	
36	Space management				
	01 space extension/truncation	X	X	X	

Chapter 6. Space Management Instructions

This chapter describes the instructions used for space management. These instructions are in alphabetic order. For an alphabetic summary of all the instructions, see Appendix A, "Instruction Summary."

Create Space (CRTS) 6-3
Materialize Space Attributes (MATS) 6-11
Modify Space Attributes (MODS) 6-15

© Copyright IBM Corp. 1991, 1993

Create Space (CRTS)

Op Code (Hex)	Operand 1	Operand 2
0072	Pointer for space addressability	Space creation template

Operand 1: System pointer.

Operand 2: Space pointer.

ILE access

```
CRTS (
  var space_obj         : system pointer;
      creation_template : space pointer
)
```

Description: A space object is created with the attributes that are specified in the space creation template specified by operand 2, and addressability to the created space is placed in a system pointer that is returned in the addressing object specified by operand 1.

Space objects, unlike other types of system objects, are used to contain a space and serve no other purposes.

The template identified by operand 2 must be 16-byte aligned in the space. The following is the format of the space creation template:

- Template size specification — Char(8)*
  - Size of template — Bin(4)*
  - Number of bytes available for materialization — Bin(4)*
- Object identification — Char(32)
  - Object type — Char(1)*
  - Object subtype — Char(1)
  - Object name — Char(30)
- Object creation options — Char(4)
  - Existence attribute — Bit 0

    0 = Temporary
    1 = Permanent
  - Space attribute — Bit 1

    0 = Fixed-length
    1 = Variable-length
  - Initial context — Bit 2

    0 = Addressability is not inserted into context
    1 = Addressability is inserted into context
  - Access group — Bit 3

    0 = Do not create as member of access group
    1 = Create as member of access group
  - Reserved (binary 0) — Bits 4-5

– Public authority specified	Bit 6
0 = No 1 = Yes	
– Initial owner specified	Bit 7
0 = No 1 = Yes	
– Reserved (binary 0)	Bits 8-11
– Set public authority in operand 1	Bit 12
0 = No 1 = Yes	
– Initialize space	Bit 13
0 = Initialize 1 = Do not initialize	
– Automatically extend space	Bit 14
0 = No 1 = Yes	
– Hardware storage protection level	Bits 15-16
00 = Reference and modify allowed for user state programs 01 = Only reference allowed for user state programs 10 = Invalid (yields *template value invalid* (hex 3801) exception) 11 = No reference or modify allowed for user state programs	
– Process temporary space accounting	Bit 17
0 = The temporary space will be tracked to the creating process 1 = The temporary space will not be tracked to the creating process	
– Reserved (binary 0)	Bits 18-31
• Recovery options	Char(4)
– Reserved (binary 0)	Char(2)
– ASP number	Char(2)
• Size of space	Bin(4)
• Initial value of space	Char(1)
• Performance class	Char(4)
– Space alignment	Bit 0

0 = The space associated with the object is allocated to allow proper alignment of pointers at 16-byte alignments within the space. If no space is specified for the object, this value must be specified for the performance class.

1 = The space associated with the object is allocated to allow proper alignment of pointers at 16-byte alignments within the space as well as to allow proper alignment of input/output buffers at 512-byte alignments within the space.

– Clear the space into main memory during creation	Bit 1

0 = Only a minimum amount (up to 4K) of the space will be in main storage upon completion of the instruction.

1 = Most of the space, with some limits enforced by the machine, will be in main storage upon completion of the instruction.

- Reserved (binary 0) — Bits 2-4
- Main storage pool selection — Bit 5

  0 = Process default main storage pool is used for object.
  1 = Machine default main storage pool is used for object.

- Transient storage pool selection — Bit 6

  0 = Default main storage pool (process default or machine default as specified for main storage pool selection) is used for object.
  1 = Transient storage pool is used for object.

- Block transfer on implicit access state modification — Bit 7

  0 = Transfer the minimum storage transfer size for this object. This value is 1 storage unit.
  1 = Transfer the machine default storage transfer size. This value is 8 storage units.

- Unit number — Bits 8-15
- Reserved (binary 0) — Bits 16-31

• Reserved (binary 0) — Char(1)

• Public authority — Char(2)

• Extension offset — Bin(4)

• Context — System pointer

• Access group — System pointer

Note: The instruction ignores the values associated with template entries annotated with an asterisk (*).

A template extension must be specified for the *initial owner specified* creation option. Also, the template extension must be specified (*extension offset* must be nonzero) to specify any of the other template extension fields (those other than the initial owner user profile) as input to the instruction.

The template extension is located by the *extension offset* field. The template extension must be 16-byte aligned in the space. The following is the format of the template extension:

• User profile — System pointer

• Largest size needed for space — Bin(4)

• Domain assigned to the object — Char(2)

  Hex 0000 The domain will be chosen by the machine.

  Hex 0001 The domain will be 'Common User'.

  Hex 8000 The domain will be 'Common System'.

• Reserved (binary 0) — Char(42)

If the created object is permanent, it is owned by the user profile governing process execution. The owning user profile is implicitly assigned all private authority states for the object. The storage occupied by the created object is charged to this owning user profile. If the created object is temporary, there is no owning user profile, and all authority states are assigned as public. Storage occupied by the created context is charged to the creating process.

The **object identification** specifies the symbolic name that identifies the space within the machine. An **object type** of hex 19 is implicitly supplied by the machine. The *object identification* is used to identify the object on materialize instructions as well as to locate the object in a context that addresses the object. The *object subtype* must be hex EF.

The **existence attribute** specifies whether the space is to be created as temporary or permanent. A *temporary* space, if not explicitly destroyed by the user, is implicitly destroyed by the machine when machine processing is terminated. A *permanent* space exists in the machine until it is explicitly destroyed by the user.

The **space attribute** specifies whether the size of the space can vary. The space may have a *fixed* size or a *variable* size. The initial allocation is as specified in the **size of space** field. The machine allocates a space of at least the size specified. The actual size allocated depends on an algorithm defined by a specific implementation. A fixed size space of zero length causes no space to be allocated.

If the **initial context** creation attribute field indicates that *addressability is to be inserted* into a context, the **context** field must contain a system pointer that identifies a context where addressability to the newly created space is to be placed. If *addressability is not to be inserted* into a context, the *context* field is ignored.

If the **access group** creation attribute field indicates that the space is to be created in an access group, the *access group* field must be a system pointer that identifies the access group in which the space is to be created. If the space is being *created as a member of an access group*, the *existence attribute* field must be *temporary* (bit 0 equals 0). If the space *is not to be created into an access group*, the *access group* field is ignored.

The **public authority specified** creation option controls whether or not the space is to be created with the public authority specified in the template. When *yes* is specified, the space is created with the public authority specified in the **public authority** field of the template. When *no* is specified, the *public authority* field is ignored and the space is created with default public authority. The default public authority depends on the value of the *existence attribute*: An *existence attribute* value of *temporary* results in a default public authority of all authority; an *existence attribute* value of *permanent* results in a default public authority of no authority. Refer to the Grant Authority instruction for a description of the public authority field and a list of allowable values.

The **initial owner specified** creation option controls whether or not the initial owner of the space is to be the user profile specified in the template. When *yes* is specified, initial ownership is assigned to the user profile specified in the **user profile** field of the template extension. When *no* is specified, initial ownership is assigned to the process user profile and the *user profile* field in the template extension is ignored. The initial owner user profile is implicitly assigned all authority states for the object. The storage occupied by the created space is charged to the initial owner. If *yes* is specified for this creation option when the *existence attribute* specifies *temporary*, a *template value invalid* (hex 3801) exception will be signaled.

The **set public authority in operand 1** creation option controls, when the *public authority specified* creation option has also been specified as *yes*, whether or not the public authority attribute for the space is to be set into the system pointer returned in operand 1. When *yes* is specified, the specified *public authority* is set into operand 1. When *no* is specified, public authority is not set into operand 1. When the *public authority specified* creation option is set to *no*, this option can not be specified as *yes* (or else a *template value invalid* (hex 3801) exception will be signalled) and the authority set into operand 1 is the default of no authority for a *permanent* or all authority for a *temporary* object (as specified by the *existence attribute*).

The **initialize space** creation option controls whether or not the space is to be initialized. When *initialize* is specified, each byte of the space is initialized to a value specified by the **initial value of space** field. Additionally, when the space is extended in size, this byte value is also used to initialize the new allocation. When *do not initialize* is specified, the *initial value of space* field is ignored and the initial value of the bytes of the space are unpredictable.

When *do not initialize* is specified for a space, internal machine algorithms do ensure that any storage resources last used for allocations to another object which are reused to satisfy allocations for the

space are reset to a machine default value to avoid possible access of data which may have been stored in the other object. To the contrary, reusage of storage areas previously used by the space object are not reset, thereby exposing subsequent reallocations of those storage areas within the space to access of the data which was previously stored within them.

The **automatically extend space** creation option controls whether the space is to be extended automatically by the machine or a *space addressing* (hex 0601) exception is to be signaled when a reference is made to an area beyond the allocated portion of the space. When *yes* is specified, the space will automatically be extended by an amount determined through internal machine algorithms. When *no* is specified, the exception will result. Note that an attempt to reference an area beyond that for the maximum size that a space can be allocated, will always result in the signaling of the *space addressing* (hex 0601) exception independently of the setting of this attribute. The *automatically extend space* creation option can only be specified when the *space attribute* has been specified as *variable length*. Invalid specification of the *automatically extend space* option results in the signaling of the *template value invalid* (hex 3801) exception.

Usage of the *automatically extend space function* is limited. Predictable results will occur only when you ensure that the automatic extension of a space will not happen in conjunction with modification of the space size by another process. That is, you must ensure that when a process is using the space in a manner that could cause it to be automatically extended, it is the sole process which can cause the space size to be modified. Note that in addition to implicit modification through automatic extension, the space size can be explicitly modified through use of the Modify Space Attributes instruction.

The **hardware storage protection level** can be used to restrict access to the contents of the space by user state programs. It is possible to limit the access of the space by user state programs into 1 of three levels:

- *Reference only* (non-modifying storage references are allowed, modifying storing storage references yield a *domain/protection violation* (hex 4401) exception).
- *No storage references* (all storage references, modifying or non-modifying yield a *domain/protection violation* (hex 4401) exception).
- *Full access* (both modifying and non-modifying storage references are allowed).

Process temporary space accounting can be used to detect when temporary space objects, created within a process, still exist at process termination time. Temporary spaces that are created with the *process temporary space accounting* field set to 0 will be "tracked" to the process which created them. Temporary spaces that are created with the *process temporary space accounting* field set to 1 will not be "tracked" to the creating process.

At process termination time, any tracked spaces that exist may cause the machine to attempt to destroy the existing tracked spaces. If this is done, the destroy attempts would be performed as if an MI program issued a Destroy Space (DESS) instruction for each of the existing spaces.

The purpose of *process temporary space accounting* is to identify objects which may be "lost" in the system (until the next IPL). It should not intentionally be used (by MI) as a method of cleaning up temporary space objects at process termination time. The machine does not guarantee that all spaces (that should be tracked) will indeed be tracked. Also, if the machine is attempting to destroy tracked spaces at process termination time, any failures in the deletion attempts (such as if a space is locked to another process) will be ignored (i.e. the space will not be destroyed) and no indication of this is presented to the MI user.

Process temporary space accounting only applies to temporary space objects. A value of 1 for the *process temporary space accounting* field when creating a permanent object will result in a *template value invalid* (hex 3801) exception. This is in spite of the fact that a value of 1 for this field would result in the same actions as when creating a permanent object (i.e. the object would not be tracked to the process). The exception is presented because this field is undefined for permanent objects.

The **ASP number** field specifies the ASP number of the ASP on which the unit is to be allocated. A value of 0 indicates an *ASP number* is not specified and results in the default of allocating the object in the system ASP. Allocation on the system ASP can only be done implicitly by not specifying an ASP number. The only non-zero values allowed are 2 through 16 which specify the user ASP on which the space object will be allocated. The *ASP number* must specify an existing ASP or else an *Auxiliary Storage Pool number invalid* (hex 1C09) exception will be signaled. The *ASP number* field may only be specified for creation of a permanent object. The *ASP number* attribute of an object can be materialized, but cannot be modified.

Invalid specification of the *ASP number* attribute results in the signaling of the *template value invalid* (hex 3801) exception.

The **performance class** fields provide information allowing the machine to more effectively manage the space object considering the overall performance objectives of operations involving the space. The **unit number** field indicates the auxiliary storage unit on which the space should be located, if possible.

The preferred *unit number* field which can be specified in the *performance class* field at object creation may not be specified in conjunction with specification of the ASP number attribute.

The **extension offset** specifies the byte offset from the beginning of the operand 2 template to the beginning of the template extension. An offset value of zero specifies that the template extension is not provided. A negative offset value is invalid. A non-zero offset must be a multiple of 16 (to cause 16 byte alignment of the extension). Except for these restrictions, the offset value is not verified for correctness relative to the location of other portions of the create template.

The **largest size needed for space** field of the template extension specifies, when nonzero, a value in bytes that indicates the largest size that will be needed for the space. This field is different from the *size of space* field which indicates the size for the initial allocation of the space. This field can be used to communicate to the machine what the largest size needed for the space will be. Specification of a value larger than the maximum size space allowed for this object is invalid and results in signaling of the *template value invalid* (hex 3801) exception. Specification of a nonzero value that is less than the *size of space* field also results in the signaling of the *template value invalid* (hex 3801) exception. For more information on the maximum allowed, see the Limitations topic at the end of the instruction's description.

Specifying the *largest size of space* needed value allows the machine, under certain circumstances, to select usage of an internal storage allocation unit which best utilizes the internal addressing resources within the machine. Note that the internal storage allocation unit selected can alter the maximum modification size of the associated space for the object. However, the machine will always use an internal storage allocation unit that will allow for extension of the space to at least the value specified in the the largest size of space needed field. The maximum size to which the space can be modified is dependent upon specific implementations of the machine and can vary with different machine implementations. For more information on the affect of this option, see the Limitations topic at the end of this create instruction definition.

The **domain assigned** field in the template extension allows the user of this instruction to override the domain for this object that would otherwise be chosen by the machine.

Any value specified for the *domain assigned* field other than those listed will result in a *template value invalid* exception (hex 3801) being signalled.

Limitations: The following are limits that apply to the functions performed by this instruction.

The maximum size space allowed for this object under any circumstances is limited to 16MB-256 Bytes. Note that the value specified in largest size of space needed field in the creation template is used by the machine as part of the criteria used in selecting the appropriate internal storage allocation unit to be used for the storage allocations for the space.

Authorization Required

- Insert
  - User profile of object owner
  - Context identified in operand 2
- Retrieve
  - Context referenced for address resolution
- Object control
  - Operand 1 if being replaced

Lock Enforcement

- Materialize
  - Contexts referenced for address resolution
- Modify
  - Context identified in operand 2
  - User profile of object owner
  - Access group identified in operand 2

Exceptions

Exception		Operands 1	2	Other
02	Access group			
	02 object exceeds available space		X	
06	Addressing			
	01 space addressing violation	X	X	
	02 boundary alignment	X	X	
	03 range	X	X	
	06 optimized addressability invalid	X	X	
08	Argument/parameter			
	01 parameter reference violation	X	X	
0A	Authorization			
	01 unauthorized for operation		X	
0E	Context operation			
	01 duplicate object identification		X	
10	Damage encountered			
	04 system object damage state	X	X	X

Exception		Operands 1	2	Other
	05 authority verification terminated due to damaged object			X
	44 partial system object damage	X	X	X
1A	Lock state			
	01 invalid lock state		X	
1C	Machine-dependent exception			
	03 machine storage limit exceeded			X
	04 object storage limit exceeded		X	
	09 auxiliary storage pool number invalid		X	
20	Machine support			
	02 machine check			X
	03 function check			X
22	Object access			
	01 object not found	X	X	
	02 object destroyed	X	X	
	03 object suspended	X	X	
	07 authority verification terminated due to destroyed object			X
	08 object compressed			X
24	Pointer specification			
	01 pointer does not exist	X	X	
	02 pointer type invalid	X	X	
	Pointer addressing invalid object		X	
2E	Resource control limit			
	01 user profile storage limit exceeded		X	
36	Space management			
	01 space extension/truncation			X
38	Template specification			
	01 template value invalid		X	

Materialize Space Attributes (MATS)

Op Code (Hex)	Operand 1	Operand 2
0036	Receiver	Space object

Operand 1: Space pointer.

Operand 2: System pointer.

ILE access

```
MATS (
      receiver     : space pointer;
  var space_object : system pointer
)
```

Description: The current attributes of the space object specified by operand 2 are materialized into the receiver specified by operand 1.

The template identified by operand 1 must be 16-byte aligned in the space. The format of the materialization is as follows:

- Materialization size specification — Char(8)
  - Number of bytes provided for materialization — Bin(4)
  - Number of bytes available for materialization (always 96 for this instruction) — Bin(4)
- Object identification — Char(32)
  - Object type — Char(1)
  - Object subtype — Char(1)
  - Object name — Char(30)
- Object creation options — Char(4)
  - Existence attribute — Bit 0

    0 = Temporary
    1 = Permanent
  - Space attribute — Bit 1

    0 = Fixed-length
    1 = Variable-length
  - Context — Bit 2

    0 = Addressability not in context
    1 = Addressability in context
  - Access group — Bit 3

    0 = Not member of access group
    1 = Member of access group
  - Reserved (binary 0) — Bits 4-12
  - Initialize space — Bit 13

    0 = Initialize

1 = Do not initialize

– Automatically extend space — Bit 14

0 = No
1 = Yes

– Hardware storage protection level — Bits 15-16

00 = Reference and modify allowed for user state programs
01 = Only reference allowed for user state programs
10 = Invalid (undefined)
11 = No reference or modify allowed for user state programs

– Reserved (binary 0) — Bits 17-31

- Reserved (binary 0) — Char(2)
- ASP number — Char(2)
- Size of space — Bin(4)
- Initial value of space — Char(1)
- Performance class — Char(4)

– Space alignment — Bit 0

0 = The space associated with the object is allocated to allow proper alignment of pointers at 16-byte alignments within the space. If no space is specified for the object, this value must be specified for the performance class.
1 = The space associated with the object is allocated to allow proper alignment of pointers at 16-byte alignments within the space as well as to allow proper alignment of input/output buffers at 512-byte alignments within the space.

– Reserved (binary 0) — Bits 1-4

– Main storage pool selection — Bit 5

0 = Process default main storage pool is used for object.
1 = Machine default main storage pool is used for object.

– Transient storage pool selection — Bit 6

0 = Default main storage pool (process default or machine default as specified for main storage pool selection) is used for object.
1 = Transient storage pool is used for object.

– Block transfer on implicit access state modification — Bit 7

0 = Transfer the minimum storage transfer size for this object. This value is 1 storage unit.
1 = Transfer the machine default storage transfer size. This value is 8 storage units.

– Unit number — Bits 8-15

– Reserved (binary 0) — Bits 16-31

- Reserved (binary 0) — Char(7)
- Context — System pointer
- Access group — System pointer

The first 4 bytes that are materialized identify the total **number of bytes provided** for use by the instruction. This value is supplied as input to the instruction and is not modified by the instruction. A value of less than 8 causes a *materialization length* (hex 3803) exception.

The second 4 bytes that are materialized identify the total **number of bytes available** to be materialized. The instruction materializes as many bytes as can be contained in the area specified as the receiver.

If the byte area identified by the receiver is greater than that required to contain the information requested, then the excess bytes are unchanged. No exceptions (other than the materialization length exception described previously) are signaled in the event that the receiver contains insufficient area for the materialization.

See the Create Space instruction for a description of these fields.

This instruction cannot be used to materialize the *public authority specified* creation option, the *initial owner specified* creation option, the *process temporary space accounting* creation option, or the template extension which can be specified on space creation. The Materialize Authority instruction can be used to materialize the current public authority for the space. The Materialize System Object instruction can be used to materialize the current owner of the space.

Authorization Required

- Operational or space authority
  - Operand 2
- Retrieve
  - Contexts referenced for address resolution

Lock Enforcement

- Materialize
  - Operand 2
  - Contexts referenced for address resolution

Exceptions

Exception		Operands 1	2	Other
06	Addressing			
	01 space addressing violation	X	X	
	02 boundary alignment	X	X	
	03 range	X	X	
	06 optimized addressability invalid	X	X	
08	Argument/parameter			
	01 parameter reference violation	X	X	
0A	Authorization			
	01 unauthorized for operation		X	
10	Damage encountered			
	04 system object damage state	X	X	X
	05 authority verification terminated due to damaged object			X
	44 partial system object damage	X	X	X
1A	Lock state			
	01 invalid lock state		X	
1C	Machine-dependent exception			
	03 machine storage limit exceeded			X

Exception		Operands 1	Operands 2	Other
20	Machine support			
	02 machine check			X
	03 function check			X
22	Object access			
	01 object not found	X	X	
	02 object destroyed	X	X	
	03 object suspended	X	X	
	07 authority verification terminated due to destroyed object			X
	08 object compressed			X
24	Pointer specification			
	01 pointer does not exist	X	X	
	02 pointer type invalid	X	X	
	03 pointer addressing invalid object		X	
2E	Resource control limit			
	01 user profile storage limit exceeded			X
36	Space management			
	01 space extension/truncation			X
38	Template specification			
	03 materialization length exception	X		

Modify Space Attributes (MODS)

Op Code (Hex)	Operand 1	Operand 2
0062	System object.	Size or Space modification template.

Operand 1: System pointer.

Operand 2: Binary scalar or character(28) scalar (fixed length).

ILE access

```
MODS1 (
 var space_object : system pointer;
 var size         : signed binary
)

OR

MODS2 (
 var system_object               : system pointer;
 var space_modification_template : aggregate
)
```

Description: The attributes of the space associated with the system object specified for operand 1 are modified with the attribute values specified in operand 2. Operand 1 may address any system object.

The operand 2 space modification template is specified with one of two formats. The abbreviated format, operand 2 specified as a binary scalar, only provides for modifying the *size of space* attribute. The full format, operand 2 specified as a character scalar, provides for modifying the full set of space attributes.

When operand 2 is a binary value, it specifies the size in bytes to which the space size is to be modified. The current allocation of the space is extended or truncated accordingly to match as closely as possible the specified size. The modified space size will be of at least the size specified. The actual size allocated is dependent upon the algorithm used within the specific implementation of the machine. If the space is of fixed size, or if the value of operand 2 is negative, or if the operand 2 size is larger than that for the largest space that can be associated with the object, the space extension/truncation exception is signaled. When operand 2 is a character scalar, it specifies a selection of space attribute values to be used to modify the attributes of the space. It must be at least 28 bytes long and have the following format:

- Modification selection — Char(4)
  - Modify space length attribute — Bit 0

    0 = No
    1 = Yes
  - Modify size of space — Bit 1

    0 = No
    1 = Yes
  - Modify initial value of space — Bit 2

    0 = No
    1 = Yes

- Modify performance class — Bit 3

  0 = No
  1 = Yes

- Modify initialize space attribute — Bit 4

  0 = No
  1 = Yes

- Reinitialize space — Bit 5

  0 = No
  1 = Yes

- Modify automatically extend space attribute — Bit 6

  0 = No
  1 = Yes

| – Create secondary associated space — Bit 7

| 0 = No
| 1 = Yes

| – Reserved (binary 0) — Bits 8-31

- Indicator attributes — Char(4)
  - Reserved (binary 0) — Bit 0
  - Space length — Bit 1

    0 = Fixed length
    1 = Variable length

  - Initialize space — Bit 2

    0 = Initialize
    1 = Do not initialize

  - Automatically extend space — Bit 3

    0 = No
    1 = Yes

| – Reserved (binary 0) — Bits 4-14

| – Hardware storage protection level* — Bits 15-16

| 00 = Reference and modify allowed for user state programs
| 01 = Only reference allowed for user state programs
| 10 = Invalid (undefined)
| 11 = No reference or modify allowed for user state programs

| – Reserved (binary 0) — Bits 17-31

| • Maximum size of space* — Bin(4)

- Size of space — Bin(4)
- Initial value of space — Char(1)
- Performance class — Char(4)

| • Reserved (binary 0) — Char(1)

| • Secondary associated space number — UBin(2)

| • Reserved (binary 0) — Char(4)

| * These fields will be ignored when *create secondary associated space* is 0.

The **modification selection** indicator fields select the modifications to be performed on the space.

The **modify space length attribute** modification selection field controls whether or not the space length attribute is to be modified. When *yes* is specified, the value of the **space length** indicator is used to modify the space to be specified *fixed* or *variable* length attribute. When *no* is specified, the *space length* indicator attribute value is ignored and the *space length* attribute is not modified.

The **modify size of space** modification selection field controls whether or not the allocation size of the space is to be modified. When *yes* is specified, the current allocation of the space is extended or truncated accordingly to match as closely as possible the specified size in the **size of space** field. The modified size will be at least the size specified. The actual size allocated is dependent upon the algorithm used within the specific implementation of the machine. When *no* is specified, the current allocation of the space is not modified and the *size of space* field is ignored.

Modification of the *size of space attribute* for a space of fixed length can only be performed in conjunction with modification of the *space length* attribute. In this case, the *space length* attribute may be modified to the same fixed length attribute or to the variable length attribute. An attempt to modify the *size of space* attribute for a space of fixed length without modification of the *space length* attribute results in the signaling of the *space extension/truncation* (hex 3601) exception. Modification of the *size of space* attribute for a space of variable length can always be performed separately from a modification of the *space length* attribute.

When the *size of space* attribute is to be modified, if the value of the *size of space* field is negative or specifies a size larger than that for the largest space that can be associated with the object, the *space extension/truncation* (hex 3601) exception is signaled.

The **modify initial value of space** modification selection field controls whether or not the *initial value of space* attribute is to be modified. When *yes* is specified, the value of the **initial value of space** field is used to modify the corresponding attribute of this space. This byte value will be used to initialize any new space allocations for this space due to an extension to the size of space attribute on the current execution of this instruction as well as any subsequent modifications. When *no* is specified, the *initial value of space* field is ignored and the *initial value of space* attribute is not modified.

The **modify performance class** modification selection field controls whether or not the *performance class* attribute of the specified system object is to be modified with the values relating to space objects. When *yes* is specified, the value of the **performance class** field is used to modify the corresponding attribute of the specified system object. When *no* is specified, the *performance class* attribute of the specified system object is not modified.

The **modify initialize space attribute** modification selection field controls whether or not the *initialize space attribute* is to be modified. When *yes* is specified, the value of the **initialize space** indicator attribute is used to modify that attribute of the specified space to the specified value. When *no* is specified, the *initialize space* indicator attribute value is ignored and the *initialize space* attribute is not modified.

Changing the value of the *initialize space* attribute only affects whether or not future extensions of the space will be initialized or not. That is, it is the state of this attribute at the time of allocation of the storage for a space that determines whether that newly allocated storage area will be initialized to the initial value specified for the space. Modifications of this attribute subsequent to the allocation of storage to a space have no effect on the value of that previously allocated storage area.

The **reinitialize space** modification selection field controls whether the storage allocated to the space is to be reinitialized in its entirety. When *no* is specified, the space is not reinitialized. When *yes* is specified, the space is reinitialized. This re-initialization is performed after all other attribute modifications which may also have been specified on the instruction have been made. Thus changes to the *size of*

the space, the *initial value of the space*, etc. will be put into effect and be considered the current attributes of the space for purposes of the re-initialization. The byte value used for the re-initialization is either the current initial value for the space if the *initialize space* attribute for the space currently specifies *yes*, or a value of hex 00 if the *initialize space* attribute currently specifies *no*.

Note that specifying *yes* for the *reinitialize space* modification selection field for a space with current attributes of fixed length size zero results in no operation, because such a space has no allocated storage to reinitialize. Also, note that re-initialization of a space will have the side effect of resetting partial damage for a space object containing the space if the space object had previously been marked as having partial damage. This only applies to space objects; i.e. re-initialization of an associated space does not have the side effect of resetting partial damage for the MI object containing it.

The **modify automatically extend space** attribute modification selection field controls whether or not the *automatically extend space* attribute is to be modified. When *yes* is specified, the value of the **automatically extend space** indicator attribute is used to modify that attribute of the specified space to the specified value. When *no* is specified, the *autmatically extend space* indicator attribute value is ignored and the *automatically extend space* attribute is not modified. The *automatically extend space* attribute can only be specified as *yes* when the *space length* attribute for the space is already *variable* length, or when the *space length* attribute is being modified to *variable* length. Invalid specification of the *automatically extend space* attribute results in the signaling of the *invalid space modification* (hex 3602) exception.

Modification to or from the state of a space being fixed length of size zero can not be performed for the following objects:

Class of service description

Controller description

Cursor

Data space

Logical unit description

Mode description

Network description

Space

Program (when attempted while in user state on a security level 40 or higher system).

If such a modification is attempted for these objects, the *invalid space modification* (hex 3602) exception is signaled.

Specifying *yes* for the **modify performance class** modification selection field is only allowed when the space to be modified is a fixed length space of size zero. This modification may be specified in conjunction with other modifications. Only bit 0 of the *performance class* field is used to modify the performance class attribute of the specified system object. A bit value of zero requests that the start of the space storage provide 16-byte multiple (pointer) machine address alignment. A bit value of one requests that the start of the space storage provide 512-byte multiple (buffer) machine address alignment. Bits 1 through 31 are ignored. Specifying *yes* for the *modify performance class* modification selection field when the space to be modified is not a fixed length space of size zero results in the signaling of the *invalid space modification* (hex 3602) exception.

The **create secondary associated space** field indicates if a secondary associated space is to be created for the object. All the indicator attributes and size and value fields are used for this create. The secondary associated space to be created is indicated by the *secondary associated space number* field. If the specified space already exists, the *invalid space modification* (hex 3602) exception is signalled. When this bit is set to *yes*, all other *modification selection* bits are ignored.

Specifying the **largest size of space needed** value allows the machine. under certain circumstances, to select usage of an internal storage allocation unit which best utilizes the internal addressing resources within the machine. Note that the internal storage allocation unit selected can alter the maximum modification size of the associated space for the object. However, the machine will always use an internal storage allocation unit that will allow for extension of the space to at least the value specified in the the largest size of space needed field. The maximum size to which the space can be modified is dependent upon specific implementations of the machine and can vary with different machine implementations.

The **secondary associated space number** field is used to indicate which secondary space is to be created or modified. When this field is zero, the primary associated space of the space object is modified. If this field is not zero and no secondary associated spaces are allowed for the object, the *scalar value invalid* (hex 3203) exception will be signalled.

A fixed length space of size zero is defined by the machine to have no internal storage allocation. Due to this, a modification to or from this state is, in essence, the same as a destroy or create for the space associated with the specified system object. The effect of modifying to this state is similar to destroying the associated space in that address references to the space through previously set pointers will result in signaling of the object destroyed exception. When a primary associated space is destroyed by using this method, any secondary associated spaces for the object are also destroyed. Additionally, an attempt to set a space pointer to the space associated with the specified system object through the Set Space Pointer from Pointer instruction will result in the signaling of the *invalid space reference* (hex 0605) exception. To the contrary, modifying the space attributes from this state is similar to creating an associated space in that the Set Space Pointer from Pointer instruction can be used to set a space pointer to the start of a storage within the associated space and the allocated space storage can be used to contain space data.

The extension and truncation of a space is always by an implementation-defined multiple of 256 bytes. This means that if, for example, the implementation defined multiple is 2 (or 512 bytes), any modification of the space size will be in increments of 512 bytes.

Authorization Required

- Object management
  - Operand 1
- Retrieve
  - Contexts referenced for address resolution

Lock Enforcement

- Materialize
  - Contexts referenced for address resolution
- Object control
  - Operand 1 (when operand 2 is binary)
- Modify
  - Operand 1 (when operand 2 is character)

Events

0002 Authorization
    0101 Object authorization violation

000C Machine resource

- 0201 Machine auxiliary storage threshold exceeded
- 0202 User auxiliary storage pool threshold exceeded
- 0203 User auxiliary storage pool overflow
- 0204 Asp unprotected space overflow
- 0501 Machine address threshold exceeded

000D Machine status
- 0101 Machine check
- 0401 Auxiliary storage device requires service

0010 Process
- 0701 Maximum processor time exceeded
- 0801 Process storage limit exceeded

0016 Machine observation
- 0101 Instruction reference

0017 Damage set
- 0401 System object damage set
- 0801 Partial system object damage set

Exceptions

Exception		Operands 1	2	Other
06	Addressing			
	01 space addressing violation	X	X	
	02 boundary alignment	X	X	
	03 range	X	X	
	05 invalid space reference			X
	06 optimized addressability invalid	X	X	
08	Argument/parameter			
	01 parameter reference violation	X	X	
0A	Authorization			
	01 unauthorized for operation	X		
10	Damage encountered			
	04 system object damage state	X	X	X
	05 authority verification terminated due to damaged object			X
	44 partial system object damage	X	X	X
1A	Lock state			
	01 invalid lock state	X		
1C	Machine-dependent exception			
	03 machine storage limit exceeded			X
	04 object storage limit exceeded	X		
20	Machine support			
	02 machine check			X
	03 function check			X

Exception		Operands 1	2	Other
22	Object access			
	01 object not found	X	X	
	02 object destroyed	X	X	
	03 object suspended	X	X	
	07 authority verification terminated due to destroyed object			X
	08 object compressed			X
24	Pointer specification			
	01 pointer does not exist	X	X	
	02 pointer type invalid	X	X	
	03 pointer addressing invalid object	X		
2A	Program creation			
	06 invalid operand type	X	X	
	07 invalid operand attribute	X	X	
	08 invalid operand value range	X	X	
	0A invalid operand length		X	
	0C invalid operand odt reference	X	X	
	0D reserved bits are not zero	X	X	X
2E	Resource control limit			
	01 user profile storage limit exceeded	X		
32	Scalar Specification			
	03 scalar value invalid		X	
36	Space management			
	01 space extension/truncation	X	X	
	02 invalid space modification	X	X	

Chapter 7. Heap Management Instructions

This chapter describes the instructions used for heap management. These instructions are in alphabetic order. For an alphabetic summary of all the instructions, see Appendix A, "Instruction Summary."

Allocate Heap Space Storage (ALCHSS) . . . 7-3
Create Heap Space (CRTHS) . . . 7-6
Destroy Heap Space (DESHS) . . . 7-11
Free Heap Space Storage (FREHSS) . . . 7-13
Free Heap Space Storage From Mark (FREHSSMK) . . . 7-15
Materialize Heap Space Attributes (MATHSAT) . . . 7-17
Reallocate Heap Space Storage (REALCHSS) . . . 7-22
Set Heap Space Storage Mark (SETHSSMK) . . . 7-25

© Copyright IBM Corp. 1991, 1993

Allocate Heap Space Storage (ALCHSS)

Op Code (Hex)	Operand 1	Operand 2	Operand 3
03B3	Space allocation	Heap Identifier	Size of space allocation

Operand 1: Space pointer.

Operand 2: Binary(4) scalar or null.

Operand 3: Binary(4) scalar.

ILE access

```
ALCHSS (
  heap_identifier          : signed binary;
  size_of_space_allocation : signed binary
) : space pointer /* space_allocation */
```

Description: A heap space storage allocation of at least the size indicated by operand 3 is provided from the heap space indicated by operand 2. The operand 1 space pointer is set to address the first byte of the allocation which will begin on a boundary at least as great as the minimum boundary specified when the heap space was created.

Each allocation associated with a heap space provides a continuum of contiguously addressable bytes. Individual allocations within a heap space have no addressability affinity with each other. The contents of the heap space allocation are unpredictable unless initialization of heap allocations was specified when the heap space was created.

Simultaneous operations against a heap space from multiple processes is not supported and may produce unpredictable results.

The maximum single allocation allowed is determined by the maximum single allocation size specified when the heap space was created. The maximum single allocation possible is 16M-64K bytes.

It is the responsibility of the using program to see that only the amount of heap space storage requested is used. Reference to heap space storage outside the bounds of the requested space will produce unpredictable results. The exact address returned must be supplied to the Free Heap Space Space Storage (FREHSS) instruction when the user has completed use of the heap space storage.

A default heap space (heap identifier value of 0) is automatically available in each activation group without issuing a Create Heap Space (CRTHS) instruction. The default heap space is created when the first Allocate Heap Space is issued against the default heap space. When operand 2 is null, the default heap space (heap identifier of 0) provides the allocation.

The machine supplied attributes of the default heap space are as follows:

- Maximum single allocation size is 16M - 64K bytes.
- Minimum boundary requirement is a 16 byte boundary.
- The creation size advisory is 4KB unless the size of the allocation request dictates a larger creation size be used.
- The extension size advisory is 4KB unless the size of the allocation request dictates a larger extension size be used.

- Domain is determined from the state of the program issuing the instruction.
- Normal allocation strategy.
- A heap space mark is not allowed.
- The transfer size is 1 storage unit.
- The process access group membership advisory value is taken from the activation group. Default activation groups get this value from the attribute specified on the Initiate Process (INITPR) instruction. Other activation groups get this value from the attribute specified on the Create Bound Program (CRTBPGM) instruction for the program which initiated the activation group.
- Heap space storage allocations are not initialized to the allocation value.
- Heap space storage allocations are not overwritten to the freed value after being freed.

Neither operand 2 nor 3 is modified by the instruction.

Events

000C Machine resource
- 0201 Machine auxiliary storage threshold exceeded
- 0501 Machine address threshold exceeded

000D Machine status
- 0101 Machine check
- 0401 Auxiliary storage device requires service

0010 Process
- 0701 Maximum processor time exceeded
- 0801 Process storage limit exceeded

0016 Machine observation
- 0101 Instruction reference
- 0401 Domain violation
- 0402 Hardware storage violation

0017 Damage set
- 0401 System object damage set
- 0801 Partial system object damage set

Exceptions

Exception		Operands 1	2	3	Other
06	Addressing				
	01 space addressing violation	X	X	X	
	02 boundary alignment	X	X	X	
	03 range	X	X	X	
	06 optimized addressability invalid	X	X	X	
08	Argument/parameter				
	01 parameter reference violation	X	X	X	X
10	Damage encountered				

Exception		Operands 1	2	3	Other
	04 system object damage state	X	X	X	X
	05 authority verification terminated due to damaged object				X
	44 partial system object damage	X	X	X	X
1C	Machine-dependent exception				
	03 machine storage limit exceeded				X
20	Machine support				
	02 machine check				X
	03 function check				X
24	Pointer specification				
	01 pointer does not exist	X	X	X	X
	02 pointer type invalid	X	X	X	X
	03 Pointer addressing invalid object		X	X	
44	Domain specification				
	01 Domain Violation	X	X	X	
45	Heap specification				
	01 Invalid Heap Identifier		X		
	03 Heap Space Full		X		
	04 Invalid Size Request			X	
	05 Heap Space Destroyed				X
	06 Invalid Heap Condition	X	X	X	X

Create Heap Space (CRTHS)

Op Code (Hex)	Operand 1	Operand 2
03B2	Heap Identifier	Heap Space creation template

Operand 1: Binary(4) variable scalar

Operand 2: Space pointer

ILE access

```
CRTHS (
  var heap_identifier   : signed binary;
      creation_template : space pointer
)
```

Description: A heap space is created with the attributes supplied in the heap space creation template specified by operand 2. The heap space identifier used to perform allocations and marks against the heap space is returned in operand 1.

The heap identifier returned in operand 1 represents the heap space. This identifier is used for the Allocate Heap Space Storage (ALCHSS), Destroy Heap Space (DESHS), Set Heap Space Storage Mark (SETHSSMK) and Materialize Heap Space Attributes (MATHSAT) instructions.

The heap space creation template identified by operand 2 must be 16-byte aligned in the space. Operand 2 is not modified by the instruction.

The following is the format of the heap space creation template:

- Reserved (binary 0) — Char(8)
- Maximum single allocation — UBin(4)
- Minimum boundary requirement — UBin(4)
- Creation size — UBin(4)
- Extension size — UBin(4)
- Domain/Storage protection — Bin(2)

  Hex 0000 = System should chose the Domain
  Hex 0001 = The heap space domain should be "Common User"
  Hex 8000 = The heap space domain should be "Common System"

- Heap space creation options — Char(6)
  - Allocation strategy — Bit 0

    0 = Normal allocation strategy
    1 = Force process space creation on each allocate

  - Heap Space Mark — Bit 1

    0 = Allow heap space mark
    1 = Prevent heap space mark

  - Block transfer — Bit 2

    0 = Transfer the minimum storage transfer size.
    This value is 1 storage unit.

1 = Transfer the machine default storage transfer size. This value is 8 storage units.

– Process access group member	Bit 3
0 = Do not create the heap space in the PAG. 1 = Create the heap space in the PAG.	
– Allocation initialization	Bit 4
0 = Do not initialize allocations. 1 = Initialize allocations.	
– Overwrite freed allocations	Bit 5
0 = Do not overwrite freed allocations. 1 = Overwrite freed allocations.	
– Reserved (binary 0)	Bits 6-7
– Allocation value	Char(1)
– Freed value	Char(1)
– Reserved (binary 0)	Char(3)
• Reserved (binary 0)	Char(64)

The **maximum single allocation** of any single allocation from the heap space is useful for controlling the use of the heap space and may also improve performance for some cases when the machine can optimize access based on this attribute. The minimum value that can be specified is 4 bytes, and the maximum value that can be specified is 16M-64K bytes. If zero is specified, the default value of 16M-64K bytes will be used. Values outside the range indicated will cause a *template value invalid* (hex 3801) exception.

The **minimum boundary alignment** associated with any allocation from the heap space can be specified in the template. The minimum value that can be specified is a 4 byte boundary alignment. The maximum boundary alignment that can be specified is a 512 byte boundary alignment. If zero is specified, a default of a 16 byte boundary will be used. The *minimum boundary alignment* will be rounded up to a power of 2. Values outside the range indicated will cause a *template value invalid* (hex 3801) exception.

The **creation size** of the heap space can be specified in the template. If zero is specified the system will compute a default value. The minimum value that can be specified is 512 bytes. The maximum value that can be specified is 16M-1K bytes. The value specified will be rounded up to a storage unit boundary. Values outside the range indicated will cause a *template value invalid* (hex 3801) exception. This is an advisory value only. The machine may decide to override the value specified based on system resource constraints.

The **extension size** of the heap space can be specified in the template. If zero is specified the system will compute a default value. The minimum value that can be specified is 512 bytes. The maximum value that can be specified is 16M-1K bytes. The value specified will be rounded up to a storage unit boundary. Values outside the range indicated will cause a *template value invalid* (hex 3801) exception. This is an advisory value only. The machine may decide to override the value specified based on system resource constraints.

The **domain/storage protection** field in the template allows the user of this instruction to override the domain for the heap space that would otherwise be chosen by the machine. The *domain/storage protection* attribute can be used to restrict access to the contents of the heap space by user state programs. It is possible to limit the access of the heap space by user state programs into 1 of two levels:

- No storage references (all storage references, modifying or non-modifying yield a *domain/protection violation* (hex 4401) exception). This is *Common System*.
- Full access (both modifying and non-modifying storage references are allowed). This is *Common User*.

Only a system state process can specify a heap space to be created with a domain of *Common System*. If a user state program attempts to specify the *domain/storage protection* as *Common System*, a *template value invalid* (hex 3801) exception will be signaled. Any value other than the ones listed will cause a *template value invalid* (hex 3801) exception to be signaled.

The **normal allocation strategy** as defined by the machine will be used unless the force process space creation on each allocation attribute is indicated. This option should only be used in unusual situations, such as when necessary for debug of application problems caused by references outside an allocation.

The **heap space mark** attribute can be used to prevent the use of the Set Heap Space Storage Mark (SETHSSMK) and Free Heap Space Storage from Mark (FREHSSMK) instructions on a heap space.

Block transfer on a heap space is used to increase the performance of a heap space based on prior knowledge of the program creating the heap space on how that heap space will be used. This attribute is used only when the heap space is not a member of a process access group (PAG).

A heap space can be created as a process access group (PAG) member of the currently executing process, if specified by the **prcoess access group member** It is possible for the PAG to overflow at which point any requested heap space creations or extensions will not reside in the PAG. Thus the specification to have the heap space as a member of the PAG is only an advisory which the machine may decide to override.

The **allocation initialization** field in the template allows the user of this instruction to specify that all storage allocations from the heap space being created will be *initialized* to the **allocation value** supplied in the template. If the user chooses *not to initialize* heap space storage allocations, the initial value of heap space storage allocations is unpredictable but will not expose data produced by a different user profile.

The **overwrite freed allocations** field in the template allows the user of this instruction to specify that all heap space storage allocations upon being freed *will be overwritten* with the the **freed value** supplied in the template. If the user chooses *not to overwrite* heap space storage allocations when freed, the contents of the freed allocations will be unaltered.

A default heap space (heap identifier value of 0) is automatically available in each activation group, without issuing a Create Heap Space (CRTHS) instruction. The default heap space is created on the first allocation request of the default heap space. See Allocate Heap Space Storage (ALCHSS) for a description of the default heap space.

A heap space is scoped to an activation group, thus the maximum life of a heap space is the life of the activation group in which the heap space was created. A heap space can only be destroyed from within the activation group in which it was created.

Simultaneous operations against a heap space from multiple processes is not supported and may produce unpredictable results.

Limitations: The following are limits that apply to the functions performed by this instruction.

The amount of heap space storage that can be allocated for a single heap space is 4G-512K bytes. Due to fragmentation a heap space may grow to 4GB-512KB without having 4GB-512KB of outstanding heap space storage allocations.

The maximum allocation size of any single allocation from the heap space is limited to 16MB-64K bytes. A smaller maximum allocation size can be specified.

Events

000C Machine resource
- 0201 Machine auxiliary storage threshold exceeded
- 0501 Machine address threshold exceeded

000D Machine status
- 0101 Machine check
- 0401 Auxiliary storage device requires service

0010 Process
- 0701 Maximum processor time exceeded
- 0801 Process storage limit exceeded

0016 Machine observation
- 0101 Instruction reference
- 0401 Domain violation
- 0402 Hardware storage violation

0017 Damage set
- 0401 System object damage set
- 0801 Partial system object damage set

Exceptions

Exception		Operands 1	Operands 2	Other
06	Addressing			
	01 space addressing violation	X	X	
	02 boundary alignment	X	X	
	03 range	X	X	
	06 optimized addressability invalid	X	X	
08	Argument/parameter			
	01 parameter reference violation	X	X	
10	Damage encountered			
	04 system object damage state	X	X	X
	05 authority verification terminated due to damaged object			X
	44 partial system object damage	X	X	X
1C	Machine-dependent exception			
	03 machine storage limit exceeded			X
	04 object storage limit exceeded			X

Exception		Operands 1	Operands 2	Other
20	Machine support			
	02 machine check			X
	03 function check			X
24	Pointer specification			
	01 pointer does not exist	X	X	
	02 pointer type invalid	X	X	
	03 Pointer addressing invalid object		X	
38	Template specification			
	01 template value invalid		X	
44	Domain specification			
	01 Domain Violation	X	X	

Destroy Heap Space (DESHS)

Op Code (Hex)	Operand 1
03B1	Heap Identifier

Operand 1: Binary(4) variable scalar.

ILE access

```
DESHS (
  var heap_identifier : signed binary
)
```

Description: This instruction destroys and removes from the current activation group the heap space specified by the heap identifier in operand 1. Subsequent use of this heap identifier within the activation group will result in an *invalid identifier* (hex 4501) exception. The heap identifier was returned on the Create Heap Space (CRTHS) instruction. An attempt to destroy the default heap space (heap identifier value of 0) will result in an *invalid request* (hex 4502) exception.

Space pointer references to heap space allocations from a destroyed heap space will cause unpredictable results.

All heap spaces are implicitly destroyed when the activation group in which they were created is destroyed.

Operand 1 is not modified by the instruction.

Events

000C Machine resource
- 0201 Machine auxiliary storage threshold exceeded
- 0501 Machine address threshold exceeded

000D Machine status
- 0101 Machine check
- 0401 Auxiliary storage device requires service

0010 Process
- 0701 Maximum processor time exceeded
- 0801 Process storage limit exceeded

0016 Machine observation
- 0101 Instruction reference
- 0401 Domain violation
- 0402 Hardware storage violation

0017 Damage set
- 0401 System object damage set
- 0801 Partial system object damage set

Exceptions

Exception	Operands 1	Other

Exception		Operands 1	Other
06	Addressing		
	01 space addressing violation	X	
	02 boundary alignment	X	
	03 range	X	
	06 optimized addressability invalid	X	
08	Argument/parameter		
	01 parameter reference violation	X	
10	Damage encountered		
	04 system object damage state	X	X
	05 authority verification terminated due to damaged object		X
	44 partial system object damage	X	X
1C	Machine-dependent exception		
	03 machine storage limit exceeded		X
20	Machine support		
	02 machine check		X
	03 function check		X
24	Pointer specification		
	01 pointer does not exist	X	
	02 pointer type invalid	X	
44	Domain specification		
	01 Domain Violation	X	
45	Heap specification		
	01 Invalid Heap Identifier	X	
	02 Invalid Request	X	
	05 Heap Space Destroyed		X
	06 Invalid Heap Condition	X	X

Free Heap Space Storage (FREHSS)

Op Code (Hex)	Operand 1
03B5	Space allocation

Operand 1: Space pointer.

ILE access

```
FREHSS (
        space_allocation : space pointer
)
```

Description: The heap space storage allocation beginning at the byte addressed by operand 1 is de-allocated from the heap space which supplied the allocation. De-allocation makes the storage available for reuse by subsequent Allocate Heap Space Storage (ALCHSS) instructions. The entire space allocation is de-allocated; partial de-allocation is not supported. A free of heap space storage can be performed without regard to the activation group in which it was allocated, as long as the allocation was done by the same process.

Operand 1 must be exactly equal to the space pointer that was returned by some previous Allocate Heap Space Storage (ALCHSS) or Reallocate Heap Space Storage (REALCHSS) instruction. If it is not, an *invalid request* (hex 4502) exception will be signaled.

Subsequent references to space allocations which have been freed cause unpredictable results.

FREHSS will signal a *domain violation* (hex 4401) exception if a program running user state attempts to de-allocate heap space storage in a heap space with a domain of Common System.

Operand 1 is not modified by the instruction.

Events

000C Machine resource
- 0201 Machine auxiliary storage threshold exceeded
- 0501 Machine address threshold exceeded

000D Machine status
- 0101 Machine check
- 0401 Auxiliary storage device requires service

0010 Process
- 0701 Maximum processor time exceeded
- 0801 Process storage limit exceeded

0016 Machine observation
- 0101 Instruction reference
- 0401 Domain violation
- 0402 Hardware storage violation

0017 Damage set
- 0401 System object damage set
- 0801 Partial system object damage set

Exceptions

Exception		Operands 1	Other
06	Addressing		
	01 space addressing violation	X	
	02 boundary alignment	X	
	03 range	X	
	06 optimized addressability invalid	X	
08	Argument/parameter		
	01 parameter reference violation	X	
10	Damage encountered		
	04 system object damage state	X	X
	05 authority verification terminated due to damaged object		X
	44 partial system object damage	X	X
1C	Machine-dependent exception		
	03 machine storage limit exceeded		X
20	Machine support		
	02 machine check		X
	03 function check		X
24	Pointer specification		
	01 pointer does not exist	X	
	02 pointer type invalid	X	
	03 Pointer addressing invalid object		
44	Domain specification		
	01 Domain Violation	X	
45	Heap specification		
	02 Invalid Request	X	
	05 Heap Space Destroyed		X
	06 Invalid Heap Condition	X	X

Free Heap Space Storage From Mark (FREHSSMK)

Op Code (Hex)	Operand 1
03B9	Mark Identifier

Operand 1: Space pointer data object.

ILE access

```
FREHSSMK (
  var mark_identifier : space pointer
)
```

Description: All heap space allocations which have occurred from the heap space since it was marked, with the mark identifier supplied in operand 1, are freed. This may include heap space storage marked by intervening Set Heap Space Storage Mark (SETHSSMK) instructions. The mark identifier specified in operand 1 and all mark identifiers obtained since the heap space was marked by operand 1 are cleared from the heap space. An attempt to free heap space storage from a mark that has already been cleared by a previous FREHSSMK instruction will result in an *invalid identifier* (hex 4507) exception. A free of heap space storage can be performed without regard to the activation group in which it was allocated, as long as the allocation was done by the same process.

FREHSSMK will signal a *domain violation* (hex 4401) exception if a program running user state attempts a free heap space storage from mark for a heap space with a *domain* of *Common System.*

Operand 1 is not modified by the instruction.

Events

000C Machine resource
- 0201 Machine auxiliary storage threshold exceeded
- 0501 Machine address threshold exceeded

000D Machine status
- 0101 Machine check
- 0401 Auxiliary storage device requires service

0010 Process
- 0701 Maximum processor time exceeded
- 0801 Process storage limit exceeded

0016 Machine observation
- 0101 Instruction reference
- 0401 Domain violation
- 0402 Hardware storage violation

0017 Damage set
- 0401 System object damage set
- 0801 Partial system object damage set

Exceptions

Exception	Operands 1	Other

Exception		Operands 1	Other
06	Addressing		
	01 space addressing violation	X	
	02 boundary alignment	X	
	03 range	X	
	06 optimized addressability invalid	X	
08	Argument/parameter		
	01 parameter reference violation	X	
10	Damage encountered		
	04 system object damage state	X	X
	05 authority verification terminated due to damaged object		X
	44 partial system object damage	X	X
1C	Machine-dependent exception		
	03 machine storage limit exceeded		X
	04 object storage limit exceeded		
	09 auxiliary storage pool number invalid		
20	Machine support		
	02 machine check		X
	03 function check		X
24	Pointer specification		
	01 pointer does not exist	X	
	02 pointer type invalid	X	
	03 Pointer addressing invalid object		
44	Domain specification		
	01 Domain Violation	X	
45	Heap specification		
	02 Invalid Request	X	
	05 Heap Space Destroyed		X
	06 Invalid Heap Condition	X	X
	07 Invalid Mark Identifier	X	

Materialize Heap Space Attributes (MATHSAT)

Op Code (Hex)	Operand 1	Operand 2	Operand 3
03B7	Materialize template	Heap identifier template	Attribute Selection

Operand 1: Space pointer.

Operand 2: Space pointer.

Operand 3: Character(1) scalar (fixed length).

ILE access

```
MATHSAT (
      materialize_template      : space pointer
      heap_identifier_template  : space pointer
  var attribute_selection       : aggregate
```

Description: This instruction will materialize the information selected by operand 3 for the heap space specified by operand 2 and return the selected information in the template indicated by operand 1.

Operand 3 can have three possible values:

- Hex 00 - Return heap space attributes
- Hex 01 - Return heap space attributes and mark information.
- Hex 02 - Return heap space attributes, mark information and allocation information.

Any value for operand 3 other than those listed will cause a *scalar value invalid* (3203) exception.

The heap space attributes template identified by operand 1 must be 16-byte aligned in the space.

If operand 3 is equal to hex 00, then only the **heap space attributes** template information is returned. The format of the attributes template information is as follows (see the Create Heap Space instruction for a description of these fields):

- Template size specification — Char(8)
  - Number of bytes provided for materialization — Bin(4)
  - Number of bytes available for materialization — Bin(4)
- Maximum single allocation — UBin(4)
- Minimum boundary requirement — UBin(4)
- Creation size — UBin(4)
- Extension size — UBin(4)
- Domain — Bin(2)

  Hex 0001 = The heap space domain is "Common User"
  Hex 8000 = The heap space domain is "Common System"
- Heap space creation options — Char(6)
  - Allocation strategy — Bit 0

    0 = Normal allocation strategy
    1 = Force implicit space creation on each allocate

— Heap Space Mark 0 = Allow heap space mark 1 = Prevent heap space mark	Bit 1
— Block transfer 0 = Transfer the minimum storage transfer size. This value is 1 storage unit. 1 = Transfer the machine default storage transfer size. This value is 8 storage units.	Bit 2
— Process access group member 0 = Do not create the heap space in the PAG. 1 = Create the heap space in the PAG.	Bit 3
— Initialization allocations 0 = Do not initialize allocations. 1 = Initialize allocations.	Bit 4
— Overwrite freed allocations 0 = Do not overwrite freed allocations. 1 = Overwrite freed allocations.	Bit 5
— Reserved (binary 0)	Bits 6-7
— Allocation value	Char(1)
— Freed value	Char(1)
— Reserved (binary 0)	Char(3)
• Reserved (binary 0)	Char(64)
• Current number of outstanding allocations	UBin(4)
• Total number of reallocations	UBin(4)
• Total number of frees	UBin(4)
• Total number of allocations	UBin(4)
• Maximum number of outstanding allocations	UBin(4)
• Size of the heap space in pages	UBin(4)
• Number of outstanding marks	UBin(4)
• Total number of extensions	UBin(4)

The first 4 bytes that are materialized identify the total **number of bytes provided** for use by the instruction. This value is supplied as input to the instruction and is not modified by the instruction. A value of less than 8 causes a *materialization length* (hex 3803) exception.

The second 4 bytes that are materialized identify the total **number of bytes available** to be materialized. The instruction materializes as many bytes as can be contained in the area specified as the receiver. If the byte area identified by the receiver is greater than that required to contain the information requested, then the excess bytes are unchanged. No exceptions (other than the materialization length exception described previously) are signaled in the event that the receiver contains insufficient area for the materialization.

If operand 3 is equal to hex 01, then the *mark template information* is added to the *heap space attributes* template information. The *mark template information* is repeated for the number of outstanding marks. This information follows the *heap space attributes* template information. The format of the **mark template information** is as follows:

- Mark template information
  - Mark identifier — Space pointer

Given the list of mark identifiers with a mark identifier being entry N and a allocation belonging to mark identifier N, that allocation also belongs to mark identifier N-X, where X has values 1 to N-1 for all N>1.

If operand 3 is equal to hex 02, then the **allocation template** information is added to the *heap space attributes* and *mark template information*. The *allocation template information* is repeated for current number of outstanding allocations. This information follows the mark information template.

• Allocation template	Char(48)
— Allocation address	Space pointer
— Mark identifier	Space pointer
— Allocation size	UBin(4)
— Reserved	Char(12)

If **mark identifier** is null, this allocation is not associated with any mark. If it is not null it contains the most recent mark identifier to which the allocation belongs.

The **heap identifier template** identified by operand 2 must be 16-byte aligned in the space. The format of the template specified by operand 2 is as follows:

• Heap Identifier Template	Char(8)
— Activation Group Mark identifier	UBin(4)
— Heap identifier	UBin(4)

The **activation group mark identifier** may be zero, indicating the heap space specified by the **heap identifier** is in the current Activation Group.

MATHSAT will signal an *activation group access violation* (hex 2C12) exception if a program attempts to materialize heap space attributes of a heap space in an activation group to which the program does not have access.

Operands 1, 2 and 3 are not modified by the instruction.

Events

000C Machine resource
- 0201 Machine auxiliary storage threshold exceeded
- 0501 Machine address threshold exceeded

000D Machine status
- 0101 Machine check
- 0401 Auxiliary storage device requires service

0010 Process
- 0701 Maximum processor time exceeded
- 0801 Process storage limit exceeded

0016 Machine observation
- 0101 Instruction reference
- 0401 Domain violation
- 0402 Hardware storage violation

0017 Damage set
- 0401 System object damage set
- 0801 Partial system object damage set

Exceptions

Exception		Operands 1	2	3	Other
06	Addressing				
	01 space addressing violation	X	X	X	
	02 boundary alignment	X	X	X	
	03 range	X	X	X	
	06 optimized addressability invalid	X	X	X	
08	Argument/parameter				
	01 parameter reference violation	X	X	X	X
10	Damage encountered				
	04 system object damage state	X	X	X	X
	05 authority verification terminated due to damaged object				X
	44 partial system object damage	X	X	X	X
1C	Machine-dependent exception				
	03 machine storage limit exceeded				X
	04 object storage limit exceeded		X		
	09 auxiliary storage pool number invalid		X		
20	Machine support				
	02 machine check				X
	03 function check				X
24	Pointer specification				
	01 pointer does not exist	X	X	X	X
	02 pointer type invalid	X	X	X	X
	03 Pointer addressing invalid object		X	X	
2C	Program execution				
	12 Activation group access violation		X		
	13 Activation group not found		X		
32	Scalar specification				
	03 Scalar value invalid			X	
38	Template specification				
	03 Materialization length			X	
44	Domain specification				
	01 Domain Violation	X	X	X	
45	Heap specification				
	01 Invalid Heap Identifier		X		

Exception	Operands 1	2	3	Other
02 Invalid Request		X		
05 Heap Space Destroyed				X
06 Invalid Heap Condition	X	X	X	X

Reallocate Heap Space Storage (REALCHSS)

Op Code (Hex)	Operand 1	Operand 2
03BA	Space allocation	Size of space reallocation

Operand 1: Space pointer.

Operand 2: Binary(4) scalar.

ILE access

```
REALCHSS (
  space_allocation          : space pointer;
  size_of_space_reallocation : signed binary
) : space pointer /* space_reallocation */
```

Description: A new heap space storage allocation of at least the size indicated by operand 2 is provided from the same heap space as the original allocation, which is indicated by operand 1. The operand 1 space pointer is set to address the first byte of the new allocation, which will begin on a boundary at least as great as the minimum boundary specified when the heap space was created.

Each allocation associated with a heap space provides a continuum of contiguously addressable bytes. Individual allocations within a heap space have no addressability affinity with each other.

The maximum single allocation allowed is determined by the maximum single allocation size specified when the heap space was created. The maximum single allocation possible is 16M-64K bytes.

Storage that is reallocated maintains the same mark/release status as the original allocation. If the original allocation was marked, the new allocation carries the same mark and will be released by a Free Heap Space Storage from Mark (FREHSSMK) which specifies that mark identifier.

The original heap space storage allocation will be freed. Subsequent references to the original allocation will cause unpredictable results.

The contents of the original allocation are preserved in the following fashion:

- If the new allocation size is greater than the original allocation size, the entire contents of the original allocation will appear in the new allocation. The contents of the rest of the new allocation are unpredictable unless initialization of heap allocations was specified when the heap space was created.
- If the new allocation size is less then or equal to the the original allocation size, the new allocation will contain at least as much of the original allocation contents as the new allocation size allows.

REALCHSS will signal a *domain violation* (hex 4401) exception if a program running user state attempts to reallocate heap space storage in a heap space with a *domain* of *Common System*.

Operand 2 is not modified by the instruction.

Events

000C Machine resource
    0201 Machine auxiliary storage threshold exceeded
    0501 Machine address threshold exceeded

| 000D Machine status
| 0101 Machine check
| 0401 Auxiliary storage device requires service

| 0010 Process
| 0701 Maximum processor time exceeded
| 0801 Process storage limit exceeded

| 0016 Machine observation
| 0101 Instruction reference
| 0401 Domain violation
| 0402 Hardware storage violation

| 0017 Damage set
| 0401 System object damage set
| 0801 Partial system object damage set

Exceptions

Exception		Operands 1	2	Other
06	Addressing			
	01 space addressing violation	X	X	
	02 boundary alignment	X	X	
	03 range	X	X	
	06 optimized addressability invalid	X	X	
08	Argument/parameter			
	01 parameter reference violation	X	X	X
10	Damage encountered			
	04 system object damage state	X	X	X
	05 authority verification terminated due to damaged object			X
	44 partial system object damage	X	X	X
1C	Machine-dependent exception			
	03 machine storage limit exceeded			X
20	Machine support			
	02 machine check			X
	03 function check			X
24	Pointer specification			
	01 pointer does not exist	X	X	X
	02 pointer type invalid	X	X	X
	03 Pointer addressing invalid object		X	
44	Domain specification			
	01 Domain Violation	X	X	
45	Heap specification			
	02 Invalid Request		X	

Exception	Operands 1	2	Other
03 Heap Space Full			X
04 Invalid Size Request		X	
05 Heap Space Destroyed			X
06 Invalid Heap Condition	X	X	X

Set Heap Space Storage Mark (SETHSSMK)

Op Code (Hex)	Operand 1	Operand 2
03B6	Mark Identifier	Heap Identifier

Operand 1: Space pointer data object.

Operand 2: Binary(4) scalar.

ILE access

```
SETHSSMK
  var mark_identifier : space pointer;
  var heap_identifier : signed binary
)
```

Description: The heap space identified by operand 2 is marked and the mark identifier is returned in operand 1.

Marking a heap space allows a subsequent Free Heap Space Storage from Mark (FREHSSMK) instruction, using the mark identifier returned in operand 1, to free all outstanding allocations that were performed against the heap space since the heap space was marked with that mark identifier. This relieves the user of performing a Free Heap Space Storage (FREHSS) for every individual heap space allocation.

A heap space may have multiple marks.

The heap identifier specified in operand 2 is the identifier that was returned on the Create Heap Space (CRTHS) instruction. An attempt to mark the default heap space (heap identifier value of 0) will result in a *invalid request* (hex 4502) exception. An attempt to mark a heap space that has been created to not allow a Set Heap Space Storage Mark will result in a *invalid request* (hex 4502) exception.

Operand 2 is not modified by the instruction.

Events

000C Machine resource
- 0201 Machine auxiliary storage threshold exceeded
- 0501 Machine address threshold exceeded

000D Machine status
- 0101 Machine check
- 0401 Auxiliary storage device requires service

0010 Process
- 0701 Maximum processor time exceeded
- 0801 Process storage limit exceeded

0016 Machine observation
- 0101 Instruction reference
- 0401 Domain violation
- 0402 Hardware storage violation

0017 Damage set

0401 System object damage set
0801 Partial system object damage set

Exceptions

Exception		Operands 1	Operands 2	Other
06	Addressing			
	01 space addressing violation	X	X	
	02 boundary alignment	X	X	
	03 range	X	X	
	06 optimized addressability invalid	X	X	
08	Argument/parameter			
	01 parameter reference violation	X	X	
10	Damage encountered			
	04 system object damage state	X	X	X
	05 authority verification terminated due to damaged object			X
	44 partial system object damage	X	X	X
1C	Machine-dependent exception			
	03 machine storage limit exceeded			X
	04 object storage limit exceeded		X	
	09 auxiliary storage pool number invalid		X	
20	Machine support			
	02 machine check			X
	03 function check			X
24	Pointer specification			
	01 pointer does not exist	X	X	
	02 pointer type invalid	X	X	
	03 Pointer addressing invalid object		X	
44	Domain specification			
	01 Domain Violation	X	X	
45	Heap specification			
	01 Invalid Heap Identifier		X	
	02 Invalid Request		X	
	03 Heap Space Full		X	
	05 Heap Space Destroyed			X
	06 Invalid Heap Condition	X	X	X

Chapter 8. Program Management Instructions

This chapter describes all instructions used for program management. These instructions are in alphabetic order. For an alphabetic summary of all the instructions, see Appendix A, "Instruction Summary"

| Materialize Bound Program (MATBPGM) 8-3
Materialize Program (MATPG) 8-24

© Copyright IBM Corp. 1991, 1993

Materialize Bound Program (MATBPGM)

Op Code (Hex)	Operand 1	Operand 2
02C6	Materialization request template	Bound Program or Bound Service Program

Operand 1: Space pointer.

Operand 2: System pointer.

ILE access

```
MATBPGM (
      request_template : space pointer
  var bound_program    : system pointer
)
```

Description: The bound program or bound service program identified by operand 2 is materialized according to the specifications provided by operand 1.

Operand 2 is a system pointer that identifies the bound program or bound service program to be materialized. If Operand 2 does not refer to a program object, an ***object not eligible for operation*** (hex 2403) exception will be signaled. If Operand 2 refers to a program, but not to a bound program or bound service program, then a ***program not eligible for operation*** (hex 220A) exception will be signaled.

The values in the materialization relate to the current attributes of the materialized bound program. Components are the materializable parts of a bound program or bound service program. Components may not be available for materialization because they were not encapsulated during bound program creation. Other components may not be available for materialization because they contain no data.

Note: The bound program materialization request template takes the form of a variable length array of materialization requests.

• Template size specification	Char(8)
— Number of bytes provided	UBin(4)
— Reserved (binary 0)	Char(4)
• Number of materialization requests	UBin(4)
• Reserved (binary 0)	Char(4)
• Array of materialization requests	Char(*)
— Receiver	Space pointer
— Bound Program materialization options	Char(4)
- General Bound Program materialization options	Char(2)
• General bound program information 0 = Do not materialize 1 = Materialize	Bit 0

1 Reserved for IBM Internal Use Only. If used, unpredictable results may occur.

- Reserved[1] — Bit 1
- Program copyright Strings — Bit 2

  0 = Do not materialize
  1 = Materialize

- Bound service programs information — Bit 3

  0 = Do not materialize
  1 = Materialize

- Bound modules information — Bit 4

  0 = Do not materialize
  1 = Materialize

- Bound program string directory component — Bit 5

  0 = Do not materialize
  1 = Materialize

- Bound program limits — Bit 6

  0 = Do not materialize
  1 = Materialize

- Reserved[1] — Bits 7-11
- Activation group data imports — Bit 12

  0 = Do not materialize
  1 = Materialize

- Activation group data exports — Bit 13

  0 = Do not materialize
  1 = Materialize

- Reserved (binary 0) — Bits 14-15

- Specific Bound Program materialization options — Char(1)
  - Specific bound program information — Bit 0

    0 = Do not materialize
    1 = Materialize if the program is a bound program and not a bound service program.

  - Reserved (binary 0) — Bits 1-7
- Specific Bound Service Program — Char(1)
- Materialization options
  - Reserved (binary 0) — Bit 0
  - Signatures information — Bit 1

    0 = Do not materialize
    1 = Materialize if the program is a bound service program.

  - Exported program procedure information — Bit 2

    0 = Do not materialize
    1 = Materialize if the program is a bound service program.

  - Exported program data information — Bit 3

    0 = Do not materialize
    1 = Materialize if the program is a bound service program.

  - Reserved (binary 0) — Bits 4-7

— Bound Modules materialization options	Char(4)
- General module information	Bit 0
0 = Do not materialize 1 = Materialize	
- Reserved[1]	Bit 1
- Module string directory component	Bit 2
0 = Do not materialize 1 = Materialize	
- Reserved[1]	Bits 3-10
- Reserved (binary 0)	Bits 11-17
- Module copyright strings	Bit 18
0 = Do not materialize 1 = Materialize	
- Reserved (binary 0)	Bits 19-31
— Bound Module materialization number	UBin(4)
— Reserved (binary 0)	Char(4)

Description of bound program materialization request template fields: Each of the reserved fields must be set to binary zeroes, or a *template value invalid* (hex 3801) exception will be signaled.

Number of bytes provided

This is the size in bytes of the materialization request template. If this size does not correspond to the actual number of bytes in the materialization request template, then a *template value invalid* (hex 3801) exception will be signaled. This does not include any storage for returned data. That storage is pointed to by the *receiver* values.

Number of materialization requests

The number of requests in the array of materialization requests is specified by this value. If this number is greater than the actual number of materialization requests provided, then a *template value invalid* (hex 3801) exception will be signaled.

Materialization requests

This is an array of materialization requests. A materialization request consists of one or more bits, and an optional module number specified to be materialized into the corresponding *receiver*. Each materialization request consists of the following fields.

Receiver

This is a pointer to a space which will hold the materialized data. The space pointed to must be aligned on a 16-byte boundary, and must be at least 8 bytes long. This is so that it can hold the *bytes provided* and *bytes available* field of the *receiver*. If the space is not at least 8 bytes long a *template value invalid* (hex 3801) exception will be signaled.

Bound Program materialization options

This bit mapped field specifies the parts of the bound program object to be materialized. A materialization request need not specify any program materialization options. If no bits are set, a bit must be set in the *bound module materialization options* field, or a *template value invalid* (hex 3801) exception will be signaled. Mutiple options may be specified. When multiple options are specified, all of the requested data will be materialized into one receiver. The pieces requested on the materialization will be placed in the receiver in the order that the option bits are defined. For example, if all data is requested, the *general bound program information* will be the first item in the receiver and the *exported program data*

information will be the last item. If options are also specified on the *bound module materialization options* field. the materialized data for those options will follow that data materialized for the *bound program materialization options*.

The *bound program materialization options* are split into three distinct materialization bit sets.

1. The **general bound program materialization options** contains bits that represent data that can be materialized for either bound programs or bound service programs.
2. The **specific bound program materialization options** contains bits that represent data that can be materialized only for bound programs, and not for bound service programs.
3. The **specific bound service program materialization options** contains bits that represent data that can be materialized only for bound service programs.

If a bit is on to materialize information that is not contained in the type of bound program being materialized, then an indication that the information is not materializable will be provided in the receiver header. No exception, in this case, will be signaled.

Each of the requested pieces will be placed on a 16-byte boundary within the *receiver*.

The *general bound program information* field specifies that general information about the bound program object should be materialized.

The **program copyright strings** field specifes that the collected program copyright strings of the constituent bound modules should be materialized.

The **bound service programs information** field specifies that information about the service programs bound to the materialized bound program should be materialized. These bound service programs are those that contain exports that resolve to imports in the materialized bound program.

The **bound modules information** field specifies that information about the modules bound into the materialized bound program should be materialized.

The **bound program string directory component** field specifies that the bound program string directory should be materialized.

The **bound program limits** field specifies that the current sizes and maximum values of the bound program components should be materialized.

The **activation group data imports** field specifies that information about those data imports resolving to weak activation data exports is be materialized.

The **activation group data exports** field specifies that information about those data exports promoted to the activation group is to be materialized.

The **specific bound program information** field specifies that information specific to a bound program, and not to a bound service program, should be materialized.

The **signatures information** field specifies that the bound program signatures should be materialized.

The **exported program procedure information** field specifies that the exported procedures specified during program creation should be materialized.

---

[2] Referential extensions are data streams that are not included in the creation templates, but are pointed to by a space pointer in the template.

The **exported program data information** field specifies that the exported data specified during program creation instruction should be materialized.

Bound Module Materialization Options

This bit mapped field specifies the parts of the modules bound into the program that are to be materialized. A module materialization request need not specify any materialization options. If no bits are set, a bit must be set in the *bound program materialization options* field, or a *template value invalid* (hex 3801) exception will be signaled. Multiple options may be specified. When multiple options are specified, all of the requested data will be materialized into one receiver. The pieces requested on the module materialization will be placed in the receiver in the order that the option bits are defined. For example, if all data is requested, the *general module information* will be the first item in the receiver and the *module string directory component* will be the last item. If options are also specified on the *bound program materialization options* field, the materialized data for those options will precede that data materialized for the *bound module materialization options*.

In addition, each of the requested pieces will be placed on a 16 byte boundary within the *receiver*.

The **general module information** field specifies that general information about the module object should be materialized.

The **module string directory component** field specifies that the string directory of the associated module(s) should be materialized.

The **module copyright strings** field specifies that the module copyright strings should be materialized.

Bound Module Materialization Number

If at least one bit of the *Bound Module Materialization Options* field is on, then this is the number of the module to materialize, from 1 through the number of modules bound into the program; or 0, to materialize information about all modules bound into the program. If this number is greater than the number of modules bound into the program, then a *template value invalid* (hex 3801) exception will be signaled.

The information that is materialized is specified in the *bound module materialization options* field.

If no bits of the *bound module materialization options* field are on, then no module information is being materialized, and this field must be binary 0 or a *template value invalid* (hex 3801) exception will be signaled.

Format of materialized data

Format of Receiver

• Number of bytes provided for materialization	UBin(4)
• Number of bytes available for materialization	UBin(4)
• Reserved (binary 0)	Char(8)
• Materialized data	Char(*)

Bytes provided

This is the number of bytes the user is providing to hold the materialized data. It must be greater than or equal to eight. If it is equal to eight, then no data will actually be materialized, and the number of bytes required to materialize the requested data will be placed in *bytes available*. If the value provided is greater than eight, but less than the number of bytes required to hold the requested data, the data will be truncated and no exception will

be signaled. Note that a value greater than eight, but less than 16 will result in no data being materialized, since bytes 9-16 are reserved.

If the receiver is smaller than the size indicated by *bytes provided*, and the materialized data is larger than the space provided in *receiver*, the *space addressing* (hex 0601) exception will be signaled unless *receiver* is an automatically extendable space object. If *receiver* is an automatically extendable space object, the space will be extended, up to its maximum size.

Bytes available

If *bytes provided* is greater than eight, this contains the number of bytes that have been used for the materialization, including any reserved bytes or bytes used for padding. If *bytes provided* is equal to eight, this contains the total size of the *receiver* needed to hold the requested materialization. A value of zero is returned if there is no data to materialize.

Materialized data

For each bit on in the *bound program materialization options* and *bound module materialization options*, this will contain the associated data. Each entry will be preceded by a common header which identifies the type of data and the offset to the next entry. When multiple bits are on in the same request, the data is returned in the order defined by the *bound program materialization options* and the *bound module materialization options.*

No exception is signaled when the size of the *receiver*, as specified by *bytes provided* is not large enough to hold data for all requested *bound program materialization options* and *bound module materialization options.* Instead, the data is truncated and *bytes provided* only reflects the actual amount of data returned. One of several conditions may arise, each with a different result.

If the receiver is not large enough to hold the materialization header, no data is returned for that *bound program materialization option* or *bound module materialization option*. The *offset to next entry* field in the previous materialization header, if one exists, is set to 0, and the *bytes available* field is set to reflect the amount of data actually materialized for the *bound program materialization options* or *bound module materialization options* that have already been processed. *Bytes available* will be set to 8, or *bytes provided* ; whichever is less, if the *receiver* is not big enough to hold the first materialization header.

If the *receiver* is big enough to hold the materialization header, but not big enough to hold all of the data requested by the *bound program materialization option* or *bound module materialization option*, the *partial data* flag will be set in the materialization header and as much data will be returned for which there is room. For data which consists of one continuous stream[3] the receiver will be filled and *bytes available* will equal *bytes provided.* For data which consists of an option specific header followed by an array of homogenous elements[4] data will be returned in such a way that no partial option specific header or array element will be returned. If there is not enough room to hold the entire option specific header, none of it will be returned. If there is room for the option specific header, but not all of the entries, only those entries that will fit will be returned. The number of entries in the option specific header will reflect the number of entries returned rather than the actual number of entires available in the module. *Bytes available* will reflect the actual amount of data returned and may not equal *bytes provided*. Note that because many option specific headers and entries are larger than the common *materialization header*, there may be more than one option for which partial data is returned.

---

[3] The items that fall into this category are general bound program information, bound program limits, specific bound program information, specific bound service program information, general module information, bound program string directory component, module string directory component and module copyright strings.

[4] The items which fall into this catagory are bound service programs information, bound modules information, signatures information, program copyright strings, exported program procedure information, activation group data imports, activation group data exports, and exported program data information.

Format of Common Materialization Header

• Offset to next entry	UBin(4)
• Bound Program materialization identifier	Char(4)
• Bound Module materialization identifier	Char(4)
• Bound Module materialization number identifier	UBin(4)
• Flags	Char(4)
– Entry presence 0 = No data present for entry 1 = Data present for entry	Bit 0
– Partial data 0 = All data in materialization was returned 1 = Partial data was returned because receiver was too small to hold all data for the requested option	Bit 1
– Valid Materialization data 0 = The data requested in this materialization request is never present for the type of bound program being materialized. 1 = The data requested in this materialization request may be present for the type of bound program being materialized.	Bit 2
– Reserved (binary 0)	Bits 3-31
• Reserved (binary 0)	Char(12)

Offset to next entry
This contains the offset from the beginning of this entry to the beginning of the next entry. It will contain zero if this is the last entry.

Bound Program materialization identifier
This indicates which portion of the bound program is contained in this entry. It is the bit which was on in *bound program materialization options* that resulted in this data being materialized. Either no bits, or a single bit of this field will be on. If no bits of this field are on, then the data contained in this entry is indicated by the *bound module materialization identifier* field.

Bound Module materialization identifier
This indicates which portion of the module, indicated by the *bound module materialization number identifier* field, is contained in this entry. It is the bit which was on in the *bound module materialization options* field that resulted in this data being materialized. Either a single bit or no bit of this field will be on. If no bit is on, then the data contained in this entry is indicated by the *bound program materialization identifier* field. If a bit is set on, then that type of information will be returned in the entry.

Bound Module materialization number identifier
If a bit of the *bound module materialization identifier* field is on, then this is the number of the module for which information has been materialized in this entry, and this field will not be 0.

If no bits of the *bound module materialization identifier* field are on, then this field will be 0.

Flags
This field specifies information about the item being materialized.

The **entry presence** field specifies whether there is data available for the requested item. Some items may not be encapsulated into the object, so no data will be returned when their materialization is requested.

The **partial data** field specifies that only a portion of the data was returned because sufficient space was not present in the *receiver* to hold all of the data for the requested materialization option.

The **valid materialization** field specifies whether the requested information is valid to be materialized for the type of bound program that is being materialized. For example, *specific bound program information* is not valid for a bound service program. Even if data may be present for the type of bound program being materialized does not mean that it actually is. Refer to the *entry present* field to see if it is.

Format of materialized general bound program information

- Length in bytes of materialization — UBin(4)
- Reserved[1] — Char(264)
- Number of secondary associated spaces — UBin(4)
- Activation group attributes — Char(4)
  - Target activation group — Char(1)

    0 = Default activation group
    1 = Caller's activation group
    2 = Named activation group
    3 = Unnamed activation group
    4-255 reserved
  - Reserved[1] — Char(3)
- Activation group name — Char(30)
- Reserved[1] — Char(14)
- Coded character set identifier — UBin(2)
- Composite language version[5] — UBin(2)
- Composite version on which module creations occurred[5] — UBin(2)
- Composite machine version[5] — UBin(2)
- Creation target version[5] — UBin(2)
- Version on which creation occurred[5] — UBin(2)
- Bound program identifier — Char(1)

  0 = reserved
  1 = Bound Program
  2 = Bound Service Program
  3-255 = reserved
- Compression status — Char(1)
  - Executable portion — Bit 0
    - 0 = Executable portion is not compressed

[5] All versions are represented as 16 bit values mapped as follows.

- Bits 0-3 — Reserved (binary 0)
- Bits 4-7 — Version
- Bits 8-11 — Release
- Bits 12-15 — Modification

- 1 = Executable portion is compressed
- Observable portion — Bit 1
  - 0 = Observable portion is not compressed
  - 1 = Observable portion is compressed
- Reserved (binary 0) — Bits 2-7

• Reserved[1] — Char(178)

Length in bytes of materialization
This is the number of bytes materialized. For the general bound program information this will always be a constant 512.

Number of secondary associated spaces
This is the number of secondary associated spaces currently associated with the object.

Activation group attributes
The *activation group attributes* specify characteristics of the activation group into which the program will be activated.

Target activation group
This is the *target activation group* value specified when the bound program or bound service program was created.

Activation group name
This is the *activation group name* specified when the bound program or bound service program was created.

Coded character set identifier
This is the CCSID value of the bound program or bound service program.

Composite language version
This is the earliest version of the operating system on which the languages used for the bound modules will allow the bound program object to be saved. This is a composite[6] of all of the language versions of the modules bound into this program.

Composite version on which module creations occurred
This is the earliest version of the operating system on which all of the modules bound into the program can be re-created. This is a composite[6] of all of the version-on-which-module-creations-occurred versions of the modules bound into this program.

Composite machine version
This is the earliest version of the operating system on which the machine will allow the bound program to be saved. This is a composite[6] of all of the machine versions of the modules bound into this program.

Creation target version
This is the version of the operating system for which the bound program object was created.

Version on which creation occurred
This is the version of the operating system on which the bound program object was created.

Bound program identifier
This field identifies the type of bound program being materialized.

[6] A composite version is defined to be the latest version, in time, of all of the versions comprising the composite. Given back-level compatibility, this would be the earliest version of the operating system on which all of the comprising versions would be compatible.

Compression status
This field identifies whether the executable or the observable portions of the bound program or bound service program are compressed.

Format of materialized program copyright strings

• Length in bytes of materialization	Ubin(4)
• Version of copyright creation extension	Ubin(4)
• Number of copyright statements in the pool	Ubin(4)
• Reserved	Char(4)
• Copyright string pool	Char(*)

Length in bytes of materialization
This is the number of bytes materialized.

Version of copyright creation extension
This is the version of the copyrights when the module was created.

Number of copyright strings in the pool.
This is the number of copyright strings that follow.

Copyright statement pool.
This is the data for all of the copyright strings. Each copyright string consists of a 4 byte length followed by the text of the string. The length reflects the actual length of the copyright string and does not include the length of the length field. All copyright strings along with their lengths are placed contiguously in the buffer with no intervening padding.

Format of the materialized bound service programs information

• Length in bytes of materialization	UBin(4)
• Number of service programs bound to this program	UBin(4)
• Reserved (binary 0)	Char(8)
• Array of bound service program records	Char(*)
— Bound service program ID	Char(24)
- Bound service program context object type	Char(1)
- Bound service program context object subtype	Char(1)
- Bound service program context name	Char(10)
- Bound service program object type	Char(1)
- Bound service program object subtype	Char(1)
- Bound service program name	Char(10)
— Referentially bound program signature	Char(16)
— Reserved (binary 0)	Char(8)

Length in bytes of materialization
The number of bytes materialized. This will be 16 + (N * 48) where N is the number of bound servide programs— those programs that contain exports that resolve imports in the bound program.

Number of service programs bound to this program
This is the number of bound service programs bound to the bound program.

Array of bound service program records

This array contains one record for each bound service program bound to the bound program. Each record contains the following information

Bound service program context type

This is the object type of the context with the given name.

Bound service program context subtype

This is the object subtype of the context with the given name.

Bound service program context name

This is the context specified during program creation where this bound service program was found when the bound program was created. This value could be set with all hex zeroes, in which case the name resolution list is used to locate the given bound service program.

Bound service program type

This is the object type of the program with the given name.

Bound service program subtype

This is the object subtype of the program with the given name.

Bound service program name

This is the name of the bound service program specified during program creation.

Bound service program signature

This is the signature of the bound service program that was used to match against the current signature of the bound program.

Format of the materialized bound modules information

• Length in bytes of materialization	UBin(4)
• Number of modules bound into this program	UBin(4)
• Reserved (binary 0)	Char(8)
• Array of bound module records	Char(*)
— Bound module ID	Char(60)
- Module qualifier	Char(30)
- Module name	Char(30)
— Reserved (binary 0)	Char(20)

Length in bytes of materialization

The number of bytes materialized. This will be 16 + (N * 80) where N is the number of modules bound into the bound program.

Number of modules bound into this program

This is the number of modules bound into the bound program.

Array of bound module records

This array contains one record for each module bound into the bound program. Each record contains the following information

Module qualifier

This is the qualifier specified during program creation where this module was found when the bound program was created. The module qualifier is used to differentiate between two different modules of the same name. This usually contains a context name.

Module name
This is the name of the module.

Format of the materialized bound program string directory component

- Length in bytes of materialization — UBin(4)
- Reserved (binary 0) — Char(12)
- String Pool — Char(*)
  - Length of the string — UBin(4)
  - CCSID of the string — UBin(2)
  - String — Char(*)

Length in bytes of materialization
The number of bytes materialized. This will be 16 + the length of the string pool.

String Pool
A memory area containing the strings defined for this program. It can be of any length addressable by a UBin(4). It contains a series of strings and lengths. String IDs specified in other materialized components can be used as indexes into this string pool.

Length of string
The length of the next string. This field contains the length of the string only, and does not include the length of the either the length or the CCSID field. The length field of a string is not subject to alignment considerations.

CCSID of string
The character code set identifier of this string. This string is encoded in the given CCSID, which is the CCSID of the module object from which this string is originally declared. The CCSID field of a string is not subject to alignment considerations.

String
Character buffer which contains one string. Its length is defined by the length field.

Format of the materialized bound program limits

- Length in bytes of materialization — UBin(4)
- Reserved (binary 0) — Char(12)
- Current size of bound program — UBin(4)
- Maximum number of associated spaces — UBin(4)
- Current number of associated spaces — UBin(4)
- Maximum number of modules bindable into program — UBin(4)
- Current number of modules bound into program — UBin(4)
- Maximum number of service programs bindable to program — UBin(4)
- Current number of service programs bound to program — UBin(4)
- Maximum size of bound program string directory — UBin(4)
- Current size of bound program string directory — UBin(4)
- Maximum size of bound program copyright strings — UBin(4)
- Current size of bound program copyright strings — UBin(4)
- Maximum number of auxiliary storage segments — UBin(4)
- Current number of auxiliary storage segments — UBin(4)
- Maximum number of static storage frames — UBin(4)

• Current number of static storage frames	UBin(4)
• Maximum number of program procedure exports	UBin(4)
• Current number of program procedure exports	UBin(4)
• Maximum number of program data exports	UBin(4)
• Current number of program data exports	UBin(4)
• Maximum number of signatures	UBin(4)
• Current number of signatures	UBin(4)
• Minimum amount of static storage required	UBin(4)
• Maximum amount of static storage required	UBin(4)
• Reserved (binary 0)	Char(148)

Length in bytes of materialization
The number of bytes materialized. This will always be a constant 256.

Current size of bound program
This is the current size, in machine-dependent units, of the bound program being materialized.

Maximum number of associated spaces
This is the maximum number of associated spaces allowed for the bound program being materialized.

Current number of associated spaces
This is the current number of associated spaces allocated to the bound program being materialized.

Maximum number of modules bindable into program
This is the maximum number of modules that can be bound into a bound program.

Current number of modules bound into program
This is the current number of modules bound into the bound program being materialized.

Maximum number of service programs bindable to program
This is the maximum number of bound service programs that can be bound to a bound program. These bound service programs contain exports to which imports from a bound program resolve.

Current number of service programs bound to program
This is the current number of bound service programs bound to the bound program being materialized.

Maximum size of bound program string directory
This is the maximum size, in bytes, of the bound program string directory.

Current size of bound program string directory
This is the current size, in bytes, of the bound program string directory.

Maximum size of bound program copyright strings
This is the maximum size, in bytes, of the bound program copyright strings.

Current size of bound program copyright strings
This is the current size, in bytes, of the bound program copyright strings.

Maximum number of auxillary storage segments
This is the maximum number of auxiliary storage segments allowed for a bound program.

Current number of auxillary storage segments
This is the current number of auxiliary storage segments in the bound program being materialized.

Maximum number of static storage frames
This is the maximum number of static storage frames allowed for a bound program.

Current number of static storage frames
This is the current number of static storage frames required by the bound program being materialized.

Maximum number of procedure exports
This is the maximum number of procedures that are allowed to be exported from a bound program. If the bound program being materialized is not a bound service program, then this value will be zero.

Current number of procedure exports
This is the current number of procedures exported from the bound program being materialized. If the bound program being materialized is not a bound service program, then this value will be zero.

Maximum number of data exports
This is the maximum number of data items that are allowed to be exported from a bound program. If the bound program being materialized is not a bound service program, then this value will be zero.

Current number of data exports
This is the current number of data items exported from the bound program being materialized. If the bound program being materialized is not a bound service program, then this value will be zero.

Maximum number of signatures
This is the maximum number of signatures allowed for a bound program. If the bound program being materialized is not a bound service program, then this value will be zero.

Current number of signatures
This is the current number of signatures contained in the bound program being materialized. If the bound program being materialized is not a bound service program, then this value will be zero.

Minimum amount of static storage required.
This is the smallest amount of static storage that is required for the bound program or service program. This measure is in bytes. The actual amount of static storage that is used may be anywhere between the minimum and the maximum amounts of required static storage, inclusive.

Maximum amount of static storage required.
This is the largest amount of static storage that may be required for the bound program or service program. This measure is in bytes. The actual amount of static storage that is used may be anywhere between the minimum and the maximum amounts of required static storage, inclusive.

Format of the materialized activation group data imports

• Length in bytes of materialization	UBin(4)
• Number of activation group data imports	UBin(4)
• Reserved (binary 0)	Char(8)
• Array of activation group data imports	Char(*)
— String ID	UBin(4)
— Reserved (binary 0)	Char(12)

Length in bytes of materialization
The number of bytes materialized. This will be (N+1)*16, where N is the number of activation group data imports contained in the bound program or bound service program.

Number of activation group data imports
The number of activation group data imports contained in the bound program or bound service program.

Array of activation group data imports
This array contains one record for each data item contained in the program or bound service program. Each record contains the following information:

String ID
This is the identification used to extract the name of this data item from the program string directory.

Format of the materialized activation group data exports

• Length in bytes of materialization	UBin(4)
• Number of activation group data exports	UBin(4)
• Reserved (binary 0)	Char(8)
• Array of activation group data exports	Char(*)
— String ID	UBin(4)
— Strength of data item	Char(1)
- 0 = Reserved	
- 1 = Export Strongly	
- 2 = Export Weakly	
- 3-255 = Reserved	
— Reserved (binary 0)	Char(3)
— Length of data item	UBin(4)
— Reserved (binary 0)	Char(4)

Length in bytes of materialization
The number of bytes materialized. This will be (N+1)*16, where N is the number of activation group data exports contained in the bound program or bound service program.

Number of activation group data exports
The number of activation group data exports contained in the bound program or bound service program.

Array of activation group data exports
This array contains one record for each data item contained in the program or bound service program. Each record contains the following information:

String ID
This is the identification used to extract the name of this data item from the program string directory.

Strength of data item
This field indicates whether the activation group export is exported strongly or weakly.

Length of data item.
The size in bytes of the activation group export.

Format of the materialized specific bound program information: Specific bound program information can only be materialized for bound programs, and not for bound service programs.

- Length in bytes of materialization — UBin(4)
- Reserved (binary 0) — Char(12)
- Program entry procedure information — Char(16)
  - Module number containing program entry procedure — UBin(4)
  - Program entry procedure string ID — UBin(4)
  - Minimum parameters — UBin(2)
  - Maximum parameters — UBin(2)
  - Reserved (binary 0) — Char(4)
- Reserved (binary 0) — Char(32)

Length in bytes of materialization
The number of bytes materialized. This will always be a constant 64.

Module number containing program entry procedure
This is the number, in the bound modules information, of the module which contains the program entry procedure for this bound program.

Program entry procedure string ID
This is the string ID for the name of this program entry procedure.

Minimum parameters
This is the minimum number of parameters that the program entry procedure can accept.

Maximum parameters
This is the maximum number of parameters that the program entry procedure can accept.

Format of the materialized signatures information: Signatures information can only be materialized for bound service programs, and not for bound programs.

- Length in bytes of materialization — UBin(4)
- Number of signatures contained in the program — UBin(4)
- Reserved (binary 0) — Char(8)
- Array of signatures — Char(*)
  - Signature — Char(16)

Length in bytes of materialization
The number of bytes materialized. This will be (N+1)*16, where N is the number of signatures contained in the program.

Number of signatures contained in the program
This is the number of signatures contained in the program.

Array of signatures
This array contains one record for each signature contained in the program. Each record contains the following information. The first record contains the current signature.

Signature
A signature of the service program.

Format of the materialized exported program procedure information: Exported program procedure information can only be materialized for bound service programs, and not for bound programs.

- Length in bytes of materialization — UBin(4)
- Number of exported procedures — UBin(4)
- Reserved (binary 0) — Char(8)
- Array of program exports — Char(*)
  - String ID for procedure export — UBin(4)
  - Export number — UBin(4)
  - Reserved (binary 0) — Char(8)

Length in bytes of materialization
The number of bytes materialized. This will be (N+1)*16, where N is the number of exported procedures.

Number of exported procedures
This is the number of procedures exported from the service program.

Array of program exports
This array contains one record for each procedure exported from the service program. Each record contains the following information:

String ID for procedure export
This is the identification used to extract the name of this exported procedure from the program string directory.

Export number
This is the number of this exported procedure.

Format of the materialized exported program data information: Exported program data information can only be materialized for bound service programs, and not for bound programs.

- Length in bytes of materialization — UBin(4)
- Number of exported data items — UBin(4)
- Reserved (binary 0) — Char(8)
- Array of data exports — Char(*)
  - String ID for data export — UBin(4)
  - Export number — UBin(4)
  - Reserved (binary 0) — Char(8)

Length in bytes of materialization
The number of bytes materialized. This will be (N+1)*16, where N is the number of exported data items.

Number of exported data items
This is the number of data items exported from the service program.

Array of data exports
This array contains one record for each data item exported from the service program. Each record contains the following information:

String ID for data export
This is the identification used to extract the name of this exported data item from the program string directory.

Export number
This is the number of this exported data item.

Format of materialized general module information

• Length in bytes of materialization	UBin(4)
• Reserved	Char(12)
• Reserved[1]	Char(276)
• Coded character set identifier	UBin(2)
• Reserved	Char(10)
• Creation target version[5]	Char(2)
• Language version[5]	Char(2)
• Version on which creation occurred[5]	Char(2)
• VLIC version[5]	Char(2)
• Reserved	Char(16)
• Number of secondary associated spaces	UBin(4)
• Reserved	Char(16)
• Reserved[1]	Char(2)
• Reserved	Char(2)
• Compiler name	Char(20)
• Program entry procedure	Char(16)
Program entry procedure attributes	Char(4)
Program entry procedure exists	Bit 0
0 = Program entry procedure does not exist in this module 1 = Program entry procedure exists in this module	
Reserved (0)	Bits 1-31
Program entry procedure dictionary id	UBin(4)
Program entry procedure string id	UBin(4)
Program entry procedure minimum parms	UBin(2)
Program entry procedure maximum parms	UBin(2)
• Reserved	Char(124)

Length in bytes of materialization
This is the number of bytes materialized. For the general module information this will always be a constant 512.

Coded character set identifier
The CCSID defines the code page of the symbols in the string directory.

Language version
This is the earliest version of the operating system on which language used will allow the module object to be saved.

Creation target version
This is the version of the operating system for which the module object was created.

Version on which creation occurred
This is the version of the operating system that was running on the system where the module object was created.

Earliest version
This is the earliest version of the operating system on which the machine will allow the module object to be saved.

Number of secondary associated spaces
This is the number of secondary associated space segments currently associated with the object.

Compiler name
This identifies the compiler which translated the user's source language.

Program entry procedure
This identifies the program entry procedure if one is present in the module.

Program entry procedure attributes
This bit mapped field identifies attributes of the program entry procedure.

The *program entry procedure existence* field specifies whether a program entry procedure is present in the module being materialized.

Dictionary id of the procedure
The dictionary id is used as a handle to uniquely identify the procedure.

String id of the procedure name
The string id may be used to extract the character string which is the procedure name from the string pool.

Program entry procedure minimum parms
This is the minimum number of parameters allowed by the program entry procedure.

Program entry procedure maximum parms
This is the maximum number of parameters allowed by the program entry procedure.

Format of the materialized module string directory component

- Length in bytes of materialization — UBin(4)
- Reserved (binary 0) — Char(12)
- String Pool — Char(*)
  - Length of the string — UBin(4)
  - String — Char(*)

Length in bytes of materialization
The number of bytes materialized. This will be 16 + the length of the string pool.

String Pool
A memory area containing the strings defined for this module. It can be of any length addressable by a UBin(4). It contains a series of strings and lengths. String IDs specified in other materialized components can be used as indexes into this string pool.

Length of string
The length of the next string. This field contains the length of the string only, and does not include the length of the length field, itself. The length field of a string is not subject to alignment considerations.

String
Character buffer which contains one string. Its length is defined by the length field.

Format of the materialized module copyright strings: The format of the materialized module copyright strings is the same as for the materialized program copyright strings.

Template Value Invalid exception reason codes: This instruction supports setting of the optional reason code field in the exception data which can be retrieved when the template value invalid exception is signaled. When the first byte of the reason code is not zero, the exception is being signaled because one of the materialization receivers is not valid.

00 Bound Program Materialization Template (pointed to by operand 1 of this instruction)
- 01 Size of template is not sufficient to hold number of requests specified.

0n nth materialization request is not valid.
- 01 The *receiver* is not aligned on a 16 byte boundary.
- 02 The materialization request *bytes provided* is less than 8.
- 03 The materialization request contains no *materialization options* or invalid *materialization options*.

  If the *length of field* data is 8, then no materialization options were specified and the *offset in field in bits* data will be 0. Otherwise, an invalid option was specified and the provided *offset to field in bytes* and *offset in field in bits* data will identify the invalid materialization option.
- 04 The materialization request contains a *module materialization number* that is greater than the number of modules bound into the program.
- 05 The materialization request contains a non-zero *module materialization number*, but no *module materialization options*.
- 06 The materialization request contains a non-zero reserved field.

Authorization Required

- Retrieve
  - Operand 2
  - Contexts referenced for address resolution

Lock Enforcement

- Materialize
  - Operand 2
  - Contexts referenced for address resolution

Exceptions

Exception		Operands 1	2	Other
06	Addressing			
	01 space addressing violation	X	X	
	02 boundary alignment	X	X	
	03 range	X	X	
	06 optimized addressability invalid	X	X	
08	Argument/parameter			
	01 parameter reference violation	X	X	
0A	Authorization			
	01 unauthorized for operation		X	
10	Damage encountered			
	04 system object damage state	X	X	X
	05 authority verification terminated due to damaged object			X
	44 partial system object damage	X	X	X

Exception		Operands 1	2	Other
1A	Lock state			
	01 invalid lock state		X	
1C	Machine-dependent exception			
	03 machine storage limit exceeded			X
20	Machine support			
	02 machine check			X
	03 function check			X
22	Object access			
	01 object not found	X	X	
	02 object destroyed	X	X	
	03 object suspended	X	X	
	04 object inelligible	X	X	
	07 authority verification terminated due to destroyed object			X
	0A program object inelligible		X	
	08 object compressed			X
24	Pointer specification			
	01 pointer does not exist	X	X	
	02 pointer type invalid	X	X	
	03 pointer addressing invalid object		X	
2E	Resource control limit			
	01 user profile storage limit exceeded			X
36	Space management			
	01 space extension/truncation			X
38	Template specification			
	01 template value invalid	X		X
	02 template size invalid	X		X
	03 materialization length exception			X

Materialize Program (MATPG)

Op Code (Hex)	Operand 1	Operand 2
0232	Attribute receiver	Program

Operand 1: Space pointer.

Operand 2: System pointer.

ILE access

```
MATPG (
     receiver   : space pointer;
 var program    : system pointer
)
```

Description: The program identified by operand 2 is materialized into the template identified by operand 1.

Operand 2 is a system pointer that identifies the program to be materialized. The values in the materialization relate to the current attributes of the materialized program.

The template identified by operand 1 must be 16-byte aligned.

The first 4 bytes of the materialization template identify the total **number of bytes provided** in the template. This value is supplied as input to the instruction and is not modified. A value of less than 8 causes the *materialization length* (hex 3803) exception to be signaled.

The second 4 bytes of the materialization template are modified by the instruction to contain a value identifying the template size required to provide for the total **number of bytes available** to be materialized. The instruction materializes as many bytes as can be contained in the area specified by the receiver. If the byte area identified by the receiver is greater than that required to contain the information requested, then the excess bytes are unchanged. No exceptions (other than the *materialization length* (hex 3803) exception) are signaled in the event that the receiver contains insufficient area for the materialization.

The following attributes apply to the materialization of a program:

- The **existence attribute** indicates whether the program is *temporary* or *permanent*.
- The **observation attribute** field specifies the template components of the programs that currently can be materialized.
- If the program has an associated space, then the **space attribute** is set to indicate either fixed- or variable-length; the initial value for the space is returned in the **initial value of space** field, and the **size of space** field is set to the current size value of the space. If the program has no associated space, the *size of space* field is set to a zero value, and the *space attribute* and *initial value of space* field values are meaningless.
- If the program is addressed by a context, then the **context addressability** attribute is set to indicate this, and a system pointer to the addressing context is returned in the **context** field. If the program is not addressed by a context, then the *context addressability* attribute is set to indicate this, and binary 0's are returned in the *context* field.
- If the program is a member of an access group, then the **access group** attribute is set to indicate this, and a system pointer to the access group is returned in the **access group** field. If the program

is not a member of an access group, then the *access group* attribute is set to indicate this, and binary 0's are returned in the *access group* field.

- The **performance class** field is set to reflect the performance class information associated with the program.
- The **user exit** attribute defines if the referenced program is allowed to be used as a user exit program.

Components of the program template may not be present if they have been removed by the Delete Program Observability instruction. Also, a template component will not be available if the program was created with its corresponding *observation attribute* in the Create Program template set to binary 0. If a template component cannot be materialized, 0's are placed in the fields of the program template that describe the size and offsets to that component.

The **offset to the OMT component** field specifies the location of the OMT component in the materialized program template. The OMT consists of a variable-length vector of 6-byte entries. The number of entries is identical to the number of ODV entries because there is one OMT entry for each ODV entry. The OMT entries correspond one for one with the ODV entries; each OMT entry gives a location mapping for the object defined by its associated ODV entry.

The following describes the formats for an OMT entry:

- OMT entry — Char(6)
  - Addressability type — Char(1)

    Hex 00 = Base addressability is from the start of the static storage

    Hex 01 = Base addressability is from the start of the automatic storage area

    Hex 02 = Base addressability is from the start of the storage area addressed by a space pointer

    Hex 03 = Base addressability is from the start of the storage area of a parameter

    Hex 04 = Base addressability is from the start of the storage area addressed by the space pointer found in the process communication object attribute of the process executing the program

    Hex FF = Base addressability not provided. The object is contained in machine storage areas to which addressability cannot be given, or a parameter has addressability to an object that is in the storage of another program

  - Offset from base — Char(3)

    For types hex 00, hex 01, hex 02, hex 03, and hex 04, this is a 3-byte logical binary value representing the offset to the object from the base addressability. For type and hex FF, the value is binary 0.

  - Base addressability — Char(2)

    For types hex 02 and hex 03, this is a 2-byte binary field containing the number of the OMT entry for the space pointer or a parameter that provides base addressability for this object. For types hex 00, hex 01, hex 04 and hex FF, the value is binary 0.

Authorization Required

- Retrieve
  - Operand 2
  - Contexts referenced for address resolution

Lock Enforcement

- Materialize
  - Operand 2
  - Contexts referenced for address resolution

Exceptions

Exception		Operands 1	Operands 2	Other
06	Addressing			
	01 space addressing violation	X	X	
	02 boundary alignment	X	X	
	03 range	X	X	
	06 optimized addressability invalid	X	X	
08	Argument/parameter			
	01 parameter reference violation	X	X	
0A	Authorization			
	01 unauthorized for operation		X	
10	Damage encountered			
	04 system object damage state	X	X	X
	05 authority verification terminated due to damaged object			X
	44 partial system object damage	X	X	X
1A	Lock state			
	01 invalid lock state		X	
1C	Machine-dependent exception			
	03 machine storage limit exceeded			X
20	Machine support			
	02 machine check			X
	03 function check			X
22	Object access			
	01 object not found	X	X	
	02 object destroyed	X	X	
	03 object suspended	X	X	
	07 authority verification terminated due to destroyed object			X
	08 object compressed			X
24	Pointer specification			
	01 pointer does not exist	X	X	
	02 pointer type invalid	X	X	
	03 pointer addressing invalid object		X	
2E	Resource control limit			
	01 user profile storage limit exceeded			X

Exception		Operands 1	2	Other
36	Space management			
	01 space extension/truncation			X
38	Template specification			
	03 materialization length exception	X		

Chapter 9. Program Execution Instructions

This chapter describes the instructions used for program execution control. These instructions are in alphabetic order. For an alphabetic summary of all the instructions, see Appendix A, "Instruction Summary."

Activate Program (ACTPG) 9-3
Call External (CALLX) 9-5
Call Internal (CALLI) 9-9
Clear Invocation Exit (CLRIEXIT) 9-11
Deactivate Program (DEACTPG) 9-12
End (END) 9-14
| Materialize Activation Attributes (MATACTAT) 9-15
| Materialize Activation Group Attributes (MATAGPAT) 9-19
Modify Automatic Storage Allocation (MODASA) 9-23
Return External (RTX) 9-25
Set Argument List Length (SETALLEN) 9-27
Set Invocation Exit (SETIEXIT) 9-29
Store Parameter List Length (STPLLEN) 9-31
Transfer Control (XCTL) 9-33

© Copyright IBM Corp. 1991, 1993

Activate Program (ACTPG)

Op Code (Hex)	Operand 1	Operand 2
0212	Program or static storage frame	Program

Operand 1: Space pointer data object, system pointer.

Operand 2: System pointer.

ILE access

```
ACTPG (
 var activation : system pointer; OR
                  space pointer;
 var program    : system pointer
)
```

Warning: The following information is subject to change from release to release. Use it with caution and be prepared to adjust for changes with each new release.

Description: This instruction creates an activation entry for the program specified by operand 2, if it uses static storage. If the program specified is a bound program, an ***invalid operation for program*** (hex 2C15) exception is signaled. No operation is performed for a program which does not require static storage.

Operand 1 receives either a space pointer or system pointer as follows:

If an activation entry is created or an activation entry exists for the program within the target activation group, then a space pointer to the static storage frame is returned. The static storage frame is allocated and initialized according to specifications within the program. The static storage frame is 16-byte aligned and begins with a 64-byte header. The header is not initialized and it is not used by the machine. The header is provided for compatibility with prior machine implementations.

- If the program does not use static storage (hence, no activation entry is created) a copy of the program pointer in operand 2 is returned.

If an attempt is made to activate an already active program then

- the activation mark of the activation entry is unchanged, and
- the static storage frame is reinitialized

A space pointer machine object may not be specified for operand 1.

Authorization Required

- Operational
  - Program referenced by operand 2
- Retrieve
  - Contexts referenced for address resolution

Lock Enforcement

- Materialize
  - Contexts referenced for address resolution

Exceptions

Exception		Operands 1	2	Other
06	Addressing			
	01 space addressing violation	X	X	
	02 boundary alignment	X	X	
	03 range	X	X	
	06 optimized addressability invalid	X	X	
0A	Authorization			
	01 unauthorized for operation		X	
10	Damage encountered			
	04 system object damage state	X	X	X
	05 authority verification terminated due to damaged object			X
	44 partial system object damage	X	X	X
1A	Lock state			
	01 invalid lock state		X	
1C	Machine-dependent exception			
	03 machine storage limit exceeded			X
20	Machine support			
	02 machine check			X
	03 function check			X
22	Object access			
	01 object not found	X	X	
	02 object destroyed	X	X	
	03 object suspended	X	X	
	07 authority verification terminated due to destroyed object			X
	08 object compressed			X
24	Pointer specification			
	01 pointer does not exist		X	
	02 pointer type invalid	X	X	
	03 pointer addressing invalid object		X	
2C	Program execution			
	15 invalid operation for program	X		
2E	Resource control limit			
	01 user profile storage limit exceeded			X
36	Space management			

Exception	Operands 1	2	Other
01 space extension/truncation			X

Call External (CALLX)

Op Code (Hex)	Operand 1	Operand 2	Operand 3
0283	Program to be called or Call template	Argument list	Return list

Operand 1: System pointer or Space pointer data object.

Operand 2: Operand list or null.

Operand 3: Instruction definition list or null.

Warning: The following information is subject to change from release to release. Use it with caution and be prepared to adjust for changes with each new release.

Description: The instruction preserves the calling invocation and passes control to either the *program entry procedure* of a bound program or the external entry point of a non-bound program. If operand 1 specifies a bound program which does not contain a *program entry procedure*, an *invalid operation for program* (hex 2C15) exception is signaled.

Operand 1 may be specified as a system pointer which directly addresses the program that is to receive control or as a space pointer to a call template which identifies the program to receive control. Specifying a template allows for additional controls over how the specified program is to be invoked. The format of the call template is the following:

- Call options — Char(4)
  - Suppress adopted user profiles — Bit 0
    - 0 = No
    - 1 = Yes
  - Reserved (binary 0) — Bits 1-30
  - Force process state to user state for call — Bit 31
    - 0 = No
    - 1 = Yes
- Reserved (binary 0) — Char(12)
- Program to be called — System pointer

The **suppress adopted user profiles** call option specifies whether or not the program adopted and propagated user profiles which may be serving as sources of authority to the process are to be suppressed from supplying authority to the new invocation. Specifying *yes* causes the propagation of adopted user profiles to be stopped as of the calling invocation, thereby, not allowing the called invocation to benefit from their authority. Specifying *no* allows the normal propagation of adopted and propagated user profiles to occur. The called program may adopt its owning user profile, if necessary, to supplement the authority available to its invocation.

The **force process state to user state** option specifies whether or not the call changes the state of the process to user state.

Common Program Call Processing: The details of processing differ for non-bound and bound programs. The following outlines the common steps.

1. A check is made to determine if the caller has authority to invoke the program and that the object is indeed a program object. The specified program must be either a bound program that contains a *program entry procedure* or a non-bound program.
2. The activation group in which the program is to be run is located or created if it doesn't exist. The activation group supplies the storage resources required for program execution: static, automatic, and heap storage.
3. If the program requires an activation entry and it is not already active within the appropriate activation group, it is activated. Bound programs always require an activation; non-bound programs require an activation only if they use static storage. The *invocation count* of a newly created activation is set to 1 while the *invocation count* of an existing activation is incremented by 1.
4. The invocation created for the target program has the following attributes (as would be reported via the Materialize Invocation Attributes instruction.)
   - the *invocation mark* is one higher than the current mark count value maintained for the process. If the program was activated as a result of the call then its invocation mark will equal the activation mark, otherwise the invocation mark is larger than the activation mark.

     Note: The so-called mark counts are generated from a counter maintained for the process. Each time a mark is required the counter is incremented. The mark counts thus form a non-decreasing sequence of unique identifiers which can be used to distinguish the time ordering of activations, invocations, and activation groups.
   - the *invocation number* is one greater than the invocation number of the calling invocation. This is merely a measure of the depth of the call-stack.
   - the *invocation type* is hex 01 to indicate a CALLPGM or CALLX invocation.
   - the *invocation number* is the same as the invocation number of the transferring invocation.
   - the *invocation type* is hex 02 to indicate a XCTL type of invocation.
5. The automatic storage frame (ASF), if required, is allocated on a 16-byte boundary.
6. Control is transferred to the program entry procedure (or external entry point) of the program.
7. Normal flow-of-control resumes at the instruction following the program call instruction after a return from the program.
8. Normal flow-of-control resume at the instruction following the caller of the program issuing the XCTL instruction.

The details of locating the target activation group and activating the program differ depending upon the model of the program.

Bound Program: A bound program is activated and run in an activation group specified by program attributes. There are two logical steps involved:

- locate the existing, or create a new activation group for the program
- locate an existing, or create a new activation entry for the program within the activation group.

After locating the activation entry for the program, control is passed to the program entry procedure for the program. If required, the activation group is destroyed when the invocation for the program entry procedure is destroyed.

Non-bound Program: The automatic storage frame begins with a 64 byte header. if the program defines no automatic data items the frame consists solely of the 64-byte header, otherwise the automatic storage items are located immediately following the header. In prior releases of the machine, this header contained invocation information which is now available via the Materialize Invocation Attributes (MATINVAT) instruction. This header is not initialized and the contents of the header are not used by the machine. (The space is allocated merely to provide for compatibility with prior implementations of the machine.) The *update PASA stack* program attribute, supported in prior implementations of the machine, is no longer meaningful and is ignored, if specified as an attribute of the program.

Following the allocation and initialization of the invocation entry, control is passed to the invoked program.

Operand 2 specifies an operand list that identifies the arguments to be passed to the invocation entry to be called. If operand 2 is null, no arguments are passed by the instruction. A *parameter list length* (hex 0802) exception is signaled if the number of arguments passed does not correspond to the number required by the parameter list of the target program.

Operand 3 specifies an IDL (instruction definition list) that identifies the instruction number(s) of alternate return points within the calling invocation. A Return External instruction in an invocation immediately subordinate to the calling invocation can indirectly reference a specific entry in the IDL to cause a return of control to the instruction associated with the referenced IDL entry. If operand 3 is null, then the calling invocation has no alternate return points associated with the call. If operand 3 is not null and operand 1 specifies a bound program, an *invalid operation for program* (hex 2C15) exception is signaled.

Authorization Required

- Operational
  - Program referenced by operand 1
- Retrieve
  - Contexts referenced for address resolution

Lock Enforcement

- Materialize
  - Contexts referenced for address resolution

Exceptions

Exception		Operands 1	2	3	Other
06	Addressing				
	01 space addressing violation	X			
	02 boundary alignment	X			
	03 range	X			
	06 optimized addressability invalid	X			
08	Argument/parameter				
	01 parameter reference violation	X			
	02 parameter list length violation		X		
0A	Authorization				
	01 unauthorized for operation	X			

Exception		Operands 1	2	3	Other
10	Damage encountered				
	04 system object damage state	X	X	X	X
	05 authority verification terminated due to damaged object				X
	44 partial system object damage	X	X	X	X
1A	Lock state				
	01 invalid lock state	X			
1C	Machine-dependent exception				
	03 machine storage limit exceeded				X
20	Machine support				
	02 machine check				X
	03 function check				X
22	Object access				
	01 object not found	X			
	02 object destroyed	X			
	03 object suspended	X			
	07 authority verification terminated due to destroyed object				X
	08 object compressed				X
24	Pointer specification				
	01 pointer does not exist	X			
	02 pointer type invalid	X			
	03 pointer addressing invalid object	X			
2C	Program execution				
	15 invalid operation for program	X		X	
	1D automatic storage overflow				X
	1E activation access violation				X
	1F program signature violation				X
	20 static storage overflow				X
	21 program import invalid				X
	22 data reference invalid				X
	23 imported object invalid				X
	24 activation group export conflict				X
	25 import not found				X
2E	Resource control limit				
	01 user profile storage limit exceeded				X
36	Space management				
	01 space extension/truncation				X
38	Template specification				
	01 template value invalid	X			

Call Internal (CALLI)

Op Code (Hex)	Operand 1	Operand 2	Operand 3
0293	Internal entry point	Argument list	Return target

Operand 1: Internal entry point.

Operand 2: Operand list or null.

Operand 3: Instruction pointer.

Warning: The following information is subject to change from release to release. Use it with caution and be prepared to adjust for changes with each new release.

Description: The internal entry point specified by operand 1 is located in the same invocation from which the Call Internal instruction is executed. A subinvocation is defined and execution control is transferred to the first instruction associated with the internal entry point. The instruction does not cause a new invocation to be established. Therefore, there is no allocation of objects and instructions in the subinvocation have access to all invocation objects.

Operand 2 specifies an operand list that identifies the arguments to be passed to the subinvocation. If operand 2 is null, no arguments are passed. After an argument has been passed on a Call Internal instruction, the corresponding parameter may be referenced. This causes an indirect reference to the storage area located by the argument. This mapping exists until the parameter is assigned a new mapping based on a subsequent Call Internal instruction. A reference to an internal parameter before its being assigned an argument mapping causes a *parameter reference violation* (hex 0801) exception to be signaled.

Operand 3 specifies an instruction pointer that identifies the pointer into which the machine places addressability to the instruction immediately following the Call Internal instruction. A branch instruction in the called subinvocation can directly reference this instruction pointer to cause control to be passed back to the instruction immediately following the Call Internal instruction.

Exceptions

Exception		Operands 1	2	3	Other
06	Addressing				
	01 space addressing violation			X	
	02 boundary alignment			X	
	03 range			X	
	06 optimized addressability invalid			X	
08	Argument/parameter				
	01 parameter reference violation		X		
10	Damage encountered				
	04 system object damage state	X	X	X	X
	44 partial system object damage	X	X	X	X
1C	Machine-dependent exception				
	03 machine storage limit exceeded				X

Exception		Operands 1	2	3	Other
20	Machine support				
	02 machine check				X
	03 function check				X
22	Object access				
	08 object compressed				X
24	Pointer specification				
	01 pointer does not exist			X	
	02 pointer type invalid			X	
2E	Resource control limit				
	01 user profile storage limit exceeded				X
36	Space management				
	01 space extension/truncation				X

Clear Invocation Exit (CLRIEXIT)

Op Code (Hex)
0250

Description: The instruction removes the invocation exit program for the requesting invocation. No exception is signaled if an invocation exit program is not specified for the current invocation. Also, an implicit clear of the invocation exit occurs when the invocation exit program is given control, or the program which set the invocation exit completes execution.

Exceptions

Exception		Other
10	Damage encountered	
	04 System object damage state	X
	44 Partial system object damage	X
1C	Machine-dependent exception	
	03 Machine storage limit exceeded	X
20	Machine support	
	02 Machine check	X
	03 Function check	X
22	Object access	
	08 object compressed	X
2E	Resource control limit	
	01 user profile storage limit exceeded	X
36	Space management	
	01 space extension/truncation	X

Deactivate Program (DEACTPG)

Op Code (Hex)	Operand 1
0225	Program

Operand 1: System pointer or null.

ILE access

```
DEACTPG (
 var program : system pointer or
               null operand
)
```

Description: This instruction, provided that certain conditions are met, deactivates a non-bound program. Subsequent invocations of the program within the same activation group will cause a new activation to be created.

Operand 1 specifies program activation entry which is to be deactivated, if permitted. The activation entry is inferred by one of two means:

1. *operand 1 is null* — the target activation entry is that associated with the current invocation
2. *operand 1 is not null* — the target activation entry associated with the program system pointer is selected from one of the two default activation groups

The target activation entry is deactivated if permitted. An *activation in use by invocation* (hex 2C05) exception is signaled if the deactivation is not permitted. If the target activation entry does not exist, then no operation is performed. If the program specified is a bound program, an *invalid operation for program* (hex 2C15) exception is signaled.

In general, only those activations with a zero invocation count can be deactivated. The following two exceptions apply:

1. A program can deactivate itself if it is the only invocation of that program in the process (its invocation count must be 1.)
2. An invocation exit program can deactivate the program on whose behalf it is running provided that the invocation count of that program is no more than 1.

Authorization Required

- Retrieve
  - Contexts referenced for address resolution

Lock Enforcement

- Materialize
  - Contexts referenced for address resolution

Exceptions invalid operation for program

Exception		Operand 1	Other
06	Addressing		

Exception		Operand 1	Other
	01 space addressing violation	X	
	02 boundary alignment	X	
	03 range	X	
	06 optimized addressability invalid	X	
08	Argument/parameter		
	01 parameter reference violation	X	
0A	Authorization		
	01 unauthorized for operation	X	
10	Damage encountered		
	04 system object damage state	X	X
	05 authority verification terminated due to damaged object		X
	44 partial system object damage	X	X
1A	Lock state		
	01 invalid lock state	X	
20	Machine support		
	02 machine check		X
	03 function check		X
22	Object access		
	01 object not found	X	
	02 object destroyed	X	
	03 object suspended	X	
	07 authority verification terminated due to destroyed object		X
	08 object compressed		X
24	Pointer specification		
	01 pointer does not exist	X	
	02 pointer type invalid	X	
	03 pointer addressing invalid object	X	
2C	Program execution		
	05 activation in use by invocation	X	
	15 invalid operation for program	X	
2E	Resource control limit		
	01 user profile storage limit exceeded		X
32	Scalar specification		
	01 scalar type invalid	X	
36	Space management		
	01 space extension/truncation		X

End (END)

Op Code (Hex)
0260

Description: The instruction delimits the end of a program's instruction stream. When this instruction is encountered in execution, it causes a return to the preceding invocation (if present) or causes termination of the process phase if the instruction is executed in the highest-level invocation for a process. The End instruction delineates the end of the instruction stream. When it is encountered in execution, the instruction functions as a Return External instruction with a null operand. Refer to the Return External instruction for a description of that instruction.

Exceptions

Exception		Other
10	Damage encountered	
	04 system object damage state	X
	44 partial system object damage	X
1C	Machine-dependent exception	
	03 machine storage limit exceeded	X
20	Machine support	
	02 machine check	X
	03 function check	X
22	Object access	
	08 object compressed	X
2E	Resource control limit	
	01 user profile storage limit exceeded	X
36	Space management	
	01 space extension/truncation	X

Materialize Activation Attributes (MATACTAT)

Op Code (Hex)	Operand 1	Operand 2	Operand 3
0213	Receiver	Activation mark	Attribute selection

Operand 1: Space pointer

Operand 2: Binary(4) scalar

Operand 3: Character(1) scalar (fixed length)

ILE access

```
MATACTAT (
      receiver             : space pointer;
  var activation_mark      : unsigned binary;
  var attribute_selection  : aggregate
)
```

Warning: The following information is subject to change from release to release. Use it with caution and be prepared to adjust for changes with each new release.

Description: This instruction will materialize the information selected by operand 3 for the program activation specified by operand 2 and return the information in the template supplied by operand 1.

The operand 3 selection operand is provided to deal with the variable-length nature of some of the returned information. All "length-of-list" type information can be gathered by selecting the first option described below.

Operand 3 can have the following values:

- hex 00 — basic activation attributes
- hex 01 — static storage frame list

Any value for operand 3 other than those listed will cause a *scalar value invalid* (hex 3203) exception.

Operand 2 supplies the activation mark of the activation for which information is to be returned. The activation mark uniquely identifies an activation within a process. A value of zero is interpreted to be a request for information about the activation of the invoking program.

The materialization template identified by operand 1 must be 16-byte aligned in the space. This materialization template has the following format:

• Template size specification	Char(8)
Number of bytes provided for materialization	Bin(4)
— Number of bytes available for materialization	Bin(4)
— Reserved (binary 0)	Char(8)
• Returned information	Char(*)

The **number of bytes provided** indicates the number of bytes provided for returned information by the user of the instruction. In all cases if the number of bytes provided is less than 8, then a *materialization length* (hex 3803) exception will be signaled.

The **number of bytes available** is set by the instruction to indicate the actual number of bytes available to be returned. In no case does the instruction return more bytes of information than those available.

The format of **returned information** is described in the following paragraphs.

Basic Activation Attributes: The following information is returned when operand 3 is hex 00.

• Program	Aystem pointer
• Activation mark	UBin(4)
• Activation group mark	UBin(4)
• Invocation count	UBin(4)
• Static frame count	UBin(4)
• Program model	Char(1)
Hex 00 = original model Hex 01 = new model Hex 02-FF = reserved	
• Activation attributes	Char(3)
— Activation status	bit 0
0 = inactive 1 = active	
— Reserved (binary 0)	bits 1-23

A description of the fields follows.

program
This is a pointer to the program. The system pointer returned does not contain authority. Within a process, a program may have more than one activation.

activation mark
The *activation mark* identifies the activation within the process. This field provides the actual activation mark when the special zero value was supplied for operand 2. Otherwise, this field has the same value as operand 2.

activation group mark
This identifies the activation group which contains the activation.

invocation count
This is a count of the number of program invocations which currently exist for this activation of the program. Recall that a program invocation results from a *program call* operation like Call Program not a *procedure call* operation like Call Bound Procedure.

static frame count
This is the number of static storage frames allocated for this activation.

program model
the model of the program. A program is either an *original model* (non-bound) or *new model* (bound) program.

activation attributes
The activation attributes identify

- whether the program is active or not. Only original model program can be deactivated by use of the Deactivate Program instruction.

Static Storage Frame List: The following information is returned when operand 3 is hex 01. This is a list of static storage frame descriptors. The *static frame count* (available in the basic activation attributes template) indicates how many entries must be accommodated by the template. The static storage frame list can be materialized only if the source activation group is permitted access to the target activation group as determined by the activation group access protection mechanism. If access is not permitted, then an *activation group access violation* (hex 2C12) exception is signaled.

The format of the list is:

- array(1..*static frame count*) of
  - Static frame base — Space pointer

    This is a pointer to the first byte of the static frame.
  - Static frame size — UBin(4)

    This is the size, in machine dependent units (currently bytes), of the static frame.
  - Reserved — Char(12)

Exceptions

Exception		Operands 1	2	3	Other
06	Addressing				
	01 space addressing violation	X	X	X	
	02 boundary alignment	X	X	X	
	03 range	X	X	X	
	06 optimized addressability invalid	X	X	X	
08	Argument/parameter				
	01 parameter reference violation	X	X	X	X
10	Damage encountered				
	04 system object damage state	X	X	X	X
	05 authority verification terminated due to damaged object				X
	44 partial system object damage	X	X	X	X
1C	Machine-dependent exception				
	03 machine storage limit exceeded				X
	04 object storage limit exceeded		X		
	09 auxiliary storage pool number invalid		X		
20	Machine support				
	02 machine check				X
	03 function check				X
22	Object access				
	02 object destroyed				X
24	Pointer specification				
	01 pointer does not exist	X	X	X	X

Exception		Operands 1	2	3	Other
	02 pointer type invalid	X	X	X	X
	03 Pointer addressing invalid object		X	X	
2C	Program execution				
	12 Activation group access violation		X		
	16 Activation not found		X		
32	Scalar specification				
	03 Scalar value invalid			X	
38	Template specification				
	03 Materialization length	X			
44	Domain specification				
	01 Domain Violation	X			

Materialize Activation Group Attributes (MATAGPAT)

Op Code (Hex)	Operand 1	Operand 2	Operand 3
02D3	Receiver	Activation group mark	Attribute selection

Operand 1: Space pointer

Operand 2: Unsigned binary(4) scalar

Operand 3: Character(1) scalar (fixed length)

ILE access

```
MATAGPAT (
        receiver                 : space pointer;
    var activation_group_mark : unsigned binary;
    var attribute_selection   : aggregate
)
```

Warning: The following information is subject to change from release to release. Use it with caution and be prepared to adjust for changes with each new release.

Description: This instruction will materialize the information selected by operand 3 for the activation group specified by operand 2 and return the information in the template supplied by operand 1. If the activation group mark specified by operand 2 is zero, then information about the activation group associated with the current invocation is returned.

In order to deal with the variable-length nature of some activation group attributes, the selection option is provided. All of the "length-of-list" type information can be gathered by selecting the first option described below.

Operand 3 can have the following values:

- hex 00 — basic activation group attributes
- hex 01 — activation group heap list option
- hex 02 — program activation list option

Any value for operand 3 other than those listed will cause a *scalar value invalid* (hex 3203) exception.

The materialization template identified by operand 1 must be 16-byte aligned in the space. This materialization template has the following format:

- Materialization size specification — Char(8)
  - Number of bytes provided for materialization — Bin(4)
  - Number of bytes available for materialization — Bin(4)
- Reserved (binary 0) — Char(8)
- Returned information — Char(*)

  The **number of bytes provided** indicates the number of bytes provided for returned information by the user of the instruction. In all cases if the number of bytes provided is less than 8, then a *materialization length* (hex 3803) exception will be signaled.

The **number of bytes available** is set by the instruction to indicate the actual number of bytes available to be returned. In no case does the instruction return more bytes of information than those available.

The format of **returned information** is described in the following paragraphs.

Basic Activation Group Attributes: The following information is returned when operand 3 is hex 00.

- Root program — System pointer or null
- Reserved (binary 0) — Char(16)
- Storage address recycling key — System pointer or null
- Activation group name — Char(30)
- Reserved (binary 0) — Char(2)
- Activation group mark — UBin(4)
- Reserved (binary 0) — Char(4)
- Heap space count — UBin(4)
- Activation count — UBin(4)
- Static storage size — UBin(4)
- Automatic storage size — UBin(4)
- Attributes — Char(1)
  - Reserved — Bit 0
  - Activation group state. — Bit 1

    0 = user
    1 = system
  - Is activation group named? — Bit 2

    0 = no
    1 = yes
  - Destroy pending? — Bit 3

    0 = no
    1 = yes
  - Reserved (binary 0) — Bits 4-7
- Process access group membership advisory attributes — Char(1)
  - Automatic storage — Bit 0

    0 = do not create in PAG
    1 = permit creation in PAG
  - Static storage — Bit 1

    0 = do not create in PAG
    1 = permit creation in PAG
  - Default heap storage — Bit 2

    0 = do not create in PAG
    1 = permit creation in PAG
  - Reserved (binary 0) — Bits 3-7

Additional Description

root program
those activation groups which are created by the machine (the default activation groups) do not have root programs, in which case this field is null.

storage address recycling key
a system pointer is returned only if the *activation group state* is specified as *user*, otherwise the field is null.

activation group name
For activation groups which do not have a symbolic name, this field contains all blanks.

heap space count
This is the number of heap spaces currently associated with the activation group.

activation count
This is the number of programs which are currently active within the activation group.

static storage size
This is the maximum amount of static storage, in machine dependent units, which has been allocated to the activation group at any particular time. Note that this does not necessarily reflect the amount of storage currently in use.

automatic storage size
This is the maximum amount of automatic storage, in machine dependent units, which has been allocated to the activation group at any particular time. Note that this does not necessarily reflect the amount of storage currently in use. It is merely a measure of the maximum "depth" the automatic storage stack has had to this point.

is activation group named?
Indicates whether the activation group is named or unnamed. The *activation group name* field contains blanks for unnamed an activation groups. The default activation groups and those created with the "unnamed" attribute are unnamed.

Activation Group Heap List: When operand 3 is hex 01, the format of the returned information is an array of heap identifiers. This is a list of the heaps which are currently associated with the activation group. The *heap space count* (available in the basic template) indicates how many entries must be accommodated by the template. The format of the list is:

- array(1..*heap space count*) of Bin(4)

Information about a specific heap may be obtained from the Materialize Heap Space Attributes (MATHSAT) instruction.

Program Activation List: When operand 3 is hex 02, the format of the returned information is an array of *activation marks*. Each activation mark represents a program activation within the activation group. (The activation mark is a number which uniquely identifies the activation within a process.) The *activation count* (available in the basic template) indicates how many entries must be accommodated by the template. The format of the list is:

- array(1..*activation count*) of UBin(4)

Information about a specific activation may be obtained from the Materialize Activation Attributes (MATACTAT) instruction.

Exceptions

Exception		Operands 1	2	3	Other
06	Addressing				
	01 space addressing violation	X	X	X	

Exception		Operands 1	2	3	Other
	02 boundary alignment	X	X	X	
	03 range	X	X	X	
	06 optimized addressability invalid	X	X	X	
08	Argument/parameter				
	01 parameter reference violation	X	X	X	X
10	Damage encountered				
	04 system object damage state	X	X	X	X
	05 authority verification terminated due to damaged object				X
	44 partial system object damage	X	X	X	X
1C	Machine-dependent exception				
	03 machine storage limit exceeded				X
	04 object storage limit exceeded		X		
	09 auxiliary storage pool number invalid		X		
20	Machine support				
	02 machine check				X
	03 function check				X
24	Pointer specification				
	01 pointer does not exist	X	X	X	X
	02 pointer type invalid	X	X	X	X
	03 Pointer addressing invalid object		X	X	
32	Scalar specification				
	03 Scalar value invalid				
2C	Program execution				
	13 Activation group not found		X		
38	Template specification				
	03 Materialization length	X			
44	Domain specification				
	01 Domain Violation	X			

Modify Automatic Storage Allocation (MODASA)

Op Code (Hex)	Operand 1	Operand 2
02F2	Storage allocation	Modification size

Operand 1: Space pointer data object or null.

Operand 2: Signed binary scalar.

ILE access

```
MODASA (
  modification_size   : signed binary; OR
                        unsigned binary;
) : space pointer /* storage_allocation  */
```

Description: The automatic storage frame (ASF) of the current invocation is extended or truncated by the size specified by operand 2. A positive value indicates that the frame is to be *extended*; a negative value indicates that the frame is to be *truncated*; a zero value does not change the ASF. If operand 1 is not null, it will be treated as follows:

- ASF extension: receives the address of the first byte of the extension. The ASF extension might *not* be contiguous with the remainder of the ASF allocation.
- ASF truncation: Operand 1 should be null for truncation. If operand 1 is not null, then addressablity to the first byte of the deallocated space is returned. This value should not be used as a space pointer since it locates space that has been deallocated.
- If a value of zero is specified for operand 2: the value returned is unpredictable.

When the ASF is extended, the extension is aligned on a 16-byte boundary. An extension is not initialized.

A *scalar value invalid* (hex 3203) exception is signaled if truncation amount would cause size of the ASF to be less than the amount of the initial allocation.

A space pointer machine object cannot be specified for operand 1.

Exceptions

Exception		Operands 1	Operands 2	Other
06	Addressing			
	01 space addressing violation	X	X	
	02 boundary alignment	X	X	
	03 range	X	X	
	06 optimized addressability invalid	X	X	
08	Argument/parameter			
	01 parameter reference violation	X	X	
10	Damage encountered			
	04 system object damage state	X	X	X
	44 partial system object damage	X	X	X

Exception		Operands 1	2	Other
1C	Machine-dependent exception			
	03 machine storage limit exceeded			X
20	Machine support			
	02 machine check			X
	03 function check			X
22	Object access			
	01 object not found	X	X	
	02 object destroyed	X	X	
	03 object suspended	X	X	
24	Pointer specification			
	01 pointer does not exist	X	X	
	02 pointer type invalid	X	X	
	08 object compressed			X
2C	Program execution			
	1D automatic storage exceeded			X
2E	Resource control limit			
	01 user profile storage limit exceeded			X
32	Scalar specification			
	01 scalar type invalid	X	X	
	02 scalar attributes invalid		X	
	03 scalar value invalid		X	

Return External (RTX)

Op Code (Hex)	Operand 1
02A1	Return point

Operand 1: Signed binary (2) scalar or null.

Description: The instruction terminates execution of the invocation in which the instruction is specified. The automatic storage frame is deallocated.

A Return External instruction can be specified within an invocation's subinvocation, and no exception is signaled.

If a higher invocation exists in the invocation hierarchy, the instruction causes execution to resume in the preceding invocation in the process' invocation hierarchy at an instruction location indirectly specified by operand 1. If operand 1 is binary 0 or null, the next instruction following the Call External instruction from which control was relinquished in the preceding invocation in the hierarchy is given execution control. If the value of operand 1 is not 0, the value represents an index into the instruction definition list (IDL) specified as the return list operand in the Call External instruction, and the value causes control to be passed to the instruction referenced by the corresponding IDL entry. The first IDL entry is referenced by a value of one. If operand 1 is not 0 and no return list was specified in the Call External instruction, or if the value of operand 1 exceeds the number of entries in the IDL, or if the value is negative, a *return point invalid* (hex 2C02) exception is signaled.

If a higher invocation does not exist, the Return External instruction causes termination of the current process state. If operand 1 is not 0 and is not null, the *return point invalid* (hex 2C02) exception is signaled. Refer to the Terminate Process instruction for the functions performed in process termination.

If the returning invocation has received control to process an event, then control is returned to the point where the event handler was invoked. In this case, if operand 1 is not 0 and is not null, then a *return point invalid* (hex 2C02) exception is signaled.

If the returning invocation has received control from the machine to process an exception, the *return instruction invalid* (hex 2C02) exception is signaled.

If the returning invocation has an activation, the invocation count in the activation is decremented by 1.

If the returning invocation currently has an invocation exit set, the invocation exit is not given control and is implicitly cleared.

Exceptions

Exception		Operand 1	Other
06	Addressing		
	01 space addressing violation	X	
	02 boundary alignment	X	
	03 range	X	
	06 optimized addressability invalid	X	
08	Argument/parameter		
	01 parameter reference violation		X

Exception		Operand 1	Operand Other
10	Damage encountered		
	04 system object damage state	X	X
	44 partial system object damage	X	X
1C	Machine-dependent exception		
	03 machine storage limit exceeded	X	
20	Machine support		
	02 machine check		X
	03 function check		X
22	Object access		
	01 object not found	X	
	02 object destroyed	X	X
	03 object suspended	X	
	08 object compressed		X
24	Pointer specification		
	01 pointer does not exist	X	X
	02 pointer type invalid	X	X
2C	Program execution		
	01 return instruction invalid		X
	02 return point invalid	X	
2E	Resource control limit		
	01 user profile storage limit exceeded		X
36	Space management		
	01 space extension/truncation		X

Set Argument List Length (SETALLEN)

Op Code (Hex)	Operand 1	Operand 2
0242	Argument list	Length

Operand 1: Operand list.

Operand 2: Binary scalar.

Description This instruction specifies the number of arguments to be passed on a succeeding Call External or Transfer Control instruction. The current length of the variable-length operand list (used as an argument list) specified by operand 1 is modified to the value indicated in the binary scalar specified by operand 2. This length value specifies the number of arguments (starting from the first) to be passed from the list when the operand list is referenced on a Call External or Transfer Control instruction.*:* Only variable-length operand lists with the argument list attribute may be modified by the instruction.

The value in operand 2 may range from 0 (meaning no arguments are to be passed) to the maximum size specified in the ODT definition of the operand list (meaning all defined arguments are to be passed).

The length of the argument list remains in effect for the duration of the current invocation or until a Set Argument List Length instruction is issued against this operand list.

Exceptions

Exception		Operands 1	2	Other
06	Addressing			
	01 space addressing violation		X	
	02 boundary alignment		X	
	03 range		X	
	06 optimized addressability invalid		X	
08	Argument/parameter			
	01 parameter reference violation		X	
	03 argument list length modification violation	X		
10	Damage encountered			
	04 system object damage state	X	X	X
	44 partial system object damage	X	X	X
1C	Machine-dependent exception			
	03 machine storage limit exceeded	X		
20	Machine support			
	02 machine check			X
	03 function check			X
22	Object access			
	01 object not found		X	
	02 object destroyed		X	

Exception		Operands 1	2	Other
	03 object suspended		X	
	08 object compressed			X
24	Pointer specification			
	01 pointer does not exist		X	
	02 pointer type invalid		X	
2E	Resource control limit			
	01 user profile storage limit exceeded			X
32	Scalar specification			
	03 scalar value invalid	X		
36	Space management			
	01 space extension/truncation			X

Set Invocation Exit (SETIEXIT)

Op Code (Hex)	Operand 1	Operand 2
0252	Invocation exit program	Argument list

Operand 1: System pointer

Operand 2: Operand list or null

Description: This instruction allows the external entry point of the program specified by operand 1 to be given control when the requesting invocation is destroyed due to normal exception handling actions, or due to any process termination. Normal exception handling actions are considered to be those actions performed by the Return From Exception (RTNEXCP) or the Signal Exception (SIGEXCP) instructions.

Operand 1 is a system pointer addressing the program that is to receive control. The operand 1 system pointer must be in either the static or automatic storage of the program invoking this instruction.

Operand 2 specifies an operand list that identifies the arguments to be passed to the invocation exit program being called. If operand 2 is null, no arguments are passed to the invocation.

No operand verification takes place when this instruction is executed. Nor are copies made of the operands, so changes made to the operand values after execution of this instruction will be used during later operand verification. Operand verification occurs on the original form of the operands when the invocation exit program is invoked. At that time operational authorization to the invocation exit program and retrieve authorization to any contexts referenced for materialization take place. Also, materialization lock enforcement occurs to contexts referenced for materialization.

If an invocation exit program currently exists for the requesting invocation, it is replaced, and no exception is signaled. The invocation exit set by this instruction is implicitly cleared when the invocation exit program is given control, or the program which set the invocation exit completes execution.

If any invocations are to be destroyed due to normal exception handling actions, then those invocation exit programs associated with the invocations to be destroyed are given control before execution proceeds to the signaled exception handler.

A *failure to invoke program* (hex 0011,04,01) event is signaled when both of the following conditions occur:

- Exception management is destroying an invocation stack due to a Signal Exception instruction, a Return From Exception instruction, or process termination.
- An invocation exit program is to be destroyed due to a second Signal Exception or a second Return From Exception instruction.

The invocation exit program that is being destroyed is terminated, and its associated invocation execution is terminated. Termination of invocations due to a previous Signal Exception instruction, a Return From Exception instruction, or a process termination is then resumed.

If a process phase is terminated and the process was not in termination phase, then the invocations are terminated. Invocation exit programs set for the terminated invocations are allowed to run. If an invocation to be terminated is an invocation exit program, then the following occurs:

- A *failure to invoke program* (hex 0011,04,01) event is signaled.
- If an invocation exit has been set for this invocation exit, it is allowed to run.

- The invocation exit is terminated and the associated invocation is terminated (the invocation exit is not reinvoked).

Invocation exit programs for the remaining invocations to be terminated are then allowed to run.

Exceptions

Exception		Operands 1	2	Other
06	Addressing			
	01 space addressing violation	X		
	02 boundary alignment	X		
	03 range	X		
	06 optimized addressability invalid	X		
08	Argument/parameter			
	01 parameter reference violation	X		
10	Damage encountered			
	04 system object damage state			X
	44 partial system object damage			X
1C	Machine-dependent exception			
	03 machine storage limit exceeded			X
20	Machine support			
	02 machine check			X
	03 function check			X
22	Object access			
	08 object compressed			X
2E	Resource control limit			
	01 user profile storage limit exceeded			X
32	Scalar specification			
	01 scalar type invalid	X		
36	Space management			
	01 space extension/truncation			X

Store Parameter List Length (STPLLEN)

Op Code (Hex)	Operand 1
0241	Length

Operand 1: Binary variable scalar.

Description: A value is returned in operand 1 that represents the number of parameters associated with the invocation's external entry point for which arguments have been passed on the preceding Call External or Transfer Control instruction.

The value can range from 0 (no parameters were received) to the maximum size possible for the parameter list associated with the external entry point.

Exceptions

Exception		Operand 1	Other
06	Addressing		
	01 space addressing violation	X	
	02 boundary alignment	X	
	03 range	X	
	06 optimized addressability invalid	X	
08	Argument/parameter		
	01 parameter reference violation	X	
10	Damage encountered		
	04 system object damage state	X	X
	44 partial system object damage	X	X
1C	Machine-dependent exception		
	03 machine storage limit exceeded	X	
20	Machine support		
	02 machine check		X
	03 function check		X
22	Object access		
	01 object not found	X	
	02 object destroyed	X	
	03 object suspended	X	
	08 object compressed		X
24	Pointer specification		
	01 pointer does not exist	X	
	02 pointer type invalid	X	
2E	Resource control limit		
	01 user profile storage limit exceeded		X
32	Scalar specification		

Exception		Operand 1	Other
	01 scalar type invalid	X	
	02 scalar attributes invalid	X	
36	Space management		
	01 space extension/truncation		X

Transfer Control (XCTL)

Op Code (Hex)	Operand 1	Operand 2
0282	Program to be called or Call template	Argument list

Operand 1: System pointer or Space pointer Data Object.

Operand 2: Operand list or null.

Description: The instruction destroys the calling invocation and passes control to either the *program entry procedure* of a bound program or the external entry point of a non-bound program. If operand 1 specifies a bound program that does not contain a *program entry procedure*, an *invalid operation for program* (hex 2C15) exception is signaled.

Operand 1 may be specified as a system pointer which directly addresses the program that is to receive control or as a space pointer to a call template which identifies the program to receive control. Specifying a template allows for additional controls over how the specified program is to be invoked. The format of the call template is the following:

- Call options — Char(4)
  - Suppress adopted user profiles — Bit 0
    - 0 = no
    - 1 = yes
  - Reserved (binary 0) — Bit 1-30
  - Force program state to user state for transfer — Bit 31
    - 0 = no
    - 1 = yes
- Reserved (binary 0) — Char(12)
- Program to be called — System Pointer

The **suppress adopted user profiles** option specifies whether or not the program adopted and propagated user profiles which may be serving as sources of authority to the process are to be suppressed from supplying authority to the new invocation. Specifying *yes* causes the propagation of adopted user profiles to be stopped as of the calling invocation, thereby, not allowing the called invocation to benefit from their authority. Specifying *no* allows the normal propagation of adopted and propagated user profiles to occur. The called program may adopt its owning user profile, if necessary, to supplement the authority available to its invocation.

The **force program state to user state for transfer** option specifies whether or not the transfer control needs to be done in the current program state or as though the transfering program were running in the user state without the transferring program changing to run in the user state.

If the transferring invocation has an activation, the invocation count is decremented by 1.

Common Program Call Processing: The details of processing differ for non-bound and bound programs. The following outlines the common steps.

1. A check is made to determine if the caller has authority to invoke the program and that the object is indeed a program object. The specified program must be either a bound program that contains a *program entry procedure* or a non-bound program.

2. The activation group in which the program is to be run is located or created if it doesn't exist. The activation group supplies the storage resources required for program execution: static, automatic, and heap storage.

3. If the program requires an activation entry and it is not already active within the appropriate activation group, it is activated. Bound programs always require an activation; non-bound programs require an activation only if they use static storage. The *invocation count* of a newly created activation is set to 1 while the *invocation count* of an existing activation is incremented by 1.

4. The invocation created for the target program has the following attributes (as would be reported via the Materialize Invocation Attributes instruction.)

   - the *invocation mark* is one higher than the current mark count value maintained for the process. If the program was activated as a result of the call then its invocation mark will equal the activation mark, otherwise the invocation mark is larger than the activation mark.

     Note: The so-called mark counts are generated from a counter maintained for the process. Each time a mark is required the counter is incremented. The mark counts thus form a non-decreasing sequence of unique identifiers which can be used to distinguish the time ordering of activations, invocations, and activation groups.

   - the *invocation number* is the same as the invocation number of the transferring invocation.

   - the *invocation type* is hex 02 to indicate a XCTL type of invocation.

5. The automatic storage frame (ASF), if required, is allocated on a 16-byte boundary.

6. Control is transferred to the program entry procedure (or external entry point) of the program.

7. Normal flow-of-control resume at the instruction following the caller of the program issuing the XCTL instruction.

The details of locating the target activation group and activating the program differ depending upon the model of the program.

Bound Program: A bound program is activated and run in an activation group specified by program attributes. There are two logical steps involved:

- locate the existing, or create a new activation group for the program
- locate an existing, or create a new activation entry for the program within the activation group

After locating the activation entry for the program, control is passed to the program entry procedure for the program. If required, the activation group is destroyed when the invocation for the program entry procedure is destroyed.

Non-bound Program: The automatic storage frame begins with a 64 byte header. if the program defines no automatic data items the frame consists solely of the 64-byte header, otherwise the automatic storage items are located immediately following the header. In prior releases of the machine, this header contained invocation information which is now available via the Materialize Invocation Attributes (MATINVAT) instruction. This header is not initialized and the contents of the header are not used by the machine. (The space is allocated merely to provide for compatibility with prior implementations of the machine.) The *update PASA stack* program attribute, supported in prior implementations of the machine, is no longer meaningful and is ignored, if specified as an attribute of the program.

Operand 2 specifies an operand list that identifies the arguments to be passed to the invocation to which control is being transferred. Automatic objects allocated by the transferring invocation are destroyed as a result of the transfer operation and, therefore, cannot be passed as arguments. A *parameter list length* (hex 0802) exception is signaled if the number of arguments passed does not correspond to the number required by the parameter list of the target program.

If the transferring invocation has received control to process an exception, or an invocation exit. the *return instruction invalid* (hex 2C01) exception is signaled.

If the transferring invocation currently has an invocation exit set, the invocation exit is not given control and is implicitly cleared.

Authorization Required

- Operand 1
  - Operational
- Retrieve
  - Contexts referenced for address resolution

Lock Enforcement

- Materialize
  - Contexts referenced for address resolution

Exceptions

Exception		Operands 1	2	Other
06	Addressing			
	01 space addressing violation	X		
	02 boundary alignment	X		
	03 range	X		
	06 optimized addressability invalid	X		
08	Argument/parameter			
	01 parameter reference violation			X
	02 parameter list length violation		X	
0A	Authorization			
	01 unauthorized for operation	X		
10	Damage encountered			
	04 system object damage state	X	X	X
	05 authority verification terminated due to damaged object			X
	44 partial system object damage	X	X	X
1A	Lock state			
	01 invalid lock state	X		
1C	Machine-dependent exception			
	02 program limitation exceeded			X
	03 machine storage limit exceeded			X
20	Machine support			
	02 machine check			X
	03 function check			X
22	Object access			

Exception		Operands 1	2	Other
	01 object not found	X		
	02 object destroyed	X		
	03 object suspended	X		
	07 authority verification terminated due to destroyed object			X
	08 object compressed			X
24	Pointer specification			
	01 pointer does not exist	X		
	02 pointer type invalid	X		
	03 pointer addressing invalid object	X		
2C	Program execution			
	01 return instruction invalid			X
	15 invalid operation for program	X		
	1D automatic storage overflow			X
	1E activation access violation			X
	1F program signature violation			X
	20 static storage overflow			X
	21 program import invalid			X
	22 data reference invalid			X
	23 imported object invalid			X
	24 activation group export conflict			X
	25 import not found			X
2E	Resource control limit			
	01 user profile storage limit exceeded			X
36	Space management			
	01 space extension/truncation			X
38	Template specification			
	01 template value invalid	X		

Chapter 10. Program Creation Control Instructions

This chapter describes all the instructions used to control the create program function. These instructions are arranged in alphabetic order. For an alphabetic summary of all the instructions, see Appendix A, "Instruction Summary."

No Operation (NOOP) 10-3
No Operation and Skip (NOOPS) 10-4
Override Program Attributes (OVRPGATR) 10-5

© Copyright IBM Corp. 1991, 1993

No Operation (NOOP)

Op Code (Hex)
0000

Description: No function is performed. The instruction consists of an operation code and no operands. The instruction may not be branched to and is not counted as an instruction in the instruction stream. The instruction may be used for inserting gaps in the instruction stream. These gaps allow instructions with adjacent instruction addresses to be physically separated.

The instruction may precede or follow any machine instruction except the End instruction, and any number of No Operation instructions may exist in succession.

No Operation and Skip (NOOPS)

Op Code (Hex)	**Operand 1**
0001	Skip count

Operand 1: Unsigned immediate value.

Description: This instruction performs no function other than to indicate a specific number of bytes within the instruction stream that are to be skipped during encapsulation. It consists of an operation code and 1 operand. Operand 1 is an unsigned immediate value that contains the number of bytes between this instruction and the next instruction to be processed. These bytes are skipped during the encapsulation of this program. A value of zero for operand 1 indicates that no bytes are to be skipped between this instruction and the next instruction to be processed.

If the operand 1 skip count indicates that the next instruction to process is beyond the end of the instruction stream, an invalid operand value range exception is signaled.

This instruction may be used to insert gaps in the instruction stream in such a manner that allows instructions with adjacent instruction addresses to not be physically adjacent.

This instruction may not be branched to, and is not counted as an instruction in the instruction stream.

The instruction may precede or follow any machine instruction except the End instruction, and any number of No Operation and Skip instructions may exist in succession.

Note: When this instruction is used in an existing program template, the following items within the template may be adversely affected:

- The actual count of instructions may be altered to be different than the count of instructions that is specified in the program template header.
- Object definitions that reference instructions may now be out of range or may not reference the intended instruction.

The actual number of bytes skipped includes the bytes containing the instruction plus the number of bytes specified by the skip count value. The number of bytes skipped per template version is as follows:

- Version 0 = 4 plus the skip count value.
- Version 1 = 5 plus the skip count value.

Override Program Attributes (OVRPGATR)

Op Code (Hex)	Operand 1	Operand 2
0006	Attribute identification	Attribute modifier

Operand 1: Unsigned immediate value.

Operand 2: Unsigned immediate value.

Description: This program creation control instruction allows one of a set of program attributes specified below to be overridden. The overridden program attribute is in effect until it is changed by another OVRPGATR instruction. The initial program attributes are set to the ones given in the program template for the Create Program (CRTPG) instruction. These same initial program attributes are the ones that are materialized when a Material Program (MATPG) is done. That is, the OVRPGATR instruction has no effect on the materialized attributes.

The OVRPGATR instruction consists of an operation code and two operands. Operand 1 is an unsigned immediate value that contains a representation of which program attribute is to be overridden. Operand 2 is an unsigned immediate value that contains a representation of how the program attribute is to be overridden.

This instruction may not be branched to, and is not counted as an instruction in the instruction stream.

The instruction may precede or follow any machine instruction.

The program attributes defined by operand 1 is overridden according to the following selection values:

Attribute Identification	Attribute Description
1	Array constrainment attribute Allowed values for operand 2: 1 = Constrain array references 2 = Do not constrain array references 3 = Fully unconstrain array references 4 = Terminate override of array constrainment attributes and resume use of the attributes specified in the program template
2	String constrainment attribute Allowed values for operand 2: 1 = Constrain string references 2 = Do not constrain string references 3 = Terminate override of string constrainment attribute and resume use of the attribute specified in the program template
3	Suppress binary size exception attribute Allowed values for operand 2: 1 = Suppress binary size exceptions 2 = Do not suppress binary size exceptions 3 = Terminate override of suppression of binary size exception attribute and resume use of the attribute specified in the program template

4	Suppress decimal data exception attribute Allowed values for operand 2: 1 = Suppress decimal data exceptions 2 = Do not suppress decimal data exceptions 3 = Terminate override of suppression of decimal data exception attribute and resume use of the attribute specified in the program template
5	Copy Bytes with Pointers (CPYBWP) alignment data check attribute Allowed values for operand 2: 1 = Constrain CPYBWP to require like alignment of operands (default) 2 = Do not constrain CPWBWP to require like alignment of operands
6	Compare Pointer for Space Addressibility (CMPPSPAD) null pointer tolerance attribute Allowed values for operand 2: 1 = Signal pointer does not exist exceptions for operands 1 and 2 (default) 2 = Do not signal pointer does not exist exceptions for operands 1 and 2

Chapter 11. Independent Index Instructions

This chapter describes the instructions used for indexes. These instructions are in alphabetic order. For an alphabetic summary of all the instructions, see Appendix A, "Instruction Summary."

Create Independent Index (CRTINX) 11-3
Destroy Independent Index (DESINX) 11-10
Find Independent Index Entry (FNDINXEN) 11-12
Insert Independent Index Entry (INSINXEN) 11-16
Materialize Independent Index Attributes (MATINXAT) 11-19
Modify Independent Index (MODINX) 11-23
Remove Independent Index Entry (RMVINXEN) 11-26

© Copyright IBM Corp. 1991, 1993

Create Independent Index (CRTINX)

Op Code (Hex)	Operand 1	Operand 2
0446	Index	Index description template

Operand 1: System pointer.

Operand 2: Space pointer.

ILE access

```
CRTINX (
  var index                : system pointer;
      description_template : space pointer
)
```

Description: This instruction creates an independent index based on the index template specified by operand 2 and returns addressability to the index in a system pointer stored in the addressing object specified by operand 1.

The format of the index description template described by operand 2 is as follows (must be aligned on a 16-byte multiple):

- Template size specification — Char(8)
  - Number of bytes provided for materialization — Bin(4)*
  - Number of bytes available for materialization — Bin(4)*
- Object identification — Char(32)
  - Object type — Char(1)*
  - Object subtype — Char(1)
  - Object name — Char(30)
- Object creation options — Char(4)
  - Existence attributes — Bit 0

    0 = Temporary
    1 = Permanent
  - Space attribute — Bit 1

    0 = Fixed-length
    1 = Variable-length
  - Initial context — Bit 2

    0 = Do not insert addressability in context
    1 = Insert addressability in context
  - Access group — Bit 3

    0 = Do not create as member of access group
    1 = Create as member of access group
  - Reserved (binary 0) — Bits 4-12
  - Initialize space — Bit 13

    0 = Initialize

1 = Do not initialize

- Reserved (binary 0) — Bits 14-31

- Recovery options — Char(4)
  - Reserved (binary 0) — Char(2)
  - ASP number — Char(2)
- Size of space — Bin(4)
- Initial value of space — Char(1)
- Performance class — Char(4)
  - Space alignment — Bit 0

    0 = The space associated with the object is allocated to allow proper alignment of pointers at 16-byte alignments within the space. If no space is specified for the object, 0 must be specified for the performance class.
    1 = The space associated with the object is allocated to allow proper alignment of pointers at 16-byte alignments within the space as well as to allow proper alignment of input/output buffers at 512-byte alignments within the the space.

  - Reserved (binary 0) — Bits 1-4
  - Main storage pool selection — Bit 5

    0 = Process default main storage pool is used for object.
    1 = Machine default main storage pool is used for object.

  - Reserved (binary 0) — Bit 6
  - Block transfer on implicit access state modification — Bit 7

    0 = Transfer the minimum storage transfer size for this object. This value is 1 storage unit.
    1 = Transfer the machine default storage transfer size. This value is 8 storage units.

  - Reserved (binary 0) — Bits 8-31
- Reserved (binary 0) — Char(3)
- Extension offset — Bin(4)
- Context — System pointer
- Access group — System pointer
- Index attributes — Char(1)
  - Entry length attribute — Bit 0

    0 = Fixed-length entries
    1 = Variable-length entries

  - Immediate update — Bit 1

    0 = No immediate update
    1 = Immediate update

  - Key insertion — Bit 2

    0 = No insertion by key
    1 = Insertion by key

  - Entry format — Bit 3

    0 = Scalar data only
    1 = Both pointers and scalar data

  - Optimized processing mode — Bit 4

0 = Optimize for random references
1 = Optimize for sequential references

- Maximum entry length — Bit 5

  0 = Maximum entry length is 120 bytes
  1 = Maximum entry length is 2000 bytes

- Reserved (binary 0) — Bits 6-7

- Argument length — Bin(2)
- Key length — Bin(2)

Note: This instruction ignores the values associated with the entries annotated with an asterisk (*).

The template identified by operand 2 must be 16-byte aligned.

The template extension is located by the **extension offset** field. The template extension must be 16-byte aligned in the space. The following is the format of the template extension:

- Reserved (binary 0) — Char(20)
- Domain assigned to the object — Char(2)
- Reserved (binary 0) — Char(42)

If the created object is permanent, it is owned by the user profile governing process execution. The owning user profile is implicitly assigned all private authority states for the object. The storage occupied by the created object is charged to this owning user profile. If the created object is temporary, there is no owning user profile, and all authority states are assigned as public. Storage occupied by the created context is charged to the creating process.

The **object identification** specifies the symbolic name that identifies the space within the machine. An **object type** of hex 0E is implicitly supplied by the machine. The *object identification* is used to identify the object on materialize instructions as well as to locate the object in a context that addresses the object.

The **existence attribute** specifies that the index is to be created as a *permanent* or a *temporary* object. A temporary index, if not explicitly destroyed by the user, is implicitly destroyed by the machine when machine processing is terminated.

A space may be associated with the created object. The space may be *fixed* or *variable* in size, as specified by the **space attribute** field. The initial allocation is as specified in the **size of space** field. The machine allocates a space of at least the size specified. The actual size allocated is dependent on an algorithm defined by a specific implementation.

If the **initial context** creation attribute field indicates that *addressability is to be inserted* in a context, the **context** field must be a system pointer that identifies a context where addressability to the newly created object is to be placed. If the *initial context* indicates that *addressability is not to be placed in a context*, the *context* field is ignored.

If the **access group** creation attribute field indicates that the object *is to be created in an access group*, the **access group** field must be a system pointer that identifies an access group in which the object is to be created. The *existence attribute* of the object must be identical to the *existence attribute* of the access group. If the object *is not to be created in the access group*, the *access group* field is ignored.

The **initialize space** creation option controls whether or not the space is to be initialized. When *initialize* is specified, each byte of the space is initialized to a value specified by the **initial value of space** field. Additionally, when the space is extended in size, this byte value is also used to initialize the new allocation. When *do not initialize* is specified, the *initial value of space* field is ignored and the initial value of the bytes of the space are unpredictable.

When *do not initialize* is specified for a space, internal machine algorithms do ensure that any storage resources last used for allocations to another object which are reused to satisfy allocations for the space are reset to a machine default value to avoid possible access of data which may have been stored in the other object. To the contrary, reusage of storage areas previously used by the space object are not reset. thereby exposing subsequent reallocations of those storage areas within the space to access of the data which was previously stored within them.

The **ASP number** attribute specifies the ASP number of the ASP on which the unit is to be allocated. A value of 0 indicates an *ASP number* is not specified and results in the default of allocating the object in the system ASP. Allocation on the system ASP can only be done implicitly by not specifying an *ASP number*. The only nonzero values allowed are 2 through 16 which provide for explicit allocation of objects on user ASPs. The ASP number must specify an existing ASP. The *ASP number* attribute may only be specified for creation of a permanent object. The *ASP number* attribute of an object can be materialized, but cannot be modified.

Invalid specification of the ASP number attribute results in the signaling of the *template value invalid* (hex 3801) exception.

The *preferred unit number* attribute which can be specified in the *performance class* field at object creation may not be specified in conjunction with specification of the *ASP number* attribute.

The **performance class** parameter provides information allowing the machine to more effectively manage the object considering the overall performance objectives of operations involving the index.

If the **entry length** attribute field specifies *fixed-length entries*, the entry length of every index entry is established at creation by the value in the **argument length** field of the index description template. If the *entry length* attribute field specifies *variable-length entries*, then entries will be variable-length (the length of each entry is supplied when the entry is inserted), and the *argument length* field is ignored.

If the **immediate update** field specifies that an *immediate update* should occur, then every update to the index will be written to auxiliary storage after every insert or remove operation.

If the **key insertion** field specifies *insertion by key*, then the **key length** field must be specified. This allows the specification of a portion of the argument (the key), which may be manipulated in either of the following ways in the Insert Index Entry instruction:

- The insert will not take place if the key portion of the argument is already in the index.
- The insert will cause the nonkey portion of the argument to be replaced if the key is already in the index.

The **entry format** field designates the index entries as containing *both pointers and scalar data* or *scalar data only*. The *both pointers and scalar data* field can be used only for indexes with fixed-length entries. If the index is created to contain both pointers and data, then

- Entries to be inserted must be 16-byte aligned.
- Each entry retrieved by the Find Independent Index Entry instruction or the Remove Independent Index Entry is 16-byte aligned.
- Pointers are allowed in both the key and nonkey portions of an index entry.
- Pointers need not be at the same location in every index entry.
- Pointers inserted into the index remain unchanged. No resolution is performed before insertion.

If the index is created to contain *scalar data only*, then:

- Entries to be inserted need not be aligned.
- Entries returned by the Find Independent Index Entry instruction or the Remove Independent Index Entry instruction are not aligned.

- Any pointers inserted into the index will be invalidated.

The **optimized processing mode** index attribute field is used to designate whether the index should be created and maintained in a manner that optimizes performance for either *random* or *sequential* operations.

If the **maximum entry length** attribute field specifies that the maximum entry length is 2000 bytes, then the maximum length allowed for independent index entries will be 2000 bytes. Otherwise, the maximum entry length allowed will be 120 bytes.

The **key length** field specifies the length of the key for the entries that are inserted into the index. The **argument length** specifies the length of the the entries when *fixed length* entries are used.

The *key length* must have a value less than or equal to the *argument length* whether specified during creation (for fixed-length entries) or during insertion (for variable length). The *key length* is not used if the *key insertion field* specifies *no insertion by key*.

The **extension offset** specifies the byte offset from the beginning of the operand 2 template to the beginning of the template extension. An offset value of zero specifies that the template extension is not provided. A negative offset value is invalid. A non-zero offset must be a multiple of 16 (to cause 16 byte alignment of the extension). Except for these restrictions, the offset value is not verified for correctness relative to the location of other portions of the create template.

The **domain assigned** field in the template extension allows the user of this instruction to override the domain for this object that would otherwise be chosen by the machine. Valid values for this field are:

Domain field	Domain assigned to the object
Hex 0000	The domain will be chosen by the machine.
Hex 0001	The domain will be 'Common User'.
Hex 8000	The domain will be 'Common System'.

Any value specified for the *domain assigned* field other than those listed above will result in a *template value invalid* (hex 3801) exception being signalled.

Limitations: The following are limits that apply to the functions performed by this instruction.

The size of the object specific portion of the object is limited to a maximum of 4 gigabytes. This size is dependent upon the amount of storage needed for the number and size of index entries and excludes the size of the associated space, if any.

The size of the associated space for this object is limited to a maximum of 16MB-32 bytes.

Authorization Required

- Insert
  - Context identified by operand 2
  - User profile of object owner
- Retrieve
  - Contexts referenced for address resolution

Lock Enforcement

- Modify
  - Access group identified by operand 2
  - User profile of object owner
  - Context identified by operand 2
- Materialize
  - Contexts referenced for address resolution

Exceptions

Exception		Operands 1	2	Other
02	Access group			
	01 object ineligible for access group		X	
	02 object exceeds available space		X	
06	Addressing			
	01 space addressing violation	X	X	
	02 boundary alignment	X	X	
	03 range	X	X	
	06 optimized addressability invalid	X	X	
08	Argument/parameter			
	01 parameter reference violation	X	X	
0A	Authorization			
	01 unauthorized for operation		X	
0E	Context			
	01 duplicate object identification		X	
10	Damage encountered			
	04 system object damage state	X	X	X
	05 authority verification terminated due to damaged object			X
	44 partial system object damage	X	X	X
1A	Lock state			
	01 invalid lock state		X	
1C	Machine-dependent exception			
	03 machine storage limit exceeded			X
	04 object storage limit exceeded		X	
	09 auxiliary storage pool number invalid		X	
20	Machine support			
	02 machine check			X
	03 function check			X
22	Object access			
	01 object not found	X	X	

Exception		Operands 1	Operands 2	Other
	02 object destroyed	X	X	
	03 object suspended	X	X	
	07 authority verification terminated due to destroyed object			X
	08 object compressed			X
24	Pointer specification			
	01 pointer does not exist	X	X	
	02 pointer type invalid	X	X	
	03 pointer addressing invalid object		X	
2E	Resource control limit			
	01 user profile storage limit exceeded		X	
36	Space management			
	01 space extension/truncation			X
38	Template specification			
	01 template value invalid		X	

Destroy Independent Index (DESINX)

Op Code (Hex)	Operand 1
0451	Index

Operand 1: System pointer.

ILE access

```
DESINX (
  var index : system pointer
)
```

Description: A previously created index identified by operand 1 is destroyed, and addressability to the object is removed from any context in which addressability exists. The system pointer identified by operand 1 is not modified by the instruction, and a subsequent reference to the destroyed index through the pointer results in an object destroyed exception.

Authorization Required

- Object control
  - Operand 1
- Retrieve
  - Contexts referenced for address resolution

Lock Enforcement

- Materialize
  - Contexts referenced for address resolution
- Object control
  - Operand 1
- Modify
  - Access group which contains operand 1
  - Context which addresses operand 1
  - User profile which owns index

Exceptions

Exception		Operand 1	Other
06	Addressing		
	01 space addressing violation	X	
	02 boundary alignment	X	
	03 range	X	
	06 optimized addressability invalid	X	
08	Argument/parameter		
	01 parameter reference violation	X	

Exception		Operand 1	Operand Other
0A	Authorization		
	01 unauthorized for operation	X	
10	Damage encountered		
	04 system object damage state	X	X
	05 authority verification terminated due to damaged object		X
	44 partial system object damage	X	X
1A	Lock state		
	01 invalid lock state	X	
1C	Machine-dependent exception		
	03 machine storage limit exceeded		X
20	Machine support		
	02 machine check		X
	03 function check		X
22	Object access		
	01 object not found	X	
	02 object destroyed	X	
	03 object suspended	X	
	07 authority verification terminated due to destroyed object		X
	08 object compressed		X
24	Pointer specification		
	01 pointer does not exist	X	
	02 pointer type invalid	X	
	03 pointer addressing invalid object	X	
2E	Resource control limit		
	01 user profile storage limit exceeded		X
36	Space management		
	01 space extension/truncation		X

Find Independent Index Entry (FNDINXEN)

Op Code (Hex)	Operand 1	Operand 2	Operand 3	Operand 4
0494	Receiver	Index	Option list	Search argument

Operand 1: Space pointer.

Operand 2: System pointer.

Operand 3: Space pointer.

Operand 4: Space pointer.

ILE access

```
FNDINXEN (
        receiver        : space pointer;
    var index           : system pointer;
        option_list     : space pointer;
        search_argument : space pointer
)
```

Warning: The following information is subject to change from release to release. Use it with caution and be prepared to adjust for changes with each new release.

Description: This instruction searches the independent index identified by operand 2 according to the search criteria specified in the option list (operand 3) and the search argument (operand 4); then it returns the desired entry or entries in the receiver field (operand 1). The maximum size of the independent index entry is either 120 bytes or 2000 bytes depending on how the maximum entry length attribute field was specified when the index was created.

The **option list** is a variable-length area that identifies the type of search to be performed, the length of the search argument(s), the number of resultant arguments to be returned, the lengths of the entries returned, and the offsets to the entries within the receiver identified by the operand 1 space pointer. The *option list* has the following format:

- Rule option — Char(2)
- Argument length — UBin(2)
- Argument offset — Bin(2)
- Occurrence count — Bin(2)
- Return count — Bin(2)

Each entry that is returned to the receiver operand contains the following:

- Entry length — UBin(2)
- Offset — Bin(2)

The **rule option** identifies the type of search to be performed and has the following meaning:

Search Type	Value (Hex)	Meaning
=	0001	Find equal occurrences of operand 4.
>	0002	Find occurrences that are greater than operand 4.

Search Type	Value (Hex)	Meaning
<	0003	Find occurrences that are less than operand 4.
≥	0004	Find occurrences that are greater than or equal to operand 4.
≤	0005	Find occurrences that are less than or equal to operand 4.
First	0006	Find the first index entry or entries.
Last	0007	Find the last index entry or entries.
Between	0008	Find all entries between the two arguments specified by operand 4 (inclusive).

The *rule option* to find *between* requires that operand 4 be a 2-element array in which element 1 is the starting argument and element 2 is the ending argument. All arguments between (and including) the starting and ending arguments are returned, but the *occurrence count* specified is not exceeded.

If the index was created to contain *both pointers and scalar data*, then the search argument must be 16-byte aligned. For the option to find between limits, both search arguments must be 16-byte aligned.

The *rule option* and the *argument length* determine the search criteria used for the index search. The *argument length* must be greater than or equal to one. The *argument length* for fixed-length entries must be less than or equal to the *argument length* specified when the index is created.

The **argument length** field specifies the length of the search argument (operand 4) to be used for the index search. When the *rule option* equals *first* or *last*, the *argument length* field is ignored. For the *rule option* to find *between*, the *argument length* field specifies the lengths of one array element. The lengths of the array elements must be equal.

The **argument offset** is the offset of the second search argument from the beginning of the entire argument field (operand 4). The *argument offset* field is ignored unless the *rule option* is find *between*.

The **occurrence count** specifies the maximum number of index entries that satisfy the search criteria to be returned. This field is limited to a maximum value of 4095. If this value is exceeded, a *template value invalid* (hex 3801) exception is signaled.

The **return count** specifies the number of index entries satisfying the search criteria that were returned in the receiver (operand 1). If this field is 0, no index arguments satisfied the search criteria.

There are two fields in the option list for each entry returned in the receiver (operand 1). The **entry length** is the length of the entry retrieved from the index. The **offset** has the following meaning:

- For the first entry, the *offset* is the number of bytes from the beginning of the receiver (operand 1) to the first byte of the first entry.
- For any succeeding entry, the *offset* is the number of bytes from the beginning of the immediately preceding entry to the first byte of the entry returned.

The entries that are retrieved as a result of the Find Independent Index Entry instruction are always returned starting with the entry that is closest to or equal to the search argument and then proceeding away from the search argument. For example, a search that is for < (less than) or ≤ (less than or equal to) returns the entries in order of decreasing value.

All the entries that satisfy the search criteria (up to the occurrence count) are returned in the space starting at the location designated by the operand 1 space pointer.

If the index was created to contain *both pointers and scalar data*, then each returned entry is 16-byte aligned.

If the index was created to contain *scalar data only*, then returned entries are contiguous.

Every entry retrieved causes the count of the find operations to be incremented by 1. The current value of this count is available through the Materialize Index Attributes instruction.

Authorization Required

- Retrieve
  - Operand 2
  - Contexts referenced for address resolution

Lock Enforcement

- Materialize
  - Operand 2
  - Contexts referenced for address resolution

Exceptions

Exception		Operands 1	2	3	4	Other
06	Addressing					
	01 space addressing violation	X	X	X	X	
	02 boundary alignment	X	X	X	X	
	03 range	X	X	X	X	
	06 optimized addressability invalid	X	X	X	X	
08	Argument/parameter					
	01 parameter reference violation	X	X	X	X	
0A	Authorization					
	01 unauthorized for operation		X			
10	Damage encountered					
	04 system object damage state	X	X	X	X	X
	05 authority verification terminated due to damaged object					X
	44 partial system object damage	X	X	X	X	X
1A	Lock state					
	01 invalid lock state		X			
1C	Machine-dependent exception					
	03 machine storage limit exceeded					X
20	Machine support					
	02 machine check					X
	03 function check					X
22	Object access					
	01 object not found	X	X	X	X	
	02 object destroyed	X	X	X	X	
	03 object suspended	X	X	X	X	

Exception		Operands 1	2	3	4	Other
	07 authority verification terminated due to destroyed object					X
	08 object compressed					X
24	Pointer specification					
	01 pointer does not exist	X	X	X	X	
	02 pointer type invalid	X	X	X	X	
	03 pointer addressing invalid object		X			
2E	Resource control limit					
	01 user profile storage limit exceeded					X
36	Space management					
	01 space extension/truncation					X
38	Template specification					
	01 template value invalid			X		
	02 template size invalid			X		

Insert Independent Index Entry (INSINXEN)

Op Code (Hex)	Operand 1	Operand 2	Operand 3
04A3	Index	Argument	Option list

Operand 1: System pointer.

Operand 2: Space pointer.

Operand 3: Space pointer.

ILE access

```
INSINXEN (
 var index       : system pointer;
     argument    : space pointer;
     option_list : space pointer
)
```

Warning: The following information is subject to change from release to release. Use it with caution and be prepared to adjust for changes with each new release.

Description: This instruction inserts one or more new entries into the independent index identified by operand 1 according to the criteria specified in the option list (operand 3). Each entry is inserted into the index at the appropriate location based on the binary value of the argument. No other collating sequence is supported. The maximum length allowed for the independent index entry is either 120 bytes or 2000 bytes depending on how the *maximum entry length* attribute field was specified when the index was created.

The argument (operand 2) and the option list (operand 3) have the same format as the argument and option list for the Find Independent Index Entry instruction.

The **rule option** identifies the type of insert to be performed and has the following meaning:

Insert Type	Value (Hex)	Meaning	Authorization
Insert	0001	Insert unique argument	Insert
Insert with replacement	0002	Insert argument, replacing the nonkey portion if the key is already in the index	Update
Insert without replacement	0003	Insert argument only if the key is not already in the index	Insert

The *insert rule option* is valid only for indexes not containing keys. The *insert with replacement rule option* and the *insert without replacement rule option* are valid for indexes containing either fixed- or variable-length entries with keys. The *duplicate key argument* (hex 1801) exception is signaled for the following conditions:

- If the *rule option* is *insert* and the argument to be inserted (operand 2) is already in the index
- If the *rule option* is *insert without replacement* and the key portion of the argument to be inserted (operand 2) is already in the index

The *argument length* and *argument offset* fields are ignored, however, the *entry length* and *offset* fields must be entered for every entry which is to be inserted into the index.

The __occurrence count__ specifies the number of arguments to be inserted. This field is limited to a maximum value of 4095. If this value is exceeded, a *template value invalid* (hex 3801) exception is signaled.

If the index was created to contain *both pointers and data*, then each entry to be inserted must be 16-byte aligned. If the index was created to contain *variable-length* entries, then the *entry length* and *offset fields* must be specified in the option list for each argument in the space identified by operand 2. The *entry length* is the length of the entry to be inserted.

If the index was created to contain *both pointer and scalar data*, the *offset* field in the option list must be supplied for each entry to be inserted. The *offset* is the number of bytes from the beginning of the previous entry to the beginning of the entry to be inserted. For the first entry, this is the offset from the start of the space identified by operand 2.

The *return count* specifies the number of entries inserted into the index. If the index was created to contain *scalar data only*, then any pointers inserted are invalidated.

Authorization Required

- Insert or update depending on insert type
  - Operand 1
- Retrieve
  - Contexts referenced for address resolution

Lock Enforcement

- Materialize
  - Contexts referenced for address resolution
- Modify
  - Operand 1

Exceptions

Exception		Operands 1	2	3	Other
02	Access group				
	01 object exceeds available space	X			
06	Addressing				
	01 space addressing violation	X	X	X	
	02 boundary alignment	X	X	X	
	03 range	X	X	X	
	06 optimized addressability invalid	X	X	X	
08	Argument/parameter				
	01 parameter reference violation	X	X	X	
0A	Authorization				
	01 unauthorized for operation	X			
10	Damage encountered				
	04 system object damage state	X	X	X	X

Exception		Operands 1	2	3	Other
	05 authority verification terminated due to damaged object				X
	44 partial system object damage	X	X	X	X
18	Independent index				
	01 duplicate key argument in index	X			
1A	Lock state				
	01 invalid lock state	X			
1C	Machine-dependent exception				
	03 machine storage limit exceeded				X
	04 object storage limit exceeded	X			
20	Machine support				
	02 machine check				X
	03 function check				X
22	Object access				
	01 object not found	X	X	X	
	02 object destroyed	X	X	X	
	03 object suspended	X	X	X	
	07 authority verification terminated due to destroyed object				X
	08 object compressed				X
24	Pointer specification				
	01 pointer does not exist	X	X	X	
	02 pointer type invalid	X	X	X	
	03 pointer addressing invalid object	X			
2E	Resource control limit				
	01 user profile storage limit exceeded	X			
36	Space management				
	01 space extension/truncation				X
38	Template specification				
	01 template value invalid			X	
	02 template size invalid			X	

Materialize Independent Index Attributes (MATINXAT)

Op Code (Hex)	Operand 1	Operand 2
0462	Receiver	Index

Operand 1: Space pointer.

Operand 2: System pointer.

ILE access

```
MATINXAT (
     receiver : space pointer;
 var index    : system pointer
)
```

Warning: The following information is subject to change from release to release. Use it with caution and be prepared to adjust for changes with each new release.

Description: The instruction materializes the creation attributes and current operational statistics of the independent index identified by operand 2 into the space identified by operand 1. The format of the attributes materialized is as follows:

- Materialization size specification — Char(8)
  - Number of bytes provided for materialization — Bin(4)
  - Number of bytes available for materialization — Bin(4)
- Object identification — Char(32)
  - Object type — Char(1)
  - Object subtype — Char(1)
  - Object name — Char(30)
- Object creation options — Char(4)
  - Existence attributes — Bit 0

    0 = Temporary
    1 = Reserved
  - Space attribute — Bit 1

    0 = Fixed-length
    1 = Variable-length
  - Context — Bit 2

    0 = Addressability not in context
    1 = Addressability in context
  - Access group — Bit 3

    0 = Not a member of access group
    1 = Member of access group
  - Reserved (binary 0) — Bits 4-12
  - Initialize space — Bit 13
  - Reserved (binary 0) — Bits 14-31

- Reserved (binary 0) — Char(4)
- Size of space — Bin(4)
- Initial value of space — Char(1)
- Performance class — Char(4)
  - Space alignment — Bit 0

    0 = The space associated with the object is allocated to allow proper alignment of pointers at 16-byte alignments within the space. If no space is specified for the object, this value must be specified for the performance class.
    1 = The space associated with the object is allocated to allow proper alignment of pointers at 16-byte alignments within the space as well as to allow proper alignment of input/output buffers at 512-byte alignments within the the space.
  - Reserved (binary 0) — Bits 1-4
  - Main storage pool selection — Bit 5

    0 = Process default main storage pool used for object.
    1 = Machine default main storage pool used for object.
  - Reserved (binary 0) — Bit 6
  - Block transfer on implicit access state modification — Bit 7

    0 = The minimum storage transfer size for this object is a value of 1 storage unit.
    1 = The machine default storage transfer size for this object is a value of 8 storage units.
  - Reserved (binary 0) — Bits 8-31
- Reserved (binary 0) — Char(7)
- Context — System pointer
- Access group — System pointer
- Index attributes — Char(1)
- Argument length — Bin(2)
- Key length — Bin(2)
- Index statistics — Char(12)
  - Entries inserted — UBin(4)
  - Entries removed — UBin(4)
  - Find operations — UBin(4)

The first 4 bytes of the materialization identify the total **number of bytes provided** that may be used by the instruction. This value is supplied as input to the instruction and is not modified by the instruction. A value of less than 8 causes the *materialization length* (hex 3803) exception to be signaled.

The second 4 bytes of the materialization identify the total **number of bytes available** to be materialized. The instruction materializes as many bytes as can be contained in the area specified as the receiver. If the byte area identified by the receiver is greater than that required to contain the information requested, then the excess bytes are unchanged.

No exceptions other than the *materialization length* (hex 3803) exception described previously are signaled in the event that the receiver contains insufficient area for the materialization.

The template identified by the operand 1 space pointer must be 16-byte aligned. Values in the template remain the same as the values specified at the creation of the independent index except that the *object identification, context*, and *size of the associated space* contain current values.

If the *entry length* is *fixed*, then the *argument length* is the value supplied in the template when the index was created. If the *entry length* is *variable*, then the *argument length* field is equal to the length of the longest entry that has ever been inserted into the index.

The number of arguments in the index equals the number of **entries inserted** minus **entries removed**. The value of the **find operations** field is initialized to 0 each time the index is materialized. The value may not be correct after an abnormal system termination.

Authorization Required

- Operational
  - Operand 2
- Retrieve
  - Contexts referenced for address resolution

Lock Enforcement

- Materialize
  - Operand 2
  - Contexts referenced for address resolution

Exceptions

Exception		Operands 1	2	Other
06	Addressing			
	01 space addressing violation	X	X	
	02 boundary alignment	X	X	
	03 range	X	X	
	06 optimized addressability invalid	X	X	
08	Argument/parameter			
	01 parameter reference violation	X	X	
0A	Authorization			
	01 unauthorized for operation		X	
10	Damage encountered			
	04 system object damage state	X	X	X
	05 authority verification terminated due to damaged object			X
	44 partial system object damage	X	X	X
1A	Lock state			
	01 invalid lock state		X	
1C	Machine-dependent exception			
	03 machine storage limit exceeded			X

Exception		Operands 1	Operands 2	Other
22	Object access			
	01 object not found	X	X	
	02 object destroyed	X	X	
	03 object suspended	X	X	
	07 authority verification terminated due to destroyed object			X
	08 object compressed			X
24	Pointer specification			
	01 pointer does not exist	X	X	
	02 pointer type invalid	X	X	
	03 pointer addressing invalid object		X	
2E	Resource control limit			
	01 user profile storage limit exceeded			X
36	Space management			
	01 space extension/truncation			X
38	Template specification			
	01 template value invalid		X	
	03 materialization length exception	X		

Modify Independent Index (MODINX)

Op Code (Hex)	Operand 1	Operand 2
0452	Independent index	Modification option

Operand 1: System pointer.

Operand 2: Character (4) scalar.

ILE access

```
MODINX (
  var index                  : system pointer;
  var modification_option    : aggregate
)
```

Warning: The following information is subject to change from release to release. Use it with caution and be prepared to adjust for changes with each new release.

Description: This instruction modifies the selected attributes of the independent index specified by operand 1 to have the values specified in operand 2. The modification options specified in operand 2 have the following format:

- Modification selection — Char(1)
  - Reserved (binary 0) — Bit 0
  - Immediate update — Bit 1

    0 = Do not change immediate update attribute
    1 = Change immediate update attribute
  - Reserved (binary 0) — Bits 2-7
- New attribute value — Char(1)
  - Reserved (binary 0) — Bit 0
  - Immediate update — Bit 1

    0 = No immediate update
    1 = Immediate update
  - Reserved (binary 0) — Bits 2-7
- Reserved (binary 0) — Char(2)

If the **modification selection immediate update** is 0, then the *immediate update* attribute is not changed. If the modification selection *immediate update* bit is 1, the *immediate update* attribute is changed to the **new immediate update attribute** value.

If the *immediate update* attribute of the index was previously set to *no immediate update*, and it is being modified to *immediate update*, then the index is ensured before the attribute is modified.

Authorization Required

- Object management
  - Operand 1
- Retrieve

- Contexts referenced for address resolution

Lock Enforcement

- Modify
  - Operand 1
- Materialization
  - Contexts referenced for address resolution

Exceptions

Exception		Operands 1	Operands 2	Other
06	Addressing			
	01 space addressing violation	X	X	
	02 boundary alignment	X	X	
	03 range	X	X	
	06 optimized addressability invalid	X	X	
08	Argument/parameter			
	01 parameter reference violation	X	X	
0A	Authorization			
	01 unauthorized for operation	X		
10	Damage encountered			
	04 system object damage	X		X
	05 authority verification terminated due to damaged object			X
	44 partial system object damage			X
1A	Lock state			
	01 invalid lock state	X		
1C	Machine-dependent exception			
	03 machine storage limit exceeded			X
20	Machine support			
	02 machine check			X
	03 function check			X
22	Object access			
	01 object not found	X	X	
	02 object destroyed	X	X	
	03 object suspended	X	X	
	07 authority verification terminated due to destroyed object			X
	08 object compressed			X
24	Pointer specification			
	01 pointer does not exist	X	X	
	02 pointer type invalid	X	X	
	03 pointer address invalid object	X		

Exception		Operands 1	2	Other
2E	Resource control limit			
	01 user profile storage limit exceeded			X
32	Scalar specification			
	01 scalar type invalid	X	X	
	02 scalar attributes invalid		X	
36	Space management			
	01 space extension/truncation			X

Remove Independent Index Entry (RMVINXEN)

Op Code (Hex)	Operand 1	Operand 2	Operand 3	Operand 4
0484	Receiver	Index	Option list	Argument

Operand 1: Space pointer or null.

Operand 2: System pointer.

Operand 3: Space pointer.

Operand 4: Space pointer.

ILE access

```
RMVINXEN (
      receiver    : space pointer; OR
                    null operand;
  var index       : system pointer;
      option_list : space pointer;
      argument    : space pointer
)
```

Warning: The following information is subject to change from release to release. Use it with caution and be prepared to adjust for changes with each new release.

Description: The index entries identified by operands 3 and 4 are removed from the independent index identified by operand 2 and optionally returned in the receiver specified by operand 1. The maximum length of an independent index entry is either 120 bytes or 2000 bytes depending on how the *maximum entry length* attribute field was specified when the index was created.

The option list (operand 3) and the argument (operand 4) have the same format and meaning as the option list and argument for the Find Independent Index Entry instruction. The **return count** designates the number of index entries that were removed from the index.

The arguments removed are returned in the receiver field if a space pointer is specified for operand 1. If operand 1 is null, the entries removed from the index are not returned. If neither space pointer nor null is specified for operand 1, the entries are returned in the same way that entries are returned for the Find Independent Index Entry instruction.

Every entry removed causes the *occurrence count* to be incremented by 1. The current value of this count is available through the Materialize Index Attributes instruction. The *occurrence count* field must be less then 4096.

Authorization Required

- Delete
  - Operand 2
- Retrieve
  - Contexts referenced for address resolution

Lock Enforcement

- Materialize
  - Contexts referenced for address resolution
- Modify
  - Operand 2

Exceptions

Exception		Operands 1	2	3	4	Other
02	Access group					
	01 object exceeds available space		X			
06	Addressing					
	01 space addressing violation	X	X	X	X	
	02 boundary alignment	X	X	X	X	
	03 range	X	X	X	X	
	06 optimized addressability invalid	X	X	X	X	
08	Argument/parameter					
	01 parameter reference violation	X	X	X	X	
0A	Authorization					
	01 unauthorized for operation		X			
10	Damage encountered					
	04 system object damage state	X	X	X	X	X
	05 authority verification terminated due to damaged object					X
	44 partial system object damage	X	X	X	X	X
1A	Lock state					
	01 invalid lock state		X			X
1C	Machine-dependent exception					
	03 machine storage limit exceeded					X
	04 object storage limit exceeded		X			
20	Machine support					
	02 machine check					X
	03 function check					X
22	Object access					
	01 object not found	X	X	X	X	
	02 object destroyed	X	X	X	X	
	03 object suspended	X	X	X	X	
	07 authority verification terminated due to destroyed object					X
	08 object compressed					X
24	Pointer specification					
	01 pointer does not exist	X	X	X	X	

Exception		Operands 1	2	3	4	Other
	02 pointer type invalid	X	X	X	X	
	03 pointer addressing invalid object		X			
2E	Resource control limit					
	01 user profile storage limit exceeded		X			
36	Space management					
	01 space extension/truncation					X
38	Template specification					
	01 template value invalid			X		
	02 template size invalid			X		

Chapter 12. Queue Management Instructions

This chapter describes the instructions used for queue management. These instructions are in alphabetic order. For an alphabetic summary of all the instructions, see Appendix A, "Instruction Summary."

Dequeue (DEQ) 12-3
Enqueue (ENQ) 12-9
Materialize Queue Attributes (MATQAT) 12-12
Materialize Queue Messages (MATQMSG) 12-16

© Copyright IBM Corp. 1991, 1993

Dequeue (DEQ)

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand 3	Operand 4-5
DEQ 1033		Message prefix	Message text	Queue or queue template	
DEQB 1C33	Branch options	Message prefix	Message text	Queue or queue template	Branch targets
DEQI 1833	Indicator options	Message prefix	Message text	Queue or queue template	Indicator targets

Operand 1: Character variable scalar (fixed-length).

Operand 2: Space pointer.

Operand 3: System pointer or space pointer data object.

Operand 4-5:

- *Branch Form*-Branch point, instruction pointer, relative instruction number, or absolute instruction number.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

ILE access

```
DEQ (
  var message_prefix : aggregate;
      message_text   : space pointer;
  var queue          : system pointer;
) : signed binary /* return_code */
```

If a message is not available, this instruction will return immediately with the return code set as follows:

Return_Code	Meaning
1	Message Dequeued.
0	Message Not Dequeued.

OR

```
DEQWAIT (
  var msg_prefix : aggregate;
      msg_text   : space pointer;
  var queue      : system pointer
)
```

If a message is not available, this instruction will wait until a message is available.

Description: The instruction retrieves a queue message based on the *queue type* (FIFO, LIFO, or keyed) specified during the queue's creation. If the queue was created with the *keyed* option, messages can be retrieved by any of the following relationships between an enqueued *message key* and a *selection key* specified in operand 1 of the Dequeue instruction: $\neq$, $>$, $<$, $\leq$, and $\geq$. If the queue was created with either the *LIFO* or *FIFO* attribute, then only the next message can be retrieved from the queue.

If a message is not found that satisfies the dequeue selection criterion and the branch or options are not specified, the process waits until a message arrives to satisfy the dequeue or until the *dequeue wait time-out* expires. If branch or indicator options are specified, the process is not placed in the dequeue wait state and either the control flow is altered according to the branch options, or indicator values are set based on the presence or absence of a message to be dequeued.

If operand 3 is a system pointer, the message is dequeued from the queue specified by operand 3. If operand 3 is a space pointer, the message is dequeued from the queue which is specified in the template pointed to by the space pointer. The format of this template is given later in this section. The criteria for message selection are given in the message prefix specified by operand 1. The message text is returned in the space specified by operand 2, and the message prefix is returned in the scalar specified by operand 1. Improper alignment results in an exception being signaled. The format of the message prefix is as follows:

- Timestamp of enqueue of message — Char(8)**
- Dequeue wait time-out value (ignored if branch options specified) — Char(8)*
- Size of message dequeued (The maximum allowable size of a queue message is 65 000 bytes.) — Bin(4)**
- Access state modification option indicator and message selection criteria — Char(1)*
  - Access state modification option when entering Dequeue wait — Bit 0*

    0 = Access state is not modified
    1 = Access state is modified
  - Access state modification option when leaving Dequeue wait — Bit 1*

    0 = Access state is not modified
    1 = Access state is modified
  - Multiprogramming level option — Bit 2*

    0 = Leave current MPL set at Dequeue wait
    1 = Remain in current MPL set at Dequeue wait
  - Time-out option — Bit 3*

    0 = Wait for specified time, then signal time-out exception
    1 = Wait indefinitely
  - Actual key to input key relationship (for keyed queue) — Bits 4-7*

    0010: Greater than
    0100: Less than
    0110: Not equal
    1000: Equal
    1010: Greater than or equal
    1100: Less than or equal

• Search key (ignored for FIFO/LIFO queues but must be present for FIFO/LIFO queues with nonzero key length values)	Char(key length)*
• Message key	Char(key length)**

Note: Fields shown here with one asterisk indicate input to the instruction, and fields shown here with two asterisks are returned by the machine.

A nonzero **dequeue wait time-out** value overrides any *dequeue wait time-out* value specified as the current process attribute. A zero *dequeue wait time-out* value causes the wait time-out value to be taken from the current process attribute. If all wait time-out values are 0 (from the Dequeue instruction and the current process attribute), an *immediate wait time-out* (hex 3A01) exception is signaled. The bits in this field are numbered from 0 to 63, and bit 41 is defined as 1024 microseconds. The maximum *dequeue wait time-out* interval allowed is a value equal to (2^{48} - 1) microseconds. Any value that indicates more time than the maximum wait time-out causes the maximum wait time-out to be used.

The **size of the message dequeued** is returned in the message prefix. The *size of the message dequeued* can be less than or equal to the *maximum size of message* specified when the queue was created. When dequeuing from a keyed queue, the length of the *search key* field and the length of the *message key* field (in the message key prefix specified in operand 1) are determined implicitly by the attributes of the queue being accessed. If the message text on the queue contains pointers, the message text operand must be 16-byte aligned.

The **access state** of the process access group is modified when a Dequeue instruction results in a wait and the following conditions exist:

- The process' *instruction wait initiation access state* control attribute specifies allow access state modification
- The *dequeue access state modification* option specifies *modify access state*
- The *multiprogramming level option* specifies *leave MPL* set during wait.

The process will remain in the current MPL set for a maximum of two seconds when a Dequeue instruction results in a wait if the *multiprogramming level option* specifies *remain in current MPL set at Dequeue wait* and the *access state modification when entering Dequeue wait* option specifies *do not modify access state*. After two seconds, the process will automatically be removed from the current MPL set. The automatic removal does not change or affect the total wait time specified for the process by the *Dequeue wait time-out* value.

Operand 3 can be a system pointer or a space pointer. If it is a system pointer, this pointer will be addressing the queue from which the message is to be dequeued. If it is a space pointer, this pointer will be addressing a template which will contain the system pointer to the queue as well as the Dequeue template extension. The template is 32 bytes in length and must be aligned on a 16-byte boundary with the format as follows:

• Queue	System pointer
• Dequeue template extension	Char(16)
— Extension Options	Char(1)
- Modify process event mask option 0 = Do not modify process event mask 1 = Modify process event mask	Bit 0 *
- Reserved (binary 0)	Bits 1-7
— Extension Area	Char(15)

- New process event mask — Bin(2) *
- Previous process event mask — Bin(2) **
- Reserved (binary 0) — Char(11)

The previous process event mask is only returned when the modify process event mask option has been set to 1.

Note: Fields shown here with one asterisk indicate input to the instruction, and fields shown here with two asterisks are returned by the machine.

The **modify process event mask** option controls the state of the event mask in the process executing this instruction. If the *modify process event mask* field specified to *modify the process event mask*, the process event mask will be changed as specified by the **new process event mask** field. When the process event mask is changed, the current process event mask will be returned in the **previous process event mask** field.

If the system security level machine attribute is hex 40 or greater and the process is running in user state, then the *modify process event mask option* is not allowed and a *template value invalid* (hex 3801) exception is signalled.

If the *process event mask* is in the *masked* state, the machine does not schedule signaled event monitors in the process. The event monitors continue to be signaled by the machine or other processes. When the process is modified to the *unmasked* state, event handlers are scheduled to handle those events that occurred while the process was masked and those events occurring while in the unmasked state. The number of signals retained while the process is masked is specified by the attributes of the event monitor associated with the process.

The process is automatically masked by the machine when event handlers are invoked. If the process is unmasked in the event handler, other events can be handled if another enabled event monitor within that process is signaled. If the process is masked when it exits from the event handler, the machine explicitly unmasks the process.

Valid masking values are:

0 Masked
256 Unmasked

Other values are reserved and must not be specified. If any other values are specified, a *template value invalid* (hex 3801) exception is signaled.

Whether masking or unmasking the current process, the new mask takes effect upon completion of a satisfied dequeue.

Resultant Conditions

- Equal - message dequeued
- Not equal - message not dequeued

Authorization Required

- Retrieve
  - Operand 3
  - Contexts referenced for address resolution

Lock Enforcement

- Materialize
  - Contexts referenced for address resolution

Exceptions

Exception		Operands 1	2	3	Other
06	Addressing				
	01 space addressing violation	X	X	X	
	02 boundary alignment	X	X	X	
	03 range	X	X	X	
	06 optimized addressability invalid	X	X	X	
08	Argument/parameter				
	01 parameter reference violation	X	X	X	
0A	Authorization				
	01 unauthorized for operation			X	X
10	Damage encountered				
	04 system object damage state	X	X	X	X
	05 authority verification terminated due to damaged object				X
	44 partial system object damage	X	X	X	X
1A	Lock state				
	01 invalid lock state		X	X	
1C	Machine-dependent exception				
	03 machine storage limit exceeded				X
20	Machine support				
	02 machine check				X
	03 function check				X
22	Object access				
	01 object not found	X		X	
	02 object destroyed	X	X	X	
	03 object suspended	X	X	X	
	07 authority verification terminated due to destroyed object				X
	08 object compressed				X
24	Pointer specification				
	01 pointer does not exist	X	X	X	
	02 pointer type invalid	X	X	X	
	03 pointer address invalid object	X			
2E	Resource control limit				
	01 user profile storage limit exceeded				X

Exception		Operands 1	2	3	Other
32	Scalar specification				
	03 scalar value invalid	X			
36	Space management				
	01 space extension/truncation				X
3A	Wait time-out				
	01 dequeue				X

Enqueue (ENQ)

Op Code (Hex)	Operand 1	Operand 2	Operand 3
036B	Queue	Message prefix	Message text

Operand 1: System pointer.

Operand 2: Character scalar.

Operand 3: Space pointer.

ILE access

```
ENQ (
  var queue          : system pointer;
  var message_prefix : aggregate;
      message_text   : space pointer
)
```

Description: A message is enqueued according to the *queue type* attribute specified during the queue's creation.

If *keyed* sequence is specified, enqueued messages are sequenced in ascending binary collating order according to the key value. If a message to be enqueued has a key value equal to an existing enqueued key value, the message being added is enqueued following the existing message.

If the queue was defined with either *last in, first out (LIFO)* or *first in, first out (FIFO)* sequencing, then enqueued messages are ordered chronologically with the latest enqueued message being either first on the queue or last on the queue, respectively. A key can be provided and associated with messages enqueued in a LIFO or FIFO queue; however, the key does not establish a message's position in the queue. The key can contain pointers, but the pointers are not considered to be pointers when they are placed on the queue by an Enqueue instruction.

Operand 1 specifies the queue to which a message is to be enqueued. Operand 2 specifies the message prefix, and operand 3 specifies the message text.

The format of the message prefix is as follows:

• Size of message to be enqueued	Bin(4)
• Enqueue key value (Ignored for FIFO/LIFO queues with key lengths equal to 0.	Char(key length)

The **size of the message to be enqueued** is supplied to inform the machine of the number of bytes in the space that are to be considered message text. The size of the message is then considered the lesser of the *size of the message to be enqueued* attribute and the *maximum message size* specified on queue creation. The message text can contain pointers. When pointers are in message text, the operand 3 space pointer must be 16-byte aligned. Improper alignment will result in an exception being signaled.

If the enqueued message causes the number of messages to exceed the maximum number of messages attribute of the queue, one of the following occurs:

- If the queue is not extendable, the *queue message limit exceeded* (hex 2602) exception and the *queue message limit exceeded* (hex 0012,03,01) event are signaled. The message is not enqueued.

- If the queue is extendable, the queue is implicitly extended by the *extension value* attribute. The message is enqueued. No exception is signaled, but the *queue extended* (hex 0012,04.01) event is signaled.

The maximum allowable queue size, including all messages currently enqueued and the machine overhead, is 16 megabytes.

Authorization Required

- Insert
  - Operand 1
- Retrieve
  - Contexts referenced for address resolution

Lock Enforcement

- Materialize
  - Contexts referenced for address resolution

Exceptions

Exception		Operands 1	2	3	Other
06	Addressing				
	01 space addressing violation	X	X	X	
	02 boundary alignment	X	X	X	
	03 range	X	X	X	
	06 optimized addressability invalid	X	X	X	
08	Argument/parameter				
	01 parameter reference violation	X	X	X	
0A	Authorization				
	01 unauthorized for operation	X			X
10	Damage encountered				
	04 system object damage state	X	X	X	X
	05 authority verification terminated due to damaged object				X
	44 partial system object damage	X	X	X	X
1A	Lock state				
	01 invalid lock state	X			X
1C	Machine-dependent exception				
	03 machine storage limit exceeded	X			X
	04 object storage limit exceeded	X			
20	Machine support				
	02 machine check				X
	03 function check				X
22	Object access				

Exception		Operands 1	2	3	Other
	01 object not found	X	X	X	
	02 object destroyed	X	X	X	
	03 object suspended	X	X	X	
	07 authority verification terminated due to destroyed object				X
	08 object compressed				X
24	Pointer specification				
	01 pointer does not exist	X	X	X	
	02 pointer type invalid	X	X	X	
	03 pointer address invalid object	X			
26	Process management				
	02 queue message limit exceeded	X			
2E	Resource control limit				
	01 user profile storage limit exceeded				X
36	Space management				
	01 space extension/truncation				X

Materialize Queue Attributes (MATQAT)

Op Code (Hex)	Operand 1	Operand 2
0336	Receiver	Queue

Operand 1: Space pointer.

Operand 2: System pointer.

ILE access

```
MATQAT (
      receiver   : space pointer;
  var queue      : system pointer
)
```

Description: The attributes of the queue specified by operand 2 are materialized into the object specified by operand 1. The format of the materialized queue attributes must be aligned on a 16-byte multiple. The format is as follows:

• Materialization size specification	Char(8)
— Number of bytes provided for materialization	Bin(4)
— Number of bytes available for materialization	Bin(4)
• Object identification	Char(32)
— Object type	Char(1)
— Object subtype	Char(1)
— Object name	Char(30)
• Object creation options	Char(4)
— Existence attributes 0 = Temporary 1 = Permanent	Bit 0
— Space attribute 0 = Fixed-length 1 = Variable-length	Bit 1
— Initial context 0 = Addressability not in context 1 = Addressability in context	Bit 2
— Access group 0 = Not a member of access group 1 = Member of access group	Bit 3
— Reserved (binary 0)	Bits 4-12
— Initialize space	Bit 13
— Reserved (binary 0)	Bits 14-31
• Reserved (binary 0)	Char(4)
• Size of space	Bin(4)

- Initial value of space — Char(1)
- Performance class — Char(4)
  - Space alignment — Bit 0

    0 = The space associated with the object is allocated to allow proper alignment of pointers at 16-byte alignments within the space. If no space is specified for the object, this value must be specified for the performance class.
    1 = The space associated with the object is allocated to allow proper alignment of pointers at 16-byte alignments within the space as well as to allow proper alignment of input/output buffers at 512-byte alignments within the the space.
  - Reserved (binary 0) — Bits 1-4
  - Main storage pool selection — Bit 5

    0 = Process default main storage pool is used for object.
    1 = Machine default main storage pool is used for object.
  - Reserved (binary 0) — Bit 6
  - Block transfer on implicit access state modification — Bit 7

    0 = Transfer the minimum storage transfer size for this object. This value is 1 storage unit.
    1 = Transfer the machine default storage transfer size. This value is 8 storage units.
  - Reserved (binary 0) — Bits 8-31
- Reserved (binary 0) — Char(7)
- Context — System pointer
- Access group — System pointer
- Queue attributes — Char(1)
  - Message content — Bit 0

    0 = Contains scalar data only
    1 = Contains pointers and scalar data
  - Queue type — Bits 1-2

    00 = Keyed
    01 = Last in, first out
    10 = First in, first out
  - Queue overflow action — Bit 3

    0 = Signal exception
    1 = Extend queue
  - Reserved (binary 0) — Bits 4-7
- Current maximum number of messages — Bin(4)
- Current number of messages enqueued — Bin(4)
- Extension value — Bin(4)
- Key length — Bin(2)
- Maximum size of message to be enqueued — Bin(4)

The first 4 bytes of the materialization identify the total **number of bytes provided** for use by the instruction. This value is supplied as input to the instruction and is not modified by the instruction. A value of less than 8 causes the *materialization length* (hex 3803) exception.

The second 4 bytes of the materialization identify the total **number of bytes available** to be materialized. The instruction materializes as many bytes as can be contained in the area specified as the receiver. If the byte area identified by the receiver is greater than that required to contain the information requested, then the excess bytes are unchanged. No exceptions (other than the *materialization length* (hex 3803) exception described previously) are signaled when the receiver contains insufficient area for the materialization.

See the Create Queue (CRTQ) instruction for a description of these fields.

Authorization Required

- Operational
  - Operand 2
- Retrieve
  - Contexts referenced for address resolution

Lock Enforcement

- Materialize
  - Operand 2
  - Contexts referenced for address resolution

Exceptions

Exception		Operands 1	2	Other
06	Addressing			
	01 space addressing violation	X	X	
	02 boundary alignment	X	X	
	03 range	X	X	
	06 optimized addressability invalid	X	X	
08	Argument/parameter			
	01 parameter reference violation	X	X	
0A	Authorization			
	01 unauthorized for operation		X	X
10	Damage encountered			
	04 system object damage state	X	X	X
	05 authority verification terminated due to damaged object			X
	44 partial system object damage	X	X	X
1A	Lock state			
	01 invalid lock state		X	X
20	Machine support			
	02 machine check			X
	03 function check			X
22	Object access			
	01 object not found	X	X	

Exception		Operands 1	Operands 2	Other
	02 object destroyed	X	X	
	03 object suspended	X	X	
	07 authority verification terminated due to destroyed object			X
	08 object compressed			X
24	Pointer specification			
	01 pointer does not exist	X	X	
	02 pointer type invalid	X	X	
	03 pointer address invalid object		X	
2E	Resource control limit			
	01 user profile storage limit exceeded			X
36	Space management			
	01 space extension/truncation			X
38	Template specification			
	03 materialization length exception	X		

Materialize Queue Messages (MATQMSG)

Op Code (Hex)	Operand 1	Operand 2	Operand 3
033B	Receiver	Queue	Message selection template

Operand 1: Space pointer.

Operand 2: System pointer.

Operand 3: Character(16) scalar (fixed length).

ILE access

```
MATQMSG (
      receiver              : space pointer;
  var queue                 : system pointer;
  var selection_template    : aggregate
)
```

Description: This instruction materializes selected messages on a queue. One or more messages on the queue specified by operand 2 is selected according to information provided in operand 3 and materialized into operand 1. The number of messages materialized and the amount of key and message text data materialized for each message is governed by the message selection template.

Note that the list of messages on a queue is a dynamic attribute and may be changing on a continual basis. The materialization of messages provided by this instruction is just a picture of the status of the queue at the point of interrogation by this instruction. As such, the actual status of the queue may differ from that described in the materialization when subsequent instructions use the information in the template as a basis for operations against the queue.

Operand 1 specifies a space that is to receive the materialized attribute values.

Operand 2 is a system pointer identifying the queue from which the messages are to be materialized.

Operand 3 is a character (16) scalar specifying which messages are to be materialized.

The operand 1 space pointer must address a 16-byte boundary. The materialization template has the following format:

- Materialization size specification — Char(8)
  - Number of bytes provided for materialization — Bin(4)
  - Number of bytes available for materialization — Bin(4)
- Materialization data — Char(4)
  - Count of messages materialized — Bin(4)
- Queue data — Char(12)
  - Count of messages on the queue — Bin(4)
  - Maximum message size — Bin(4)
  - Key size — Bin(4)
- Reserved — Char(8)
- Message data (repeated for each message) — Char(*)

• Message attributes	Char(16)
– Message enqueue time	Char(8)
– Message length	Bin(4)
– Reserved	Char(4)
• Message key	Char(*)
• Message text	Char(*)

The first 4 bytes of the materialization identify the total **number of bytes provided** for use by the instruction. This value is supplied as input to the instruction and is not modified by the instruction. A value of less than 8 causes the *materialization length* (hex 3803) exception to be signaled.

The second 4 bytes of the materialization identify the total **number of bytes available** to be materialized. The instruction materializes as many bytes as can be contained in the area specified as the receiver. If the byte area identified by the receiver is greater than that required to contain the information requested, then the excess bytes are unchanged. No exceptions are signaled in the event that the receiver contains insufficient area for the materialization, other than the *materialization length* (hex 3803) exception described previously.

The **maximum message size** and **key size** are values specified when the queue was created. If the queue is not a keyed queue, the value materialized for the key size is zero.

The length of the message key and message text fields is determined by values supplied in operand 3, message selection data. If the length supplied in operand 3 exceeds the actual data length, the remaining space will be padded with binary zeros.

The message selection template identified by operand 3 must be at least 16 bytes and must be on a 16-byte boundary. The format of the message selection template is as follows:

• Message selection	Char(2)
– Type 0001 = All messages 0010 = First 0100 = Last 1000 = Keyed All other values are reserved	Bits 0-3
– Key relationship (if needed) 0010 = Greater than 0100 = Less than 0110 = Not equal 1000 = Equal 1010 = Greater than or equal 1100 = Less than or equal All other values are reserved	Bits 4-7
– Reserved	Bits 8-15
• Lengths	Char(8)
– Number of key bytes to materialize	Bin(4)
– Number of message text bytes to materialize	Bin(4)
• Reserved	Char(6)
• Key (if needed)	Char(*)

The <u>**message selection type**</u> must not specify *keyed* if the queue was not created as a keyed queue.

Both of the fields specified under lengths must be zero or an integer multiple of 16. The maximum value allowed for the key length is 256. The maximum value allowed for the message text is 65536.

Authorization Required

- Retrieve
  - Operand 2
  - Contexts referenced for address resolution

Lock Enforcement

- Materialization
  - Operand 2
  - Contexts referenced for address resolution

Exceptions

Exception		Operands 1	2	3	4	Other
06	Addressing					
	01 space addressing violation	X	X	X		
	02 boundary alignment	X	X	X		
	03 range	X	X	X		
	06 optimized addressability invalid	X	X	X		
08	Argument/parameter					
	01 parameter reference violation	X	X	X		
0A	Authorization					
	01 unauthorized for operation		X			
10	Damage encountered					
	04 system object damage state		X			X
	05 authority verification terminated due to damaged object					X
	44 partial system object damage					X
1A	Lock state					
	01 invalid lock state				X	
20	Machine support					
	02 machine check					X
	03 function check					X
22	Object access					
	01 object not found	X	X	X		
	02 object destroyed	X	X	X		
	03 object suspended	X	X	X		
	07 authority verification terminated due to destroyed object					X
	08 object compressed					X

Exception		Operands 1	2	3	4	Other
24	Pointer specification					
	01 pointer does not exist	X	X	X		
	02 pointer type invalid	X	X	X		
	03 pointer address invalid object		X			
28	Process state					
	02 process control space not associated with a process		X			
2E	Resource control limit					
	01 user profile storage limit exceeded					X
32	Scalar specification					
	01 scalar type invalid	X	X	X		
	02 scalar attributes invalid	X	X	X		
	03 scalar value invalid			X		
36	Space management					
	01 space extension/truncation					X
38	Template specification					
	03 materialization length exception	X				

Chapter 13. Object Lock Management Instructions

This chapter describes the lock management instructions. The instructions are in alphabetic order. For an alphabetic summary of all the instructions, see Appendix A, "Instruction Summary."

Lock Object (LOCK) 13-3
Lock Space Location (LOCKSL) 13-8
Materialize Data Space Record Locks (MATDRECL) 13-13
Materialize Process Locks (MATPRLK) 13-17
Materialize Process Record Locks (MATPRECL) 13-20
Materialize Selected Locks (MATSELLK) 13-24
Transfer Object Lock (XFRLOCK) 13-27
Unlock Object (UNLOCK) 13-30
Unlock Space Location (UNLOCKSL) 13-33

© Copyright IBM Corp. 1991, 1993

Lock Object (LOCK)

Op Code (Hex)	Operand 1
03F5	Lock request template

Operand 1: Space pointer.

ILE access

```
LOCK (
  lock_request_template : space pointer
)
```

Description: The instruction requests that locks for system objects identified by system pointers in the space object (operand 1) be allocated to the issuing process. The lock state desired for each object is specified by a value associated with each system pointer in the lock template (operand 1).

The lock request template must be aligned on a 16-byte boundary. The format is as follows:

- Number of lock requests in template — Bin(4)
- Offset to lock state selection values — Bin(2)
- Wait time-out value for instruction — Char(8)
- Lock request options — Char(1)
  - Lock request type — Bits 0-1

    00 = Immediate request- If all locks cannot be immediately granted, signal *lock request not grantable* (hex 1A02) exception.
    01 = Synchronous request- Wait until all locks can be granted.
    10 = Asynchronous request- Allow processing to continue and signal event when the object is available.
  - Access state modifications — Bits 2-3
    - When the process is entering lock wait for synchronous request: — Bit 2

      0 = Access state should not be modified.
      1 = Access state should be modified.
    - When the process is leaving lock wait: — Bit 3

      0 = Access state should not be modified.
      1 = Access state should be modified.
  - Reserved (binary 0) — Bits 4-5*
  - Time-out option — Bit 6

    0 = Wait for specified time, then signal time-out exception.
    1 = Wait indefinitely.
  - Template extension specified — Bit 7

    0 = Template is not specified.
    1 = Template is specified.
- Reserved (binary 0) — Char(1)

The Lock Object template extension is only present if *template extension specified* is indicated above. Otherwise. the *Object(s) to be locked* should immediately follow.

- Lock Object template extension — Char(16)
  - Extension options — Char(1)
    - Modify process event mask option — Bit 0

      0 = Do not modify process event mask
      1 = Modify process event mask
    - Reserved (binary 0) — Bits 1-7
  - Extension area — Char(15)
    - New process event mask — UBin(2)
    - Previous process event mask — UBin(2)
    - Reserved (binary 0) — Char(11)
- Object(s) to be locked — System pointer

  This should be repeated as specified by *number of lock requests in template* above.

The *lock state selection* is located by adding the *offset to lock state selection values* above to operand 1.

- Lock state selection (repeated for each pointer in the template) — Char(1)
  - Requested lock state (1 = lock requested, 0 = lock not requested) — Bits 0-4

    Only one state may be requested.
    - LSRD lock — Bit 0
    - LSRO lock — Bit 1
    - LSUP lock — Bit 2
    - LEAR lock — Bit 3
    - LENR lock — Bit 4
  - Reserved (binary 0) — Bits 5-6*
  - Entry active indicator — Bit 7

    0 = Entry not active - This entry is not used.
    1 = Entry active - Obtain this lock.

Note: Entries indicated with an asterisk are ignored by the instruction.

Lock Allocation Procedure: A single Lock instruction can request the allocation of one or more lock states on one or more objects. Locks are allocated sequentially until all locks requested are allocated.

The **offset to lock state selection** values specifies an offset from the beginning of the lock request. This offset is used to locate the lock state selection values.

The **wait time-out** field establishes the maximum amount of time that a process competes for the requested set of locks when either *lock request type* is either *synchronous* or *asynchronous*. The bits in this field are numbered from 0 to 63, and bit 41 is defined as 1024 microseconds. The maximum wait time-out interval allowed is a value equal to (2^{48} - 1) microseconds. Any value that indicates more time than the maximum wait time-out causes the maximum wait time-out to be used. If the *wait time-out* field is specified with a value of binary 0, then the value associated with the default wait time-out field in the process definition template establishes the time interval.

When a requested lock state cannot be immediately granted, any locks already allocated by this Lock instruction are released, and the **lock request type** specified in the lock request template establishes the machine action. The lock request types are described in the following paragraphs.

- *Immediate Request-* If the requested locks cannot be granted immediately, this option causes the *lock request not grantable* (hex 1A02) exception to be signaled. No locks are granted and the lock request is canceled.
- *Synchronous Request-* This option causes the process requesting the locks to be placed in the wait state until all requested locks can be granted. If the locks cannot be granted in the time interval established by the *wait time-out* field specified in the lock request template, the *lock wait time-out* (hex 3A02) exception is signaled to the requesting process at the end of the interval. No locks are granted, and the lock request is canceled.
- *Asynchronous Request-* This option allows the requesting process to proceed with execution while the machine asynchronously attempts to satisfy the lock request.

When the *synchronous request* option is specified and the requested locks cannot be immediately allocated, the **access state modification** field in the lock request template specifies whether the access state of the process access group is to be modified on entering and/or returning from the lock wait. The field has no effect if the process instruction wait access state control attribute specifies that no access state modification is allowed. If the process attribute value specifies that access state modification is allowed and the wait on event access state modification option specifies modify access state, the machine modifies the access state for the specified process access group.

If the *lock request type* is *synchronous* and the invocation containing the lock instruction is terminated, then the lock request is canceled.

If the lock request is satisfied, then the *object locked* (hex 000A,01,01) event is signaled to the requesting process. If the request is not satisfied in the time interval established by the *wait time-out* field specified in the lock request template, the *asyncronous lock wait timeout* (hex 000A,04,01) event is signaled to the requesting process. No locks are granted, and the lock request is canceled. If an object is destroyed while a process has a pending request to lock the object, the *object destroyed* (hex 000A,02,01) event is signaled to the waiting process.

If the *lock request type* is *asynchronous* and the invocation containing the Lock instruction is terminated, then the lock request remains active.

When two or more processes are competing for a conflicting lock allocation on a system object, the machine attempts to first satisfy the lock allocation request of the process with the highest priority. Within that priority, the machine attempts to satisfy the request that has been waiting longest.

If any exception is identified during the instruction's execution, any locks already granted by the instruction are released, and the lock request is canceled.

For each system object lock, counts are kept by lock state and by process. When a lock request is granted, the appropriate lock count(s) of each lock state specified is incremented by 1.

If a previously unsatisfied lock request is satisfied by the transfer of a lock from another process, the lock request and transfer lock are treated as independent events relative to lock accounting. The appropriate lock counts are incremented for both the lock request and the transfer lock function.

The **modify process event mask option** controls the state of the event mask in the process executing this instruction. If the event mask is in the *masked* state, the machine does not schedule signaled event monitors in the process. The event monitors continue to be signaled by the machine or other processes. When the process is modified to the *unmasked* state, event handlers are scheduled to handle those events that occurred while the process was masked and those events occurring while in

the unmasked state. The number of events retained while the process is masked is specified by the attributes of the event monitor associated with the process.

A lock request with an *asynchronous lock request type* cannot have the *modify process event mask option* set to 1.

| If the system security level machine attribute is hex 40 or greater and the process is running in user
| state, then the *modify process event mask option* is not allowed and a *template value invalid* (hex 3801)
| exception is signalled.

When the *modify process event mask* is set to 1, the **previous process event mask** will be returned and the **new process event mask** will take effect only when the lock(s) have been successfully granted. If the lock request is not successful, the *previous process event mask* value is not returned, nor does the *new process event mask* take effect.

The process is automatically masked by the machine when event handlers are invoked. If the process is unmasked in the event handler, other events can be handled if another enabled event monitor within that process is signaled. If the process is masked when it exits from the event handler, the machine explicitly unmasks the process.

Valid masking values are:

0 Masked
256 Unmasked

Other values are reserved and must not be specified. If any other values are specified, a *template value invalid* (hex 3801) exception is signaled.

Whether masking or unmasking the current process, the new mask takes effect upon completion of a satisfied lock object.

Authorization Required

- Some authority or ownership
  - Objects to be locked
- Retrieve
  - Contexts referenced for address resolution

Lock Enforcement

- Materialize
  - Contexts referenced for address resolution

Exceptions

Exception		Operands 1	Other
06	Addressing		
	01 space addressing violation	X	
	02 boundary alignment	X	
	03 range	X	
	06 optimized addressability invalid	X	
08	Argument/parameter		

Exception		Operands 1	Other
	01 parameter reference violation	X	
0A	Authorization		
	01 unauthorized for operation	X	
10	Damage encountered		
	04 system object damage state	X	X
	05 authority verification terminated due to damaged object		X
	44 partial system object damage	X	X
1A	Lock state		
	01 invalid lock state		X
	02 lock request not grantable	X	
1C	Machine-dependent exception		
	03 machine storage limit exceeded		X
	06 machine lock limit exceeded		X
20	Machine support		
	02 machine check		X
	03 function check		X
22	Object access		
	01 object not found	X	
	02 object destroyed	X	
	03 object suspended	X	
	07 authority verification terminated due to destroyed object		X
	08 object compressed		X
24	Pointer specification		
	01 pointer does not exist	X	
	02 pointer type invalid	X	
2E	Resource control limit		
	01 user profile storage limit exceeded		X
36	Space management		
	01 space extension/truncation		X
38	Template specification		
	01 template value invalid	X	
3A	Wait time-out		
	02 lock		X

Lock Space Location (LOCKSL)

Op Code (Hex)	Operand 1	Operand 2
03F6	Space location or Lock Request Template	Lock request

Operand 1: Space pointer data object.

Operand 2: Char(1) scalar or null.

ILE access

```
LOCKSL (
 var space_location : space pointer;
 var lock_request   : aggregate OR
                      null operand
)
```

Description: When operand 2 is not null, the space location (operand 1) is granted to the issuing process according to the lock request specified by operand 2. When operand 2 is null, the instruction requests that the space locations identified in the lock request template (operand 1) be granted to the issuing process.

Locking a space location does not prevent any byte operation from referencing that location, nor does it prevent the space from being extended, truncated, or destroyed. Space location locks follow the normal locking rules with respect to conflicts and waits but are strictly symbolic in nature.

A space pointer machine object cannot be specified for operand 1.

The following is the format of operand 2 when not null:

• Lock request	Char(1)
— Lock state selection (1 = lock requested, 0 = lock not requested)	Bits 0-4
Only one state may be requested.	
- LSRD lock	Bit 0
- LSRO lock	Bit 1
- LSUP lock	Bit 2
- LEAR lock	Bit 3
- LENR lock	Bit 4
— Reserved (binary 0)	Bits 5-7

For this form, if the requested lock cannot be immediately granted, the process will enter a synchronous wait for the lock for a period of up to the interval specified by the process default time-out value. If the wait exceeds this time limit, a space location lock wait exception is signaled, and the requested lock is not granted.

During the wait, the process access state may be modified. This can occur if the process' *instruction wait access state* control attribute is set to *allow access state modification.*

When operand 2 is null, the lock request template identified by operand 1 must be aligned on a 16-byte boundary. The format of operand 1 is as follows:

• Number of space location lock requests in template	Bin(4)
• Offset to lock state selection values	Bin(2)
• Wait time-out value for instruction	Char(8)
• Lock request options	Char(3)
— Reserved (binary 0)	Bit 0
— Lock request type	Bit 1
0 = Immediate request-If all locks cannot be immediately granted, signal exception. 1 = Synchronous request-Wait until all locks can be granted.	
— Access state modifications	Bits 2-3
- When the process is entering lock wait for synchronous request:	Bit 2
0= Access state should not be modified. 1= Access state should be modified.	
- When the process is leaving lock wait:	Bit 3
0= Access state should not be modified. 1= Access state should be modified.	
— Reserved (binary 0)	Bits 4-5
— Time-out option	Bit 6
0= Wait for specified time, then signal time-out exception. 1= Wait indefinitely.	
— Reserved (binary 0)	Bits 7-15
— Modify process event mask option	Bit 16
0 = Do not modify process event mask 1 = Modify process event mask	
— Reserved (binary 0)	Bits 17-23
• Modify process event mask control	Char(4)
— New process event mask	UBin(2)
— Previous process event mask	UBin(2)
• Reserved (binary 0)	Char(11)
• Space location(s) to be locked	Space pointer data object
This should be repeated as specified by *number of lock requests in template* above.	

The *lock state selection* is located by adding the *offset to lock state selection values* above to operand 1.

• Lock state selection (repeated for each pointer in the template)	Char(1)
— Requested lock state	Bits 0-4

(1 = lock requested, 0 = lock not requested)

Only one state may be requested.

- LSRD lock	Bit 0
- LSRO lock	Bit 1
- LSUP lock	Bit 2
- LEAR lock	Bit 3
- LENR lock	Bit 4
— Reserved (binary 0)	Bit 5-6
— Entry active indicator	Bit 7

0 = Entry not active- This entry is not used.
1 = Entry active- Obtain this lock.

Lock Allocation Procedure: A single Lock Space Location instruction can request the allocation of one or more lock states on one or more space locations. Space location locks are granted sequentially until all the locks requested are granted.

The **wait time-out** field establishes the maximum amount of time that a process competes for the requested set of locks when the *lock request type* is *synchronous*. The bits in this field are numbered from 0 to 63, and bit 41 is defined as 1024 microseconds. The maximum wait time-out interval allowed is a value equal to (2^{48} - 1) microseconds. Any value that indicates more time than the maximum wait time-out causes the maximum wait time-out to be used. If the *wait time-out* field is specified with a value of binary 0, then the value associated with the default wait time-out parameter in the process definition template establishes the time interval.

When a requested lock state cannot be immediately granted, any locks already granted by this Lock Space Location instruction are released, and the *lock request type* specified in the lock request template establishes the machine action. The **lock request type** values are described in the following paragraphs.

- *Immediate Request*- If the requested space location locks cannot be granted immediately, this option causes the *lock request not grantable* (hex 1A02) exception to be signaled. No space location locks are granted, and the lock request is canceled.
- *Synchronous Request*- This option causes the process requesting the locks to be placed in the wait state until all requested locks can be granted. If the locks cannot be granted in the time interval established by the *wait time-out* field specified in the lock request template, the *space location lock wait time-out* (hex 3A04) exception is signaled to the requesting process at the end of the interval. No locks are granted, and the lock request is canceled.

If the *lock request type* is *synchronous* and the requested locks cannot be immediately granted, the **access state modification** field in the lock request template specifies whether the access state of the process access group is to be modified on entering and/or returning from the lock wait. The parameter has no effect if the process instruction wait access state control attribute specifies that no access state modification is allowed. If the process attribute value specifies that access state modification is allowed and the wait on event access state modification option specifies modify access state, the machine modifies the access state for the specified process access group.

If the *lock request type* is *synchronous* and the invocation containing the Lock Space Location instruction is terminated, then the lock request is canceled.

The **modify process event mask option** controls the state of the event mask in the process executing this instruction. When the process event mask is in the *masked* state, the machine does not schedule signaled event monitors in the process. The event monitors continue to be signaled by the machine or other processes. When the process event mask is modified to the *unmasked* state, event handlers are

scheduled to handle those events that occurred while the process was masked and those events occurring while in the unmasked state.

If the system security level machine attribute is hex 40 or greater and the process is running in user state, then the *modify process event mask option* is not allowed and a *template value invalid* (hex 3801) exception is signalled.

When the *modify process event mask* is set to 1, the **previous process event mask** will be returned and the **new process event mask** will take effect only when the space location lock(s) have been successfully granted. If the space location lock request is not successful, the *previous process event mask* value is not returned, nor does the *new process event mask* take effect.

The process event mask values are validity checked only when the modify process event mask is set to 1, and ignored otherwise. Valid masking values are:

0 Masked
256 Unmasked

Other values are reserved and must not be specified, otherwise a *template value invalid* (hex 3801) exception is signaled.

If any exception is identified during the instruction's execution, any locks already granted by the instruction are released, and the lock request is canceled.

For each space location lock, counts are kept by lock state and by process. When a lock request is granted, the appropriate *lock-count* of each lock state specified is incremented by 1.

Exceptions

Exception		Operands 1	Operands 2	Other
06	Addressing			
	01 space addressing violation	X	X	
	02 boundary alignment	X	X	
	03 range	X	X	
	06 optimized addressability invalid	X	X	
08	Argument/parameter			
	01 parameter reference violation	X	X	
10	Damage encountered			
	04 system object damage state			X
	44 partial system object damage			X
1A	Lock state			
	02 lock request not grantable	X		
1C	Machine-dependent exception			
	03 machine storage limit exceeded			X
	06 machine lock limit exceeded			X
20	Machine support			
	02 machine check			X
	03 function check			X

Exception		Operands 1	2	Other
22	Object access			
	02 object destroyed	X	X	
	08 object compressed			X
24	Pointer specification			
	01 pointer does not exist	X	X	
	02 pointer type invalid	X	X	
2E	Resource control limit			
	01 user profile storage limit exceeded			X
32	Scalar specification			
	01 scalar type invalid	X	X	
	03 scalar value invalid		X	
36	Space management			
	01 space extension/truncation			X
38	Template specification			
	01 template value invalid	X		
3A	Wait time-out			
	04 space location lock wait	X		

Materialize Data Space Record Locks (MATDRECL)

Op Code (Hex)	Operand 1	Operand 2
032E	Receiver	Record selection template

Operand 1: Space pointer.

Operand 2: Space pointer.

ILE access

```
MATDRECL (
     receiver           : space pointer;
     selection_template : space pointer
)
```

Description: This instruction materializes the current allocated locks on the specified data space record.

The current lock status of the data space record identified by the template in operand 2 is materialized into the space identified by operand 1.

The record selection template identified by operand 2 must be 16-byte aligned. The format of the record selection template is as follows.

- Record selection — Char(24)
  - Data space identification — System pointer
  - Record number — Bin(4)
  - Reserved — Char(4)
- Lock selection — Char(8)
  - Materialize data space locks held — Bit 0

    1 = Materialize
    0 = Do not materialize
  - Materialize data space locks waited for — Bit 1

    1 = Materialize
    0 = Do not materialize
  - Reserved — Bits 2-7
  - Reserved — Char(7)

The **data space identification** must be a system pointer to a data space.

The **record number** is a relative record number within that data space. If the *record number* is zero then all locks on the specified data space will be materialized. If the *record number* is not valid for the specified data space a *template value invalid* (hex 3801) exception is signaled.

Both of the fields specified under **lock selection** are bits which determine the locks to be materialized. If the **materialize data space locks held** is *materialize*, the current holders of the specified data space record lock are materialized. If the **materialize data space locks waited for** is *materialize*, any process waiting to lock the specified data space record is materialized.

The materialization template identified by operand 1 must be 16-byte aligned. The format of the materialization is as follows:

• Materialization size specification	Char(8)
– Number of bytes provided for materialization	Bin(4)
– Number of bytes available for materialization	Bin(4)
• Materialization data	Char(8)
– Count of locks held	UBin(2)
– Count of locks waited for	UBin(2)
– Reserved	Char(4)
• Locks held identification (repeated for each lock held)	Char(32)
– Process control space	System pointer
– Record number	Bin(4)
– Lock state being described Hex C0 = DLRD lock state Hex F8 = DLUP lock state All other values are reserved.	Char(1)
– Reserved	Char(11)
• Locks waited for identification (repeated for each lock waited for)	Char(32)
– Process control space	System pointer
– Record number	Bin(4)
– Lock state being described Hex C0 = DLRD lock state Hex F8 = DLUP lock state All other values are reserved.	Char(1)
– Reserved	Char(11)

The first 4 bytes of the materialization identify the total **number of bytes provided** for use by the instruction. This value is supplied as input to the instruction and is not modified by the instruction. A value of less than 8 causes the *materialization length* (hex 3803) exception to be signaled.

The second 4 bytes of the materialization identify the total **number of bytes available** to be materialized. The instruction materializes as many bytes as can be contained in the area specified as the receiver. If the byte area identified by the receiver is greater than that required to contain the information requested, the excess bytes are unchanged. No exceptions are signaled in the event that the receiver contains insufficient area for the materialization, other than the materialization length exception described previously.

The **count of locks held** contains the number of locks held. One system pointer to the **process control space** (PCS) of each process holding a lock, the relative **record number** which is locked, and the **lock state** are materialized in the area identified as **locks held identification**. These fields contain data only if **materialize data space locks held** is *materialize*.

The **count of locks waited** for contains the number of locks being waited for. One system pointer to the **process control space** (PCS) of each process waiting for a lock, the relative **record number**, and the

lock state which the process is waitng for are materialized in the area identified as locks waited for identification. These fields contain data only if **materialize data space locks waited for** is *materialize*.

Authorization Required

- Retrieve
  - Contexts referenced for address resolution

Lock Enforcement

- Materialize
  - Contexts referenced for address resolution

Exceptions

Exception		Operands 1	Operands 2	Other
06	Addressing			
	01 space addressing violation	X	X	
	02 boundary alignment	X	X	
	03 range	X	X	
	06 optimized addressability invalid	X	X	
08	Argument/parameter			
	01 parameter reference violation	X	X	
0A	Authorization			
	01 unauthorized for operation		X	X
10	Damage encountered			
	04 system object damage state		X	X
	05 authority verification terminated due to damaged object			X
	44 partial system object damage			X
1A	Lock state			
	01 invalid lock state		X	
1C	Machine-dependent exception			
	03 machine storage limit exceeded			X
20	Machine support			
	02 machine check			X
	03 function check			X
22	Object access			
	01 object not found	X	X	
	02 object destroyed	X	X	
	03 object suspended	X	X	
	07 authority verification terminated due to destroyed object			X
	08 object compressed			X
24	Pointer specification			

Exception		Operands 1	2	Other
	01 pointer does not exist	X	X	
	02 pointer type invalid	X	X	
2E	Resource control limit			
	01 user profile storage limit exceeded			X
32	Scalar specification			
	01 scalar type invalid	X	X	
36	Space management			
	01 space extension/truncation			X
38	Template specification			
	01 template value invalid		X	
	03 materialization length exception	X		

Materialize Process Locks (MATPRLK)

Op Code (Hex)	Operand 1	Operand 2
0312	Receiver	Process control space

Operand 1: Space pointer.

Operand 2: System pointer or null.

ILE access

```
MATPRLK (
     receiver                : space pointer;
 var process_control_spacec : system pointer OR
                              null operand
)
```

Description: The lock status of the process identified by operand 2 is materialized into the receiver specified by operand 1. If operand 2 is null, the lock status is materialized for the process issuing the instruction. The materialization identifies each object or space location for which the process has a lock allocated or for which the process is in a synchronous or asynchronous wait. The format of the materialization is as follows:

• Materialization size specification	Char(8)
– Number of bytes provided for materialization	Bin(4)
– Number of bytes available for materialization	Bin(4)
• Number of lock entries	Bin(2)
• Expanded number of lock entries	Bin(4)
• Reserved (binary 0)	Char(2)
• Lock status (repeated for each lock currently allocated or waited for by the process)	Char(32)
– Object, space location, or binary 0 if no pointer exists	System pointer or Space pointer
– Lock state	Char(1)
- LSRD	Bit 0
- LSRO	Bit 1
- LSUP	Bit 2
- LEAR	Bit 3
- LENR	Bit 4
- Reserved (binary 0)	Bits 5-7
– Status of lock state for process	Char(1)
- Reserved	Bits 0-1
- Object or space location no longer exists	Bit 2
- Waiting because this lock is not available	Bit 3
- Process in asynchronous wait for lock	Bit 4

- Process in synchronous wait for lock	Bit 5
- Implicit lock (machine-applied)	Bit 6
- Lock held by process	Bit 7
• Reserved (binary 0)	Char(14)

The first 4 bytes of the materialization identify the total **number of bytes provided** for use by the instruction. This value is supplied as input to the instruction and is not modified by the instruction. A value of less than 8 causes the *materialization length* (hex 3803) exception to be signaled.

The second 4 bytes of the materialization identify the total **number of bytes available** to be materialized. The instruction materializes as many bytes as can be contained in the area specified as the receiver. If the byte area identified by the receiver is greater than that required to contain the information requested, then the excess bytes are unchanged. No exceptions (other than the *materialization length* (hex 3803) exception described previously) are signaled if the receiver contains insufficient area for the materialization.

The **number of lock entries** field identifies the number of lock entries that are materialized. When a process holds more than 32,767 locks, this field is set with its maximum value of 32,767. This field has been retained in the template for compatibility with programs using the template prior to the changes made to support materialization of more than 32,767 lock entries.

The **expanded number of lock entries** field identifies the number of lock entries that are materialized. This field is always set in addition to the number of lock entries field described previously; however, it does not have a maximum limit of 32,767, so it can be used to specify that more than 32,767 locks have been materialized. When a process holds more than 32,767 locks, the *number of lock entries* field will equal 32,767, which would be incorrect. The *expanded number of lock entries* field, however, will identify the correct number of lock entries materialized. In all cases, this field should be used instead of the *number of lock entries* field to get the correct count of lock entries materialized.

Authorization Required

- Retrieve
  - Context referenced by address resolution

Lock Enforcement

- Materialize
  - Contexts referenced for address resolution

Exceptions

Exception		Operands 1	Operands 2	Other
06	Addressing			
	01 space addressing violation	X	X	
	02 boundary alignment	X	X	
	03 range	X	X	
	06 optimized addressability invalid	X	X	
08	Argument/parameter			
	01 parameter reference violation	X	X	
10	Damage encountered			

Exception		Operands 1	Operands 2	Other
	04 system object damage state	X	X	X
	05 authority verification terminated due to damaged object			X
	44 partial system object damage	X	X	X
1C	Machine-dependent exception			
	03 machine storage limit exceeded			X
20	Machine support			
	02 machine check			X
	03 function check			X
22	Object access			
	01 object not found	X	X	
	02 object destroyed	X	X	
	03 object suspended	X	X	
	07 authority verification terminated due to destroyed object			X
	08 object compressed			X
24	Pointer specification			
	01 pointer does not exist	X	X	
	02 pointer type invalid	X	X	
	03 pointer addressing invalid object		X	
28	Process state			
	02 process control space not associated with a process		X	
2E	Resource control limit			
	01 user profile storage limit exceeded			X
36	Space management			
	01 space extension/truncation			X
38	Template specification			
	03 materialization length exception	X		

Materialize Process Record Locks (MATPRECL)

Op Code (Hex)	Operand 1	Operand 2
031E	Receiver	Process selection template

Operand 1: Space pointer.

Operand 2: Space pointer.

ILE access

```
MATPRECL (
     receiver           : space pointer;
     selection_template : space pointer
)
```

Description: This instruction materializes the current allocated data space record locks held by the process. The current lock status of the process identified in the process selection template specified by operand 2 is materialized into the receiver identified by operand 1. The materialization identifies each data space record lock which the process has or the process is waiting to obtain.

If the process control space (PCS) pointer is null or all zeros, the lock activity for the process issuing the instruction is materialized.

The process selection template identified by operand 2 must be 16-byte aligned. The format of the process selection template is as follows:

- Process selection — Char(16)
  - Process identification — System pointer
- Lock selection — Char(8)
  - Materialize held locks — Bit 0
    - 1 = Materialize
    - 0 = Do not materialize
  - Materialize locks waited for — Bit 1
    - 1 = Materialize
    - 0 = Do not materialize
  - Reserved — Bits 2-7
  - Reserved — Char(7)

The **process identification** must be a system pointer to a process control space (PCS) or null, all zeros.

Both of the fields specified under **lock selection** are bits which determine the locks to be materialized. If the **materialize held locks** is *materialize*, any data base record lock held by the process is materialized. If the **materialize lock waited for** is *materialize* any data base record lock the process is waiting for is materialized.

The materialization template identified by operand 1 must be 16-byte aligned. The format of the materialization is as follows:

- Materialization size specification — Char(8)
  - Number of bytes provided for materialization — Bin(4)

– Number of bytes available for materialization	Bin(4)
• Materialization data	Char(8)
– Count of locks held	Bin(2)
– Count of locks waited for	Bin(2)
– Reserved	Char(4)
• Locks held identification (repeated for each lock held)	Char(32)
– Data space identification	System pointer
– Relative record number	Bin(4)
– Lock state being described Hex C0 = DLRD lock state Hex F8 = DLUP lock state All other values are reserved.	Char(1)
– Reserved	Char(11)
• Locks waited for identification (repeated for each lock waited for)	Char(32)
– Data space identification	System pointer
– Relative record number	Bin(4)
– Lock state being described Hex C0 = DLRD lock state Hex F8 = DLUP lock state All other values are reserved.	Char(1)
– Reserved	Char(11)

The first 4 bytes of the materialization identify the total **number of bytes provided** for use by the instruction. This value is supplied as input to the instruction and is not modified by the instruction. A value of less than 8 causes the *materialization length* (hex 3803) exception to be signaled.

The second 4 bytes of the materialization identify the total **number of bytes available** to be materialized. The instruction materializes as many bytes as can be contained in the area specified as the receiver. If the byte area identified by the receiver is greater than that required to contain the information requested, the excess bytes are unchanged. No exceptions are signaled in the event that the receiver contains insufficient area for the materialization, other than the *materialization length* (hex 3803) exception described previously.

The **count of locks held** contains the number of locks held by the process. One system pointer to the **data space**, **relative record number** in the data space, and **lock state** is materialized in the area identified as **locks held identification** for each lock. These fields contain data only if *materialize held locks* is *materialize*.

The **count of locks waited for** contains the number of locks that the process is waiting for. One system pointer to the **data space**, **relative record number** in the data space, and **lock state** is materialized in the area identified as locks waited for identification for each lock waited for. These fields contain data only if *materialize locks waited for* is *materialize*.

Authorization Required

- Retrieve
  - Contexts referenced for address resolution

Lock Enforcement

- Materialize
  - Contexts referenced for address resolution

Exceptions

Exception		Operands 1	2	Other
06	Addressing			
	01 space addressing violation	X	X	
	02 boundary alignment	X	X	
	03 range	X	X	
	06 optimized addressability invalid	X	X	
08	Argument/parameter			
	01 parameter reference violation	X	X	
0A	Authorization			
	01 unauthorized for operation		X	X
10	Damage encountered			
	04 system object damage state		X	X
	05 authority verification terminated due to damaged object			X
	44 partial system object damage			X
1A	Lock state			
	01 invalid lock state		X	
1C	Machine-dependent exception			
	03 machine storage limit exceeded			X
20	Machine support			
	02 machine check			X
	03 function check			X
22	Object access			
	01 object not found	X	X	
	02 object destroyed	X	X	
	03 object suspended	X	X	
	07 authority verification terminated due to destroyed object			X
	08 object compressed			X
24	Pointer specification			
	01 pointer does not exist	X	X	
	02 pointer type invalid	X	X	

Exception		Operands 1	2	Other
2E	Resource control limit			
	01 user profile storage limit exceeded			X
32	Scalar specification			
	01 scalar type invalid	X	X	
36	Space management			
	01 space extension/truncation			X
38	Template specification			
	01 template value invalid		X	
	03 materialization length exception	X		

Materialize Selected Locks (MATSELLK)

Op Code (Hex)	Operand 1	Operand 2
033E	Receiver	Object or space location

Operand 1: Space pointer.

Operand 2: System pointer or space pointer data object.

ILE access

```
MATSELLK (
    receiver                 : space pointer;
 var system_or_space_pointer : pointer
)
```

Description: The locks held by the process issuing this instruction for the object or space location referenced by operand 2 are materialized into the template specified by operand 1. The format of the materialization template is as follows:

- Materialization size specification — Char(8)
  - Number of bytes provided for materialization — Bin(4)
  - Number of bytes available for materialization — Bin(4)
- Cumulative lock status for all locks on operand 2 — Char(1)
  - Lock state — Char(1)
    - LSRD — Bit 0
    - LSRO — Bit 1
    - LSUP — Bit 2
    - LEAR — Bit 3
    - LENR — Bit 4
  - Reserved (binary 0) — Bits 5-7
- Reserved — Char(1)
- Number of lock entries — Bin(2)
- Reserved — Char(2)
- Lock status (repeated for each lock currently allocated) — Char(2)
  - Lock state — Char(1)

    Hex 80 = LSRD lock request
    Hex 40 = LSRO lock request
    Hex 20 = LSUP lock request
    Hex 10 = LEAR lock request
    Hex 08 = LENR lock request

    All other values are reserved
  - Status of lock — Char(1)
    - Reserved (binary 0) — Bits 0-5

- Implicit lock — Bit 6

  0 = Not implicit lock
  1 = Is implicit lock

- Reserved (binary 1) — Bit 7

The first 4 bytes of the materialization identifies the total **number of bytes provided** for use by the instruction. This value is supplied as input to the instruction and is not modified by the instruction. A value of less than 8 causes the *materialization length* (hex 3803) exception to be signaled.

The second 4 bytes of the materialization identifies the total **number of bytes available** to be materialized. The instruction materializes as many bytes as can be contained in the area specified as the receiver. If the byte area identified by the receiver is greater than that required to contain the information requested, then the excess bytes are unchanged. No exceptions are signaled in the event that the receiver contains insufficient area for the materialization, other than the *materialization length* (hex 3803) exception described previously.

A space pointer machine object cannot be specified for operand 2.

Authorization

- Retrieve
  - Context referenced by address resolution

Lock Enforcement:

- Materialize
  - Contexts referenced for address resolution

Exceptions

Exception		Operands 1	Operands 2	Other
06	Addressing			
	01 space addressing violation	X	X	
	02 boundary alignment	X	X	
	03 range	X	X	
	06 optimized addressability invalid	X	X	
08	Argument/parameter			
	01 parameter reference violation	X	X	
0A	Authorization			
	02 unauthorized for operation		X	
10	Damage encountered			
	04 system object	X	X	X
	05 authority verification terminated due to damaged object			X
	44 partial system object damage			X
1A	Lock state			
	01 invalid lock state		X	
1C	Machine-dependent exception			

Exception		Operands 1	2	Other
	03 machine storage limit exceeded			X
20	Machine support			
	02 machine check			X
	03 function check			X
22	Object access			
	01 object not found	X	X	
	02 object destroyed	X	X	
	03 object suspended	X	X	
	07 authority verification terminated due to destroyed object			X
	08 object compressed			X
24	Pointer specification			
	01 pointer does not exist	X	X	
	02 pointer type invalid	X	X	
	03 pointer addressing invalid object		X	
28	Process state			
	02 process control space not associated with a process		X	
2E	Resource control limit			
	01 user profile storage limit exceeded			X
32	Scalar specification			
	01 scalar type invalid	X	X	
36	Space management			
	01 space extension/truncation			X
38	Template specification			
	03 materialization length exception	X		

Transfer Object Lock (XFRLOCK)

Op Code (Hex)	Operand 1	Operand 2
0382	Receiving process control space	Lock transfer template

Operand 1: System pointer.

Operand 2: Space pointer.

ILE access

```
XFRLOCK (
  var receiving_process_control_space : system pointer
      lock_transfer_template          : space pointer
)
```

Description: The receiving process (operand 1) is allocated the locks designated in the lock transfer template (operand 2). Upon completion of the transfer lock request, the current process no longer holds the transferred lock(s).

Operand 2 identifies the objects and the associated lock states that are to be transferred to the receiving process. The space contains a system pointer to each object that is to have a lock transferred and a byte which defines whether this entry is active. If the entry is active, the space also contains the lock states to be transferred. Operand 2 must be aligned on a 16-byte boundary. The format is as follows:

- Number of lock transfer requests in template — Bin(4)
- Offset to lock state selection bytes — Bin(2)
- Reserved (binary 0) — Char(10)*
- Object lock(s) to be transferred — System pointer

  This should be repeated as specified by *number of lock transfer requests in template* above.

The *lock state selection* is located by adding the *offset to lock state selection values* above to operand 1.

- Lock state selection (repeated for each pointer in the template) — Char(1)
  - Lock state to transfer. Only one state may be requested. (1 = transfer) — Bits 0-4
    - LSRD — Bit 0
    - LSRO — Bit 1
    - LSUP — Bit 2
    - LEAR — Bit 3
    - LENR — Bit 4
  - Reserved (binary 0) — Bit 5*
  - Lock count — Bit 6

    0 = The current lock count is transferred.
    1 = A lock count of 1 is transferred.

— Entry active indicator Bit 7

0 = Entry not active. This *entry* is not used.
1 Entry active. This lock is transferred.

Note: Entries indicated by an asterisk are ignored by the instruction.

If the receiving process is issuing the instruction, then no operation is performed, and no exception is signaled. The **lock count** transferred is either the lock count held by the transferring process or a count of 1. If the receiving process already holds an identical lock, then the final lock count is the sum of the count originally held by the receiving process and the transferred count.

Only locks currently allocated to the process issuing the instruction can be transferred. If the transfer of an allocated lock would result in the violation of the lock allocation rules, then the lock cannot be transferred. An implicit lock may not be transferred.

No locks are transferred if an entry in the template is invalid.

The locks specified by operand 2 are transferred sequentially and individually. If one lock cannot be transferred because the process does not hold the indicated lock on the object, then exception data is saved to identify the lock that could not be transferred. Processing of the next lock to be transferred continues.

After all locks specified in operand 2 have been processed, the *object lock transferred* (hex 000A,03,01) event is signaled to the process receiving the locks if any locks were transferred. If any lock was not transferred, the *invalid object lock transfer request* (hex 1A04) exception is signaled.

When an object lock is transferred, the transferring process synchronously loses the record of the lock, and the object is locked to the receiving process. However, the receiving process obtains the lock asynchronously after the instruction currently being executed is completed. If the transferring process holds multiple locks for the object, any lock states not transferred are retained in the process.

Authorization Required

- Retrieve
  - Contexts referenced for address resolution

Lock Enforcement

- Materialize
  - Contexts referenced for address resolution

Exceptions

Exception		Operands 1	2	Other
06	Addressing			
	01 space addressing violation	X	X	
	02 boundary alignment	X	X	
	03 range	X	X	
	06 optimized addressability invalid	X	X	
08	Argument/parameter			
	01 parameter reference violation	X	X	

Exception		Operands 1	Operands 2	Other
0A	Authorization			
	01 unauthorized for operation		X	
10	Damage encountered			
	04 system object damage state	X	X	X
	05 authority verification terminated due to damaged object			X
	44 partial system object damage	X	X	X
1A	Lock state			
	01 invalid lock state			X
	04 invalid object lock transfer request		X	
1C	Machine-dependent exception			
	03 machine storage limit exceeded			X
20	Machine support			
	02 machine check			X
	03 function check			X
22	Object access			
	01 object not found	X	X	
	02 object destroyed	X	X	
	03 object suspended	X	X	
	07 authority verification terminated due to destroyed object			X
	08 object compressed			X
24	Pointer specification			
	01 pointer does not exist	X	X	
	02 pointer type invalid	X	X	
	03 pointer addressing invalid object		X	
28	Process state			
	02 process control space not associated with a process	X		
2E	Resource control limit			
	01 user profile storage limit exceeded			X
36	Space management			
	01 space extension/truncation			X
38	Template specification			
	01 template value invalid		X	

Unlock Object (UNLOCK)

Op Code (Hex)	Operand 1
03F1	Unlock template

Operand 1: Space pointer.

ILE access

```
UNLOCK (
    unlock_template : space pointer
)
```

Description: The instruction releases the object locks that are specified in the unlock template. The template specified by operand 1 identifies the system objects and the lock states (on those objects) that are to be released. The unlock template must be aligned on a 16-byte boundary. The format is as follows:

- Number of unlock requests in template — Bin(4)
- Offset to lock state selection bytes — Bin(2)
- Reserved (binary 0) — Char(8)*
- Unlock option — Char(1)
  - Reserved (binary 0) — Bits 0-3*
  - Unlock type — Bits 4-5

    00 = Unlock specific locks now allocated to process
    01 = Cancel specific asynchronously waiting lock request or allocated locks
    10 = Cancel all asynchronously waiting lock requests
    11 = Invalid
  - Reserved (binary 0) — Bit 6-7
- Reserved (binary 0) — Char(1)
- Object to unlock (one for each unlock request) — System pointer

The *Unlock options* is located by adding the **offset to lock state selection bytes** above to operand 1.

- Unlock options (repeated for unlock request) — Char(1)
  - Lock state to unlock (only one state can be selected) (1 = unlock) — Bits 0-4
    - LSRD — Bit 0
    - LSRO — Bit 1
    - LSUP — Bit 2
    - LEAR — Bit 3
    - LENR — Bit 4
  - Lock count option — Bit 5

    0 = Lock count reduced by 1
    1 = All locks are unlocked. The lock count is set to 0
  - Reserved (binary 0) — Bit 6*

— Entry active indicators Bit 7

0 = Entry not active. This entry is not used.
1 = Entry active. These locks are unlocked.

Note: Entries indicated by an asterisk are ignored by the instruction.

The **unlock type** field specifies if locks are to be released or outstanding lock requests are to be canceled.

If all *asynchronous lock waits* are being canceled (*unlock type* specified as 10), then *objects to unlock* and *unlock options* for each object are not required. If the asynchronous lock fields are provided in the template, then the data is ignored.

Specifying 01 for *unlock type* attempts to cancel an asynchronous lock request that is identical to the one defined in the template. After the instruction attempts to cancel the specified request, program execution continues just as if 00 had been specified for *unlock type*. A waiting lock request is canceled if the number of active requests in the template, the objects, the objects corresponding lock states, and the order of the active entries in the template all match.

When a lock is released, the lock count is reduced by 1 or set to 0 in the specified state. This option is specified by the **lock count option** parameter.

If 01 is specified for *unlock type* is specified and the unlock count option for an object lock is 0 (lock count reduced by 1), then a successful cancel satisfies this request, and no additional locks on the object are unlocked. If the *lock count option* for an object lock is set to 1 (*set lock count to 0*), the results of the cancel are disregarded, and all held locks on the object are unlocked.

Specific locks can be unlocked only if they are allocated to the process issuing the unlock instruction. Implicit locks may not be unlocked with this instruction. No locks are unlocked if an entry in the template is invalid.

Object locks to unlock are processed sequentially and individually. If one specific object lock cannot be unlocked because the process does not hold the indicated lock on the object, then exception data is saved, but processing of the instruction continues.

After all requested object locks have been processed, the *invalid unlock request* (hex 1A03) exception is signaled if any object lock was not unlocked.

If 01 is specified for *unlock type* is selected and the cancel attempt is unsuccessful, an *invalid unlock request* (hex 1A03) exception is signaled when any object lock in the template is not unlocked.

Authorization Required

- Retrieve
  - Contexts referenced for address resolution

Lock Enforcement

- Materialize
  - Contexts referenced for address resolution

Exceptions

Exception	Operands 1	Other

Exception		Operands 1	Other
06	Addressing		
	01 space addressing violation	X	
	02 boundary alignment	X	
	03 range	X	
	06 optimized addressability invalid	X	
08	Argument/parameter		
	01 parameter reference violation	X	
0A	Authorization		
	01 unauthorized for operation	X	
10	Damage encountered		
	04 system object damage state	X	X
	05 authority verification terminated due to damaged object		X
	44 partial system object damage	X	X
1A	Lock state		
	01 invalid lock state		X
	03 invalid unlock request	X	
1C	Machine-dependent exception		
	03 machine storage limit exceeded		X
20	Machine support		
	02 machine check		X
	03 function check		X
22	Object access		
	01 object not found	X	
	02 object destroyed	X	
	03 object suspended	X	
	07 authority verification terminated due to destroyed object		X
	08 object compressed		X
24	Pointer specification		
	01 pointer does not exist	X	
	02 pointer type invalid	X	
2E	Resource control limit		
	01 user profile storage limit exceeded		X
36	Space management		
	01 space extension/truncation		X
38	Template specification		
	01 template value invalid	X	

Unlock Space Location (UNLOCKSL)

Op Code (Hex)	**Operand 1**	**Operand 2**
03F2	Space location or Unlock template	Lock request

Operand 1: Space pointer data object.

Operand 2: Char(1) scalar or null.

ILE access

```
UNLOCKSL (
 var space_location : space pointer;
 var lock_request   : aggregate OR
                      null operand
)
```

Description: When operand 2 is not null, the lock type specified by operand 2 is removed from the space location (operand 1). When the operand 2 is null, the lock type is removed for the space locations specified in the unlock template (operand 1).

Any space location(s) specified by operand 1, or within the template specified by operand 1, need not exist when this instruction is issued although the space pointer must be a valid pointer as used to lock the space location.

A space pointer machine object cannot be specified for operand 1.

The following is the format of operand 2 when not null:

- Lock request — Char(1)
  - Lock state selection (1 = lock requested, 0 = lock not requested) — Bits 0-4
    - Only one state may be requested.
    - LSRD lock — Bit 0
    - LSRO lock — Bit 1
    - LSUP lock — Bit 2
    - LEAR lock — Bit 3
    - LENR lock — Bit 4
  - Reserved (binary 0) — Bits 5-7

If a space location lock cannot be unlocked because the process does not hold the indicated lock, then the *invalid space location unlock* (hex 1A05) exception is signaled.

When operand 2 is null, the lock request template identified by operand 1 must be aligned on a 16-byte boundary. The format of operand 1 is as follows:

- Number of space location unlock requests in template — Bin(4)
- Offset to lock state selection values — Bin(2)
- Reserved (binary 0) — Char(26)*

- Space location(s) to be unlocked — Space pointer

  This should be repeated as specified by *number of space location unlock requests in template* above.

The *unlock options* is located by adding the *offset to lock state selection values* above to operand 1.

- Unlock options (repeated for each unlock request) — Char(1)
  - Lock state to unlock (1 = unlock requested, 0 = unlock not requested) — Bits 0-4
    - Only one state may be requested.
    - LSRD lock — Bit 0
    - LSRO lock — Bit 1
    - LSUP lock — Bit 2
    - LEAR lock — Bit 3
    - LENR lock — Bit 4
  - Lock count option — Bit 5

    0 = Lock count reduced by 1
    1 = All locks are unlocked. (The lock count is set to 0).
  - Reserved (binary 0) — Bit 6
  - Entry active indicator — Bit 7

    0 = Entry not active. This entry is not used.
    1 = Entry active. Lock is to be unlocked.

Note: Entries indicated with an asterisk are ignored by the instruction.

This instruction can request the deallocation of one or more lock states on one or more space locations. The locks are deallocated sequentially until all specified locks are deallocated. When a lock is deallocated, the lock count is either reduced by 1 or set to 0 for the specified state. This option is specified by the **lock count option**.

Specific locks can be unlocked only if they are held by the process issuing the unlock instruction. If a space location lock cannot be unlocked because the process does not hold the indicated lock, then exception data is saved but processing of the instruction continues. After all requested space location locks have been processed, the *invalid unlock request* (hex 1A03) exception is signaled if any space location lock was not unlocked.

No locks are unlocked if a template value is invalid.

Exceptions

Exception		Operands 1	2	Other
06	Addressing			
	01 space addressing violation	X		
	02 boundary alignment	X		
	03 range	X		
	06 optimized addressability invalid	X		
08	Argument/parameter			

Exception		Operands 1	Operands 2	Other
	01 parameter reference violation	X		
10	Damage encountered			
	04 system object	X		X
	44 partial system object damage			X
1A	Lock state			
	03 invalid unlock request	X		
	05 invalid space location unlock	X		
1C	Machine-dependent exception			
	03 machine storage limit exceeded			X
	06 machine lock limit exceeded			X
20	Machine support			
	02 machine check			X
	03 function check			X
22	Object access			
	02 object destroyed	X	X	
	08 object compressed			X
24	Pointer specification			
	01 pointer does not exist	X		
	02 pointer type invalid	X		
2E	Resource control limit			
	01 user profile storage limit exceeded			X
32	Scalar specification			
	01 scalar type invalid	X	X	
	03 scalar value invalid		X	
36	Space management			
	01 space extension/truncation			X
38	Template specification			
	01 template value invalid	X		

Chapter 14. Exception Management Instructions

This chapter describes all instructions used for exception management. These instructions are in alphabetic order. For an alphabetic summary of all the instructions, see Appendix A, "Instruction Summary."

Materialize Exception Description (MATEXCPD) 14-3
Modify Exception Description (MODEXCPD) 14-6
Retrieve Exception Data (RETEXCPD) 14-9
Return From Exception (RTNEXCP) 14-12
Sense Exception Description (SNSEXCPD) 14-15
Signal Exception (SIGEXCP) 14-19
Test Exception (TESTEXCP) 14-24

© Copyright IBM Corp. 1991, 1993

Materialize Exception Description (MATEXCPD)

Op Code (Hex)	Operand 1	Operand 2	Operand 3
03D7	Attribute receiver	Exception description	Materialization option

Operand 1: Space pointer.

Operand 2: Exception description.

Operand 3: Character(1) scalar.

Description: The instruction materializes the attributes (operand 3) of an exception description (operand 2) into the receiver specified by operand 1.

The template identified by operand 1 must be a 16-byte aligned area in the space if the *materialization option* is hex 00.

Operand 2 identifies the exception description to be materialized.

The value of operand 3 specifies the **materialization option**. If the *materialization option* is hex 00, the format of the exception description materialization is as follows:

- Template size — Char(8)
  - Number of bytes provided for materialization — Bin(4)
  - Number of bytes available for materialization — Bin(4)
- Control flags — Char(2)
  - Exception handling action — Bits 0-2

    000 = Do not handle. (Ignore occurrence of exception and continue processing.)
    001 = Do not handle. (Disable this exception description and continue to search this invocation for another exception description to handle the exception.)
    010 = Do not handle. (Continue to search for an exception description by resignaling the exception to the preceding invocation.)
    100 = Defer handling. (Save exception data for later exception handling.)
    101 = Pass control to the specified exception handler.
  - No data — Bit 3

    0 = Exception data is returned
    1 = Exception data is not returned
  - Reserved (binary 0) — Bit 4
  - User data indicator — Bit 5

    0 = User data not present
    1 = User data present
  - Reserved (binary 0) — Bits 6-7
  - Exception handler type — Bits 8-9

    00 = External entry point
    01 = Internal entry point
    10 = Branch point
  - Reserved (binary 0) — Bits 10-15
- Instruction number to be given control — UBin(2)

(if *exception handler type* is *internal entry point* or *branch point*; otherwise, 0)

- Length of compare value (maximum of 32 bytes) — Bin(2)
- Compare value (size established by value of **length of compare value** field) — Char(32)
- Number of exception IDs — Bin(2)
- System pointer to the exception handling program (if *exception handler type* is *external entry point*) — System pointer
- Pointer to user data (not present if value of *user data indicator* is 0) — Space pointer
- Exception ID (one for each exception ID dictated by the **number of exception IDs** field) — Char(2)

If the *materialization option* is hex 01, the format of the materialization is as follows:

- Template size — Char(8)
  - Number of bytes provided for materialization — Bin(4)
  - Number of bytes available for materialization — Bin(4)
- Control flags — Char(2)
  - Exception handling action — Bits 0-2

    000 = Do not handle. (Ignore occurrence of exception and continue processing.)
    001 = Do not handle. (Disable this exception description and continue to search this invocation for another exception description to handle the exception.)
    010 = Do not handle. (Continue to search for an exception description by resignaling the exception to the preceding invocation.)
    100 = Defer handling. (Save exception data for later exception handling.)
    101 = Pass control to the specified exception handler.

  - No data — Bit 3

    0 = Exception data is returned
    1 = Exception data is not returned

  - Reserved (binary 0) — Bit 4-15

If the *materialization option* is hex 02, the format of the materialization is as follows:

- Template size — Char(8)
  - Number of bytes provided for materialization — Bin(4)
  - Number of bytes available for materialization — Bin(4)
- Compare value length (maximum of 32 bytes) — Bin(2)
- Compare value — Char(32)

The first 4 bytes of the materialization identify the total **number of bytes provided** for use by the instruction. This value is supplied as input to the instruction and is not modified by the instruction. A value of less than 8 causes the *materialization length* (hex 3803) exception to be signaled.

The second 4 bytes of the materialization identify the total **number of bytes available** to be materialized. The instruction materializes as many bytes as can be contained in the area specified as the receiver. If the byte area identified by the receiver is greater than that required to contain the information requested, then the excess bytes are unchanged. No exceptions (other than the *materialization length* (hex 3803) exception) are signaled in the event that the receiver operand contains insufficient area for the materialization.

Exceptions:

Exception		Operands 1	2	3	Other
06	Addressing				
	01 space addressing violation	X		X	
	02 boundary alignment	X		X	
	03 range	X		X	
	06 optimized addressability invalid	X		X	
08	Argument/parameter				
	01 parameter reference violation	X		X	
10	Damage encountered				
	04 system object damage state	X	X	X	X
	44 partial system object damage	X	X	X	X
1C	Machine-dependent exception				
	03 machine storage limit exceeded				X
20	Machine support				
	02 machine check				X
	03 function check				X
22	Object access				
	01 object not found	X		X	
	02 object destroyed	X		X	
	03 object suspended	X		X	
	08 object compressed				X
24	Pointer specification				
	01 pointer does not exist	X		X	
	02 pointer type invalid	X		X	
2E	Resource control limit				
	01 user profile storage limit exceeded				X
32	Scalar specification				
	03 scalar value invalid			X	
36	Space management				
	01 space extension/truncation				X
38	Template specification				
	03 materialization length exception	X			

Modify Exception Description (MODEXCPD)

Op Code (Hex)	Operand 1	Operand 2	Operand 3
03EF	Exception description	Modifying attributes	Modification option

Operand 1: Exception description.

Operand 2: Space pointer or character(2) constant.

Operand 3: Character(1) scalar.

Description: The exception description attributes specified by operand 3 are modified with the values of operand 2.

Operand 1 references the exception description.

Operand 2 specifies the new attribute values. Operand 2 may be either a character constant or a space pointer to the modification template. When operand 3 is a constant, operand 2 is a character constant; when operand 3 is not a constant, operand 2 is a space pointer.

The value of operand 3 specifies the **modification option**. If the *modification option* is hex 01 and operand 2 specifies a space pointer, the format of the modifying attributes pointed to by operand 2 is as follows:

- Template size — Char(8)
  - Number of bytes provided for materialization (must be at least 10) — Bin(4)
  - Number of bytes available for materialization — Bin(4)*
- Control flags — Char(2)
  - Exception handling action — Bits 0-2

    000 = Do not handle. (Ignore occurrence of exception and continue processing.)
    001 = Do not handle. (Disable this exception description and continue to search this invocation for another exception description to handle the exception.)
    010 = Do not handle. (Continue to search for an exception description by resignaling the exception to the preceding invocation.)
    100 = Defer handling. (Save exception data for later exception handling.)
    101 = Pass control to the specified exception handler.
  - No data — Bit 3

    0 = Exception data is returned
    1 = Exception data is not returned
  - Reserved (binary 0) — Bits 4-15

If the exception description was in the deferred state prior to the modification, the deferred signal, if present, is lost.

When the **no data** field is set to *exception data is not returned*, no data is returned for the Retrieve Exception Data or Test Exception instructions, and the *number of bytes available* for materialization field is set to 0. This option can also be selected in the object definition table entry of the exception description.

If the *modification option* of operand 3 is a constant value of hex 01, then operand 2 may specify a character constant. The operand 2 constant has the same format as the control flags entry previously described.

If the *modification option* is hex 02, then operand 2 must specify a space pointer. The format of the modification is as follows:

- Template size — Char(8)
  - Number of bytes provided — Bin(4)
    (must be at least 10 plus the length of the *compare value* in the exception description)
  - Number of bytes available for materialization — Bin(4)*
- Compare value length — Bin(2)*
  (maximum of 32 bytes)
- Compare value — Char(32)

Note: Entries shown here with an asterisk (*) are ignored by the instruction.

The number of bytes in the *compare value* is dictated by the **compare value length** specified in the exception description as originally specified in the object definition table.

An external exception handling program can be modified by resolving addressability to a new program into the system pointer designated for the exception description.

The presence of user data is not a modifiable attribute of exception descriptions. If the exception description has user data, it can be modified by changing the value of the data object specified in the exception description.

Exceptions

Exception		Operands 1	2	3	Other
06	Addressing				
	01 space addressing violation		X	X	
	03 range		X	X	
	06 optimized addressability invalid		X	X	
08	Argument/parameter				
	01 parameter reference violation		X	X	
10	Damage encountered				
	04 system object damage state	X	X	X	X
	44 partial system object damage	X	X	X	X
1C	Machine-dependent exception				
	03 machine storage limit exceeded				X
20	Machine support				
	02 machine check				X
	03 function check				X
22	Object access				
	01 object not found		X	X	
	02 object destroyed		X	X	

Exception		Operands 1	2	3	Other
	03 object suspended		X	X	
	08 object compressed				X
24	Pointer specification				
	01 pointer does not exist		X	X	
	02 pointer type invalid		X	X	
2E	Resource control limit				
	01 user profile storage limit exceeded				X
32	Scalar specification				
	03 scalar value invalid			X	
36	Space management				
	01 space extension/truncation				X
38	Template specification				
	01 template value invalid		X		
	02 template size invalid		X		

Retrieve Exception Data (RETEXCPD)

Op Code (Hex)	Operand 1	Operand 2
03E2	Receiver	Retrieve options

Operand 1: Space pointer.

Operand 2: Character(1) scalar.

Description: The data related to a particular occurrence of an exception is returned and placed in the specified space.

Operand 1 is a space pointer that identifies the receiver template. The template identified by operand 1 must be 16-byte aligned in the space.

The value of operand 2 specifies the type of exception handler for which the exception data is to be retrieved. The exception handler may be a branch point exception handler, an internal entry point exception handler, or an external entry point exception handler.

An *exception state of process invalid* (hex 1602) exception is signaled to the invocation issuing the Retrieve Exception Data instruction if the retrieve option is not consistent with the process's exception handling state. For example, the exception is signaled if the retrieve option specifies retrieve for internal entry point exception handler and the process exception state indicates that an internal exception handler has not been invoked.

After an invocation has been destroyed, exception data associated with a signaled exception description within that invocation is lost.

The format of operand 1 for the materialization is as follows:

- Template size — Char(8)
  - Number of bytes provided for retrieval — Bin(4)
  - Number of bytes available for retrieval — Bin(4)
- Exception identification — Char(2)
- Compare value length (maximum of 32 bytes) — Bin(2)
- Compare value — Char(32)
- Message reference key — Char(4)
- Exception specific data — Char(*)
- Source invocation — Invocation pointer or Null
- Target invocation — Invocation pointer
- Source invocation address — UBin(2)
- Target invocation address — UBin(2)
- Machine-dependent data — Char(10)

The first 4 bytes of the materialization identify the total **number of bytes provided** for retrieval of the exception data. This value is supplied as input to the instruction and is not modified by the instruction. A value of less than 8 causes the *materialization length exception* (hex 3803) exception to be signaled.

The second 4 bytes of the materialization identify the total **number of bytes available** to be materialized. The instruction materializes as many bytes as can be contained in the area specified as the

receiver. If the byte area identified by the receiver is greater than that required to contain the information requested. then the excess bytes are unchanged. No exceptions (other than the *materialization length* (hex 3803) exception) are signaled in the event that the receiver contains insufficient area for the materialization.

The **message reference key** field returns the architected value that uniquely identifies the message in the process queue space.

The **source invocation** and **source invocation address** identify the invocation that caused the exception to be signaled. For machine exceptions, this invocation pointer identifies the invocation executing when the exception occurred. For user-signaled exceptions, this invocation pointer locates the invocation that executed the Signal Exception instruction or the Send Process Message instruction. The pointer will be null if the source invocation no longer exists at the time that this instruction is executed. The *source instruction address* field locates the instruction that caused the exception to be signaled. This field in a bound program invocation will be set to 0.

The **target invocation** and **target invocation address** identify the invocation that is the target of the exception. This invocation is the last invocation that was given the chance to handle the exception. For machine exceptions, the first target invocation is the invocation incurring the exception. For user-signaled exceptions, the Signal Exception instruction may initially locate the current or any previous invocation. For Send Process Message, the source and target invocations are specified as input parameters. If the target invocation handles the exception by resignaling the exception, the immediately previous invocation is considered to be the target invocation. This may occur repetitively until no more prior invocations exist in the process and the signaled program invocation entry is assigned a value of binary 0. If an invocation handles the exception in any manner other than resignaling or does not handle the exception, that invocation is considered to be the target.

The *target instruction address* field specifies the number of the instruction that is currently being executed in the target invocation.

The machine extends the area beyond the *exception specific data* area with binary 0's so that the pointers to program invocations are aligned on a 16 byte boundary.

The operand 2 values are defined as follows:

- Retrieve options — Char(1)
  - Hex 00 = Retrieve for a branch point exception handler
  - Hex 01 = Retrieve for an internal entry point exception handler
  - Hex 02 = Retrieve for an external entry point exception handler

If the *exception data retention* option is set to 1 (do not save), the *number of bytes available for retrieval* is set to 0.

Exception data is always available to the process default exception handler.

Exceptions

Exception		Operands 1	Operands 2	Other
06	Addressing			
	01 space addressing violation	X	X	
	02 boundary alignment	X	X	
	03 range	X	X	
	06 optimized addressability invalid	X	X	

Exception		Operands 1	2	Other
08	Argument/parameter			
	01 parameter reference violation	X	X	
10	Damage encountered			
	04 system object damage state	X	X	X
	44 partial system object damage	X	X	X
16	Exception management			
	02 exception state of process invalid		X	
1C	Machine-dependent exception			
	03 machine storage limit exceeded			X
20	Machine support			
	02 machine check			X
	03 function check			X
22	Object access			
	01 object not found	X	X	
	02 object destroyed	X	X	
	03 object suspended	X	X	
	08 object compressed			X
24	Pointer specification			
	01 pointer does not exist	X	X	
2E	Resource control limit			
	01 user profile storage limit exceeded			X
32	Scalar specification			
	03 scalar value invalid		X	
36	Space management			
	01 space extension/truncation			X
38	Template specification			
	03 materialization length exception	X		

Return From Exception (RTNEXCP)

Op Code (Hex)	Operand 1
03E1	Return target

Operand 1: Space pointer.

Description: An internal exception handler subinvocation or an external exception handler invocation is terminated, and control is passed to the specified instruction in the specified invocation. All intervening invocations are marked as cancelled, down to, but not including, the invocation that is being returned to. When each of these invocations are returned to, their return handlers and invaocation exit (I-exit) routines/cancel handlers will be found and run.

Note: This instruction is not allowed from a bound program invocation.

The template identified by operand 1 must be 16-byte aligned in the space. It specifies the target invocation and target instruction in the invocation where control is to be passed. The format of operand 1 is as follows:

- Invocation address/offset — Space/Invocation pointer
- Reserved (binary 0) — Char(1)
- Action — Char(2)
  - Reserved (binary 0) — Bits 0-4
  - Use offset option — Bit 5

    0 = Use invocation address as a pointer value
    1 = Use invocation address as an offset value
  - Unstack option — Bit 6

    0 = The action performed is determined by the setting of the following action code (bit 7).
    1 = If the exception handler is an internal exception handler, resume execution with the instruction that follows the RTNEXCP instruction and terminate the internal exception handler subinvocation.
  - Action code — Bit 7

    0 = Re-execute the instruction that caused the exception or the instruction that invoked the invocation.
    1 = Resume execution with the instruction that follows the instruction that caused the exception or resume execution with the instruction that follows the instruction that invoked the invocation.
  - Reserved (binary 0) — Char(1)

The **invocation address/offset** field is a space/invocation pointer that identifies the invocation to which control will be passed.

The target *invocation address* field can also be an offset value from the current requesting invocation to the invocation to be searched. This is done by setting the **use offset option** field that follows the *invocation address* field to 1. If the *invocation offset* value locates the invocation stack base entry, the *invocation offset outside range of current stack* (hex 2C1A) exception is signaled. If the *invocation offset* value is a positive number (which represents newer invocations on the stack) a *template value invalid* (hex 3801) exception is signaled. The current instruction in an invocation is the one that caused another invocation to be created.

The **unstack option** is only valid when issued in an internal exception handler subinvocation and is ignored for an external exception handler invocation. This option will cause the internal exception

handler subinvocation to be terminated and control will resume at the instruction immediately following the RTNEXCP instruction. In effect, this option will cause the current subinvocation to be unstacked.

If the **action code** is 0, then the current instruction of the addressed invocation is reexecuted, if it is allowed. If the *action code* is 1, execution resumes with the instruction following the current instruction of the addressed invocation, if it is allowed. If it is not, a *retry/resume invalid* (hex 1604) exception will be signaled.

The Return From Exception instruction may be issued only from the initial invocation of an external exception handling sequence or from an invocation that has an active internal exception handler.

If the instruction is issued from an invocation that is not an external exception handler and has no internal exception handler subinvocations, the *return instruction invalid* (hex 2C01) exception is signaled.

The following table shows the actions performed by the Return From Exception instruction:

Invocation Issuing Instruction	Addressing Own Invocation/Option	Addressing Higher Invocation/Option
Not handling exception	Error (see note 1)	Error (sse note 1)
Handling internal exception(s)	Allowed (see note 2)	Allowed (see note 3)
Handling external exception(s)	Error (see note 1)	Allowed (see note 3)
Handling external exception(s) and internal exception(s)	Allowed (see note 2)	Allowed (see note 3)

Notes:

1. A *return instruction invalid* (hex 2C01) exception is signaled. If there are no more internal exception handler subinvocations active and this invocation is not an external exception handler, the instruction may not be issued.
2. The current internal exception handler subinvocation is terminated.
3. All invocations after the addressed invocation are terminated and execution proceeds within the addressed invocation. Any invocation exit programs set for the terminated invocations will be given control before execution proceeds within the addressed invocation.

Whenever an invocation is terminated, the invocation count in the corresponding activation entry (if any) is decremented by 1.

An *action code* of 1 specifies completion of an instruction rather than execution of the following instruction if the current instruction in the addressed invocation signaled a size exception or a floating-point inexact result exception.

Note: The previous condition does not apply if any of the above exceptions were explicitly signaled by a Signal Exception instruction.

A Return From Exception instruction cannot be used or recognized in conjunction with a branch point internal exception handler.

If a failure to invoke an invocation exit handler occurs, a failure to invoke program event is signaled.

Exceptions

Exception		Operands 1	Other
06	Addressing		

Exception		Operands 1	Other
	01 space addressing violation	X	
	02 boundary alignment	X	
	03 range	X	
	06 optimized addressability invalid	X	
08	Argument/parameter		
	01 parameter reference violation	X	
10	Damage encountered		
	04 system object damage state	X	X
	44 partial system object damage	X	X
16	Exception management		
	03 invalid invocation	X	
	04 retry/resume invalid	X	
1C	Machine-dependent exception		
	03 machine storage limit exceeded		X
20	Machine support		
	02 machine check		X
	03 function check		X
22	Object access		
	02 object destroyed	X	
	03 object suspended	X	
	08 object compressed		X
24	Pointer specification		
	01 pointer does not exist	X	
	02 pointer type invalid	X	
2C	Program execution		
	01 return instruction invalid		X
	12 activation group access protection violation	X	
	1A invocation offset outside range of current stack	X	
2E	Resource control limit		
	01 user profile storage limit exceeded		X
36	Space management		
	01 space extension/truncation		X
38	Template specification		
	01 template value invalid	X	

Sense Exception Description (SNSEXCPD)

Op Code (Hex)	Operand 1	Operand 2	Operand 3
03E3	Attribute receiver	Invocation template	Exception template

Operand 1: Space pointer.

Operand 2: Space pointer.

Operand 3: Space pointer.

Note: A change has been made in the way in which exceptions are handled for bound programs. This instruction is intended for use with Original Program Model (OPM) programs, but can be used against New Program Model (NPM) bound programs. The data that is returned when an NPM program is accessed will always say that there is an external handler for the sensed exception, that there is no exception data being returned and a starting exception description number of 0.

Description: The Sense Exception Description instruction searches the invocation specified by operand 2 for an exception description that matches the *exception identifier* and *compare value* specified by operand 3 and returns the *user data* and *exception handling action* specified in the exception description. The exception descriptions of the invocation are searched in ascending Object Definition Table (ODT) number sequence.

The template identified by operand 1 must be 16-byte aligned.

The format of the attribute receiver is as follows:

- Template size — Char(8)
  - Number of bytes provided for materialization — Bin(4)
  - Number of bytes available for materialization — Bin(4)
- Control flags — Char(2)
  - Exception handling action — Bits 0-2

    000 = Do not handle- Ignore occurrence of exception and continue processing
    010 = Do not handle- Continue search for an exception description by resignaling the exception to the immediately preceding invocation
    100 = Defer handling- Save exception data for later exception handling
    101 = Pass control to the specified exception handler
  - No data — Bit 3

    0 = Exception data is returned
    1 = Exception data is not returned
  - Reserved (binary 0) — Bit 4
  - User data indicator — Bit 5

    0 = User data not present
    1 = User data present
  - Reserved (binary 0) — Bits 6-7
  - Exception handler type — Bits 8-9

    00 = External entry point
    01 = Internal entry point
    10 = Branch point

— Reserved (binary 0)	Bits 10-15
• Relative exception description number	Bin(2)
• Reserved (binary 0)	Char(4)
• Pointer to user data (binary 0 if value of *user data indicator* is 0)	Space pointer

The first 4 bytes of the materialization identify the total **number of bytes provided** for use by the instruction. This value is supplied as input to the instruction and is not modified by the instruction. A value of less than 8 causes the *materialization length* (hex 3803) exception to be signaled.

The second 4 bytes of the materialization identify the total **number of bytes available** to be materialized. The instruction materializes as many bytes as can be contained in the area specified as the receiver. If the byte area identified by the receiver is greater than that required to contain the information requested, then the excess bytes are unchanged. No exception is signaled in the event the receiver contains insufficient area for the materialization, other than the *materialization length* (hex 3803) exception described previously.

The **relative exception description number** field identifies the relative number of the exception description that matched the search criteria. The order of definition of the exception descriptions in the ODT determines the value of the index. A value of 1 indicates that the first exception description defined in the ODT matched the search criteria.

The format of the invocation template is as follows:

• Invocation address/offset	Space/Invocation pointer
• Search flags	Char(2)
— Use offset option 0 = Use invocation address as a pointer value 1 = Use invocation address as an offset value	Bit 0
— Reserved (binary 0)	Bits 1-15
• First exception description to search	Bin(2)

The template identified by operand 2 must be 16-byte aligned. The **invocation address/offset** field is a space/invocation pointer that identifies the invocation to be searched. The invocation is searched for a matching exception description. If the *invocation address* locates either an invalid invocation or the invocation stack base entry, the *invalid invocation address* (hex 1603) exception is signaled.

The *invocation address/offset* field can also be an offset value from the current requesting invocation to the invocation to be searched. This is setting the **use offset option** bit field that follows the *invocation address* field to 1. If the *invocation offset* value locates the invocation stack base entry, the *invocation offset outside range of current stack* (hex 2C1A) exception is signaled. If the *invocation offset* value is positive or zero, a *template value invalid* (hex 3803) exception is signaled.

The **first exception description to search** field specifies the relative number of the exception description to be used to start the search. The number must be a nonzero positive binary number determined by the order of definition of exception descriptions in the ODT. A value of 1 indicates that the first exception description in the invocation is to be used to begin the search. If the value is greater than the number of exception descriptions for the invocation, the operand 1 template is materialized with the *number of bytes available* for materialization set to 0.

The operand 3 exception template specifies the exception-related data to be used as a search argument. The format of the template is as follows:

• Template size	Char(8)

– Number of bytes provided for materialization (must be at least 44)	Bin(4)
– Number of bytes available for materialization	Bin(4)*
• Exception identifier	Char(2)
• Compare value length (maximum of 32)	Bin(2)
• Compare value	Char(32)

Note: Entries noted with an asterisk (*) are ignored by the instruction.

The **exception identifier** in the exception description can be specified in one of the following ways:

Hex 0000 = Any exception ID will result in a match

Hex nn00 = Any exception ID in class nn will result in a match

Hex nnmm = Only exception ID nnmm will result in a match

If a match on *exception ID* is detected, the corresponding **compare values** are matched. If the *compare value length* in the exception description is less than the *compare value* in the search template, the length of the *compare value* in the exception description is used for the match. If the *compare value length* in the exception description is greater than the *compare value* in the search template, an automatic mismatch results.

If a match on *exception ID* and *compare value* is detected, the exception handling action of the exception description determines which of the following actions is taken:

IGNORE The operand 1 template is materialized.

DISABLE The exception description is bypassed and the search for an exception description continues with the next exception description defined for the invocation.

RESIGNAL The operand 1 template is materialized.

DEFER The operand 1 template is materialized.

HANDLE The operand 1 template is materialized.

If no exception description of the invocation matches the exception ID and compare value of operand 3, the number of bytes available for materialization on the operand 1 template is set to 0.

Exceptions

Exception		Operands 1	2	3	Other
06	Addressing				
	01 space addressing violation	X	X	X	
	02 boundary alignment	X	X	X	
	03 range	X	X	X	
	06 optimized addressability invalid	X	X	X	
08	Argument/parameter				
	01 parameter reference violation	X	X	X	
10	Damage encountered				
	04 system object damage				X
	44 partial system object damage				X
16	Exception management				

Exception		Operands 1	2	3	Other
	03 invalid invocation address		X		
1C	Machine-dependent exception				
	03 machine storage limit exceeded				X
20	Machine support				
	02 machine check				X
	03 function check				X
22	Object access				
	01 object not found	X	X	X	
	02 object destroyed	X	X	X	
	03 object suspended	X	X	X	
	08 object compressed				X
24	Pointer specification				
	01 pointer does not exist	X	X	X	
	02 pointer type invalid	X	X	X	
2C	Program execution				
	1A invocation offset outside range of current stack		X		
2E	Resource control limit				
	01 user profile storage limit exceeded				X
32	Scalar specification				
	01 scalar type invalid	X	X	X	
36	Space management				
	01 space extension/truncation				X
38	Template specification				
	01 template value invalid		X	X	
	02 template size invalid			X	
	03 materialization length exception	X			

Signal Exception (SIGEXCP)

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand 3-4
SIGEXCP 10CA		Attribute template	Exception data	
SIGEXCPB 1CCA	Branch options	Attribute template	Exception data	Branch targets
SIGEXCPI 18CA	Indicator options	Attribute template	Exception data	Indicator targets

Operand 1: Space pointer.

Operand 2: Space pointer.

Operand 3-4:

- *Branch Form*-Branch point, instruction pointer, relative instruction number, or absolute instruction number.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Description: The Signal Exception instruction signals a new exception or resignals an existing exception to the process. Optionally, the instruction branches to one of the specified targets based on the results of the signal and the selected branch options in the extender field, or it sets indicators based on the results of the signal. The signal is presented starting at the invocation identified in the signal template.

The SIGEXCP instruction is a subset of the Send Process Message (SNDPRMSG) instruction. Additional parameters used in the SNDPRMSG instruction will be defaulted when the SIGEXCP instruction is used. See the SNDPRMSG instruction for more information about the complete set of parameters available when signalling an exception.

The template identified by operand 1 specifies the signal option and starting point. It must be 16-byte aligned in the space with the following format.

- Target invocation address — Space/Invocation pointer
- Signal option — Char(1)
  - Signal/resignal option — Bit 0

    0 = Signal new exception.
    1 = Resignal currently handled exception (valid only for an external exception handler).

  - Invoke PDEH (process default exception handler) option — Bit 1

    0 = Invoke PDEH if no exception description found for invocation.
    1 = Do not invoke PDEH if no exception description found for invocation (ignore if base invocation entry specified).

  - Exception description search control — Bit 2

    0 = Exception description search control not present
    1 = Exception description present

  - Reserved (binary 0) — Bits 3-7
- Reserved (binary 0) — Char(1)

- First exception description to search Bin(2)

The **target invocation address** pointer uniquely identifies the invocation to which the exception is to be signalled. Signalling directly to the PDEH can not be accomlished via this instruction. Support for this function is being provided in the SNDPRMSG instruction. If the *target invocation address* pointer locates neither a valid invocation entry nor the base invocation entry, the *invalid invocation* (hex 1603) exception is signaled.

The invocation which issued this instruction will be checked to ensure it has the proper authority to send an exception message to the target invocation. If the authority check fails, *activation group access violation* (hex 2C12) will be signaled. If the program associated with the invocation has defined an exception description to handle the exception, the specified action is taken; otherwise, the PDEH is invoked unless the **invoke PDEH option** bit is 1 (the exception is considered ignored). If the base invocation entry is addressed instead of an existing invocation, the PDEH will be invoked.

A change has been made to the way in which exception handlers are determined for bound programs. The following description relates only to the invocation of exception handlers related to OPM programs. In both instances the actions of signalling and handling have been broken apart.

Note:

Exception descriptions of an invocation are searched in ascending ODT number sequence. If the **exception description search control** control specified that an *exception description is not present*, the search begins with the first exception description defined in the ODT. Otherwise, the **first exception description to search** value identifies the relative number of the exception description to be used to start the search. The value must be a nonzero positive binary number determined by the order of definition of exception descriptions in the ODT. This value is also returned by the Sense Exception Description instruction. A value of 1 indicates that the first exception description in the invocation is to be used to begin the search. If the value is greater than the number of exception descriptions for the invocation, the *template value invalid* (hex 3801) exception is signaled.

If an **exception ID** in an exception description corresponds to the signaled exception, the corresponding **compare values** are verified. If the *compare value length* in the exception description is less than the *compare value length* in the signal template, the length of the compare value in the exception description is used for the match. If the *compare value length* in the exception description is greater than the *compare value length* in the signal template, an automatic mismatch results. Machine-signaled exceptions have a 4-byte compare value of binary 0's.

An exception description may monitor for an exception with a generic ID as follows:

Hex 0000 = Any signaled exception ID results in a match.

Hex nn00 = Any signaled exception ID in class nn results in a match.

Hex nnmm = The signaled exception ID must be exactly nnmm in order for a match to occur.

An exception description may be in one of five states, each of which determines an action to be taken when the match criteria on the exception ID and compare value are met.

IGNORE No exception handling occurs. The Signal Exception instruction is assigned a resultant condition of ignored. If a corresponding branch or indicator setting is present, that action takes place.

DISABLE The exception description is bypassed, and the search for a monitor continues with the next exception description defined for the invocation.

RESIGNAL The search for a monitoring exception description is to be reinitiated at the preceding invocation. A resignal from the initial invocation in the process results in the invocation of the process default exception handler. A resignal from an invocation exit program results in an unhandled exception that causes process termination.

DEFER The exception description is signaled, and the Signal Exception instruction is assigned the resultant condition of deferred. If a corresponding branch or indicator setting is present, that action takes place. To take future action on a deferred exception, the exception description must be synchronously tested with the Test Exception instruction in the signaled invocation.

HANDLE Control is passed to the indicated exception handler, which may be a branch point, an internal subinvocation, or an external invocation.

If the exception description is in the ignore or defer state and if the Signal Exception instruction does not specify a branch or indicator condition or if it specifies branch or indicator conditions that are not met, then the instruction following the Signal Exception instruction is executed.

When control is given to an internal or branch point exception handler, all invocations up to, but not including, the exception handling invocation are terminated. Any invocation exit programs set for the terminated invocations will be given control before execution proceeds in the signaled exception handler.

When this instruction is invoked with the *Resignal* option, all invocations up to, but not including, the interrupted invocation are cancelled and the message is signalled to the next oldest invocation in the stack. This implies that the Return from Exception (RTNEXCP) instruction can no longer return to the invocation that issued the resignal request. Any cancel handlers set for the cancelled invocations will be given control before execution proceeds in the signaled exception handler.

If a failure to invoke an external exception handler or an invocation exit occurs, a failure to invoke program event is signaled. For each destroyed invocation, the invocation count in the corresponding activation entry (if any) is decremented by 1.

The template identified by operand 2 must be 16-byte aligned in the space. It specifies the exception-related data to be passed with the exception signal. The format of the exception data is the same as that returned by the Retrieve Exception Data instruction. The format is as follows:

- Template size — Char(8)
  - Number of bytes of data to be signaled (must be at least 48 bytes) — Bin(4)
  - Number of bytes available for materialization — Bin(4)*
- Exception identification — Char(2)
- Compare value length (maximum of 32 bytes) — Bin (2)
- Compare value — Char(32)
- Reserved — Char(4)*
- Exception specific data — Char(*)

Note: Entries shown here with an asterisk (*) are ignored by the instruction.

Operand 2 is ignored if *signal/resignal option* is *resignal* because the exception-related data is the same as for the exception currently being processed; however, it must be specified when signaling a new exception.

The maximum size for *exception specific* data that is to accompany an exception signaled by the Signal Exception instruction is 65 503 bytes, including the standard exception data.

The following required parameters only available on the SNDPRMSG instruction will be given the following default values:

Message status - Log message + Retain + Action pending

Initial monitor priority - 64

Interrupt class mask - Message generated by Signal Exception instruction

Source invocation - invocation issuing SIGEXCP instruction

Resultant Conditions

- Exception ignored
- Exception deferred.

Authorization Required: The invocation which originated the exception must have proper activation group access to the target invocation. The following algorithm is used to determine this access.

1. The invocation which invoked the SIGEXCP instruction must have access to the invocation identified as the Originating Invocation.
2. The Originating Invocation must have access to the invocation identified as the Source Invocation or to the invocation directly called by the Source invocation.
3. The Originating Invocation must have access to the invocation identified as the Target Invocation or to the invocation directly called by the Target Invocation.

If any of the access checks fail then an *activation group access violation* (hex 2C12) exception will be signaled.

Exceptions

Exception		Operands 1	Operands 2	Other
06	Addressing			
	01 space addressing violation	X	X	
	02 boundary alignment	X	X	
	03 range	X	X	
	06 optimized addressability invalid	X	X	
08	Argument/parameter			
	01 parameter reference violation	X	X	
10	Damage encountered			
	04 system object damage state	X	X	X
	44 partial system object damage	X	X	X
16	Exception management			
	02 exception state of process invalid			X
	03 invalid invocation	X		
1C	Machine-dependent exception			
	03 machine storage limit exceeded			X
20	Machine support			
	02 machine check			X

Exception		Operands 1	2	Other
	03 function check			X
22	Object access			
	01 object not found	X	X	
	02 object destroyed	X	X	
	03 object suspended	X	X	
	08 object compressed			X
24	Pointer specification			
	01 pointer does not exist	X	X	
	02 pointer type invalid	X	X	
2C	Program execution			
	12 activation group access violation	X		
2E	Resource control limit			
	01 user profile storage limit exceeded			X
36	Space management			
	01 space extension/truncation			X
38	Template specification			
	01 template value invalid	X		
	02 template size invalid	X		

Test Exception (TESTEXCP)

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand 3-4
TESTEXCP 104A		Receiver	Exception description	
TESTEXCPB 1C4A	Branch options	Receiver	Exception description	Branch options
TESTEXCPI 184A	Indicator options	Receiver	Exception description	Indicator options

Operand 1: Space pointer.

Operand 2: Exception description.

Operand 3-4:

- *Branch Form*-Branch point, instruction pointer, relative instruction number, or absolute instruction number.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

Description: The instruction tests the signaled status of the exception description specified in operand 2, and optionally alters the control flow or sets the specified indicators based on the test. Exception data is returned at the location identified by operand 1. The branch or indicator setting occurs based on the conditions specified in the extender field depending on whether or not the specified exception description is signaled.

Operand 2 is an exception description whose signaled status is to be tested. An exception can be signaled only if the referenced exception description is in the deferred state.

Operand 1 addresses a space into which the exception data is placed if an exception identified by the exception description has been signaled.

The template identified by operand 1 must be 16-byte aligned in the space and is formatted as follows:

Field	Type
• Template size	Char(8)
– Number of bytes provided for materialization	Bin(4)
– Number of bytes available for materialization (0 if exception description is not signaled)	Bin(4)
• Exception identification	Char(2)
• Compare value length (maximum of 32 bytes)	Bin(2)
• Compare value	Char(32)
• Message reference key	Char(4)
• Exception-specific data	Char(*)
• Source invocation address	Invocation/Null pointer
• Target invocation address	Invocation pointer
• Signaling program instruction address	UBin(2)
• Signaled program instruction address	UBin(2)
• Machine-dependent data	Char(10)

The first 4 bytes of the materialization identify the total **number of bytes provided** for use by the instruction. This value is supplied as input to the instruction and is not modified by the instruction. A value of less than 8 causes the *materialization length* (hex 3803) exception to be signaled.

The second 4 bytes of the materialization identify the total **number of bytes available** to be materialized. The instruction materializes as many bytes as can be contained in the area specified as the receiver. If the byte area identified by the receiver is greater than that required to contain the information requested, then the excess bytes are unchanged. No exceptions (other than the *materialization length* (hex 3803) exception) are signaled in the event that the receiver contains insufficient area for the materialization.

If the exception description is not in the signaled state, the *number of bytes available* for the materialization entry is set to binary 0's, and no other bytes are modified.

The **message reference key** field holds the architected value that uniquely identifies the exception message in a process queue space.

The **source invocation address** field will contain a null pointer if the source invocation no longer exists when this instruction is executed.

The area beyond the *exception-specific data* area is extended with binary 0's so that pointers to program invocations are properly aligned.

If no branch options are specified, instruction execution proceeds at the instruction following the Test Exception instruction.

If the *exception data retention option*, from the exception description, is set to 1 (do not save), no data is returned by this instruction.

Resultant Conditions

- Exception signaled
- Exception not signaled.

Exceptions

Exception		Operands 1	2	Other
06	Addressing			
	01 space addressing violation	X		
	02 boundary alignment	X		
	03 range	X		
	06 optimized addressability invalid	X		
08	Argument/parameter			
	01 parameter reference violation	X		
10	Damage encountered			
	04 system object damage state	X	X	X
	44 partial system object damage	X	X	X
16	Exception management			
	01 exception description status invalid		X	
1C	Machine-dependent exception			

Exception		Operands 1	2	Other
	03 machine storage limit exceeded			X
20	Machine support			
	02 machine check			X
	03 function check			X
22	Object access			
	08 object compressed			X
24	Pointer specification			
	01 pointer does not exist	X		
	02 pointer type invalid	X		
2E	Resource control limit			
	01 user profile storage limit exceeded			X
36	Space management			
	01 space extension/truncation			X
38	Template specification			
	03 materialization length exception	X		

Chapter 15. Queue Space Management Instructions

This chapter describes the instructions used for queue space management. These instructions are in alphabetic order. For an alphabetic summary of all the instructions, see Appendix A, "Instruction Summary."

Materialize Process Message (MATPRMSG) 15-3

© Copyright IBM Corp. 1991, 1993

Materialize Process Message (MATPRMSG)

Op Code (Hex)	Operand 1	Operand 2	Operand 3	Operand 4
039C	Materialization template	Message template	Source template	Selection template

Operand 1: Space pointer

Operand 2: Space pointer

Operand 3: Space pointer or null

Operand 4: Space pointer

ILE access

```
MATPRMSG (
        receiver_template  : space pointer;
        message_template   : space pointer;
        source_template    : space pointer; OR
                             null operand;
        selection_template : space pointer
)
```

Description: A message is materialized from a queue space according to the options specified. The message is located on a queue space queue specified by *operand 3*. The message is selected by the *operand 4* criteria. *Operands 1* and *2* contain the materialized information from the process message.

The template identified by *operand 1* must be 16-byte aligned. Following is the format of the materialization template:

Template size specification	Char(8)*
— Number of bytes provided for materialization	Bin(4)
— Number of bytes available for materialization	Bin(4)
• Process queue space queue offset	Bin(4)
• Reserved (binary 0)	Char(4)
• Time sent	Char(8)
• Time modified	Char(8)
• Interupted invocation	Invocation pointer
• Target invocation	Invocation pointer, or System pointer
• Original target invocation	Invocation pointer
• Source program location	Suspend pointer
• Target program location	Suspend pointer
• Originating program location	Suspend pointer
• Invocation mark	UBin(4)
• Activation group mark	UBin(4)
• Reserved (binary 0)	Char(8)

The first 4 bytes of the materialization template identify the total **number of bytes provided** for use by the instruction. This number is supplied as input to the instruction and is not modified by the instruction. If the value is zero for this field, then the operand 1 template is not returned. A number less than 128 (but not 0) causes the *materialization length* (hex 3803) exception.

Queue space queue offset
This value indicates which queue in the queue space a message resides on. A value of -1 will be returned if the message is on the external queue, and zero if the message is on the message log. If the message resides on an invocation queue this field will be zero.

Time sent
The value of the system time-of-day clock when the message was originally sent (Send Process Message instruction, Signal Exception, or message originated as a result of an exception).

Time modified
This value is initially equal to the *time sent* value. However, it can be modified by the Modify Process Message instruction.

Interrupted invocation
An invocation pointer that addresses the invocation which currently has this message as its interrupt cause. This pointer is null if the message is not an exception message.

Target invocation
An invocation pointer or system pointer that identifies which queue contains the message. If the message resides on an invocation queue, this field contains an invocation pointer that addresses the invocation whose invocation message queue currently contains the message. If the message does not reside on an invocation queue, then a system pointer to the Queue Space is returned. The *queue space queue offset* field indicates which queue on which the message resides, the message log or the external queue.

Original target invocation
An invocation pointer that addresses the invocation which originally was sent the message. This pointer is null if the original target invocation no longer exists.

Source program location
A suspend pointer which identifies the program, module, procedure, and statement where the source invocation was suspended (due to a CALL or some form of interrupt). If the message was originally sent as a non-interrupt message, then the source invocation is the one identified by the Send Process Message instruction, and the suspend point identified is the one that was current when that instruction was executed; otherwise the source invocation is the invocation for which the message was most recently an interrupt cause.

Target program location
A suspend pointer which identifies the program, module, procedure, and statement where the target invocation was suspended (due to a CALL or some form of interrupt). If the message is no longer in an invocation message queue, then this pointer reflects the invocation of the last (most recent) invocation message queue in which the message resided. This pointer is null if the message has never resided in an invocation message queue.

Originating program location
A suspend pointer which identifies the program, module, procedure, and statement of the Send Process Message instruction that sent the message. For messages sent by the machine, this pointer area contains a machine-dependent representation of the source machine component.

Invocation mark
If the message has ever resided on an invocation, this field will be non-zero. It contains the mark of the invocation where the message was last queued.

Activation group mark

If the message has ever resided on an invocation. this field will be non-zero. It contains the mark of the activation group which contains the invocation where the message was last queued.

The template identified by *operand 2* is used to contain the materialized message. It must be 16-byte aligned with the following format.

- Template size
  - Number of bytes provided for materialization — Bin(4)
  - Number of bytes available for materialization — Bin(4)
- Message type — Char(1)
  - hex 00 = Informational message 0
  - hex 01 = Informational message 1
  - hex 04 = Exception message
  - hex 06 = Return/Transfer Control message
  - hex 07 = Return message

  All other values are reserved.
- Reserved (binary 0) — Char(1)
- Message severity — Bin(2)
- Reply/Inquiry message reference key — Char(4)
- Message status mask — Char(8)
  - Log message — Bit 0

    0 = The message is not queued to the Process Message Log.
    1 = The message is queued to the Process Message Log.
  - Inquiry — Bit 1

    0 = Message will not accept a reply.
    1 = Message will accept a reply.
  - Reply — Bit 2

    0 = Message does not represent a reply message.
    1 = Message represents a reply message.
  - Answered — Bit 3

    0 = For messages with a *status* of *inquiry*, this indicates that a reply message has not been received.
    1 = For messages with a *status* of *inquiry*, this indicates that a reply message has been received.
  - Message being processed — Bit 4

    0 = This value of the flag has no particular meaning.
    1 = For a *message type* of *exception*, this indicates that the interrupt is currently being handled.
  - Retain — Bit 5

    0 = Message will be dequeued and its contents discarded when the following message status bits are FALSE: *Log*, *Message being processed*, and *Action pending*.
    1 = Keeps the message from being dequeued after all other message status bits are FALSE.

- Action pending — Bit 6

  0 = Indicates no actions are pending based on this message.
  1 = Indicates that the message is either an interrupt cause or is a Return, Return/Transfer Control message.

- Invoke Process Default Exception Handler (PDEH) — Bit 7

  0 = Do not invoke PDEH.
  1 = Invoke PDEH, only valid if *message type* is *exception*.

- Error — Bit 8

  0 = No error has occurred in the sending of this message.
  1 = An error has occurred in the sending of this message.

- PDEH previously invoked — Bit 9

  0 = PDEH has not previously been invoked for this message.
  1 = PDEH has previously been invoked for this message.

- Reserved (binary 0) — Bits 10-31
- User defined status — Bits 32-63

• Interrupt class mask — Char(8)

- Binary overflow or divide by zero — Bit 0
- Decimal overflow or divide by zero — Bit 1
- Decimal data error — Bit 2
- Floating-point overflow or divide by zero — Bit 3
- Floating-point underflow or inexact result — Bit 4
- Floating-point invalid operand or conversion error — Bit 5
- Other data error (edit mask, etc) — Bit 6
- Specification (operand alignment) error — Bit 7
- Pointer not set/pointer type invalid — Bit 8
- Object not found — Bit 9
- Object destroyed — Bit 10
- Address computation underflow/overflow — Bit 11
- Space not allocated as specified offset — Bit 12
- Domain/State protection violation — Bit 13
- Authorization violation — Bit 14
- Reserved (binary 0) — Bit 15-28
- Other MI generated exception (not function check) — Bit 29
- MI generated function check/machine check — Bit 30
- Message generated by Signal Exception instruction — Bit 31
- Send Process Message instruction — Bit 32
- Send Process Message instruction — Bit 33
- Send Process Message instruction — Bit 34
- Send Process Message instruction — Bit 35
- Send Process Message instruction — Bit 36

– Send Process Message instruction	Bit 37
– Send Process Message instruction	Bit 38
– Send Process Message instruction	Bit 39
– User defined	Bits 40-63
• Initial handler priority	Char(1)
• Current handler priority	Char(1)
• Exception ID	UBin(2)
• PDEH Reason Code	Char(1)
• Reserved (binary 0)	Char(1)
• Compare data length	Bin(2)
• Message ID	Char(7)
• Reserved (binary 0)	Char(1)
• Max message data length	Bin(4)[1]
• Message data length	Bin(4)
• Max length of message extension data	Bin(4)[1]
• Message extension data length	Bin(4)
• Message data pointer	Space pointer[1]
• Message data extension pointer	Space pointer[1]
• Message format information	Char(32)
– If the message is not a *return* or *return/transfer control* message.	
- Compare data	Char(32)
– If the message is a *return* or *return/transfer control* message.	
- Return handler identifier	System or Procedure pointer
- Reserved	Char(16)
• Reserved (binary 0)	Char(32)

The first 4 bytes of the message template identify the total **number of bytes provided** for use the instruction. This number is supplied as input to the instruction and is not modified by the instruction. A number less than 160 causes the *materialization length* (hex 3803) exception.

Message type

This value determines the type of the message. The type of message determines which *message status values* have meaning.

The following *message status* are valid for all *informational message types*.

- Log Message
- Reply
- Inquiry

The following *message status* are valid for a *message type* of *Exception*.

- Log Message

[1] Input to the instruction

- Retain
- Action pending
- Invoke-PDEH
- Inquiry
- Reply

The following *message status* are valid for a *message type* of *return* or *return/transfer control*.

- Action pending

The following describes each *message type* in detail.

- *Informational 0* - There are no special requirements as to what message status contains or what the Target Invocation pointer actually identifies.
- *Informational 1* - There are no special requirements as to what message status contains or what the Target Invocation pointer actually identifies.
- *Exception* - This type of message has an *interrupt cause* for the *interrupeted invocation*.
- *Return* - This message type indicates that a return handler will be invoked if the target invocation is exited for any reason other than Transfer Control (XCTL). In the case of XCTL, the message is preserved and associated with the transferred to invocation. Additionally, messages of this type interpret the *message format information* field to identify a program or procedure to be invoked as the return handler.
- *Return/Transfer control* - This message type indicates that a return handler is invoked when the target invocation is exited for any reason including XCTL. Additionally, messages of this type interpret the *message format information* field to identify a program or procedure to be invoked as the return handler.

Message severity

A value indicating the severity of the message.

Reply/Inquiry message reference key

If the message materialized is an *inquiry* message, that has been answered this is the *message reference key* of its reply message.

If the message materialized is a *reply* message, this is the *message reference key* of its inquiry message.

This value only has meaning for *exception*, and *informational* messages.

Message status mask

A bit-significant value indicating the original status of the message.

- **Log message** status. If this bit is TRUE, then the message is queued to the Process Message Log until it is explicitly removed.
- **Inquiry** status. If this bit is TRUE , then this message will accept a *reply* message.
- **Reply** status. If this value is TRUE , then this message is a Reply to an *inquiry* message. The *reply message reference key* is used to identify the message for which the message was replied.
- **Answered** status. If this value is TRUE, then the message is a inquiry message for which a reply has been sent. The *reply message reference key* is used to identify the reply message.
- **Retain** status. If this bit is TRUE, then the message is kept even in the invocation message queue after the following message status bits are FALSE: *Log*, *Message being processed*, and *Action pending*

- **.Action pending**. status. If this bit is TRUE, this message represents an exception which is the current interrupt cause for the specified Source Invocation or else it is a Return or Return/Transfer Control message which has not yet been processed.
- **Invoke Process Default Exception Handler**. This status only has meaning for *exception* messages.

Interrupt class mask
A bit-significant value indicating the cause of the interrupt. The MI user is allowed to use the machine-defined classes since machine-generated errors may be re-sent by the MI user.

This value only has meaning for *exception* messages.

Initial handler priority
An unsigned eight-bit binary number which selects the initial interrupt handler priority. This value is within the range of of 64 - 255.

This value only has meaning for *exception* messages.

Exception ID
A two-byte field that identifies the exception being defined by this message.

This value only has meaning for *exception* messages.

PDEH Reason Code
A value defined by the user which indicates the type of processing to be attempted by the Process Default Exception Handler.

This value only has meaning for *exception* messages.

Compare data length
A value indicating the number of bytes provided as compare data.

This value only has meaning for *exception* messages.

Message ID
Specifies the message identifier of a message description whose predefined message is being sent.

Max length of message data
Input to the instruction that specifies the number of bytes supplied for the message data. The maximum value allowed is 65504.

Message data length
A value indicating the number of bytes of message data for this process message.

Max length of message data extension
Input to the instruction that specifies the number of bytes supplied for the message data extension. The maximum value allowed is 65504.

Message data extension length
A value indicating the number of bytes of message data extension for this process message.

Message data pointer
A pointer to the area to receive the message data. This field is ignored if the *max length of message data* field is zero.

Message data extension pointer
A pointer to the area to receive the message data extension. This field is ignored if the *max length of message data extension* field is zero.

Message format information
A 32 byte field that contains either compare data or two 16 bytes fields which contain information related to a return type message.

- If *message type* is not a *return* or *return/transfer control* message. this field is defined as compare data used to determine which exception handler is to given control. Up to 32 bytes may be specified.
- If *message type* is a *return* or *return/transfer control* message, then the first 16 bytes of this field are defined as a *system pointer* to an *program* object, or else a *procedure pointer* to a New Model procedure. The last 16 bytes of the field are reserved for future use.

This value is ignored if the message does not represent an *exception, return* or *return/transfer control* message.

The template identified by *operand 3* specifies the source invocation of the message. This operand can be null (which indicates the requesting invocation is to be used for the Source Invocation) or specify either an Invocation pointer to an invocation, a null pointer (which indicates the current invocation), or a pointer to a Process Queue Space. It must be 16-byte aligned with the following format.

• Source invocation offset	Bin(4)
• Originating invocation offset	Bin(4)
• Invocation range	Bin(4)
• Reserved (binary 0)	Char(4)
• Source invocation/Process Queue Space pointer	Invocation/System pointer
• Reserved (binary 0)	Char(16)

Source invocation offset

A signed numerical value indicating an invocation relative to the invocation located by the *source invocation pointer*. A value of zero denotes the the invocation addressed by the *source invocation pointer*, with increasingly positive numbers denoting increasingly later invocations in the stack, and increasingly negative numbers denoting increasingly earlier invocations in the stack. If a Process Queue Space is specified as the message source, then the only valid values for this field are 0, -1 and -2. A value of -1 indicates to materialize from the external queue of the Process Queue Space. A zero value indicates to materialize from the message log of the Process Queue Space. A value of -2 indicates to attempt to locate the message using the *message reference index* supplied in operand 4 without regard to the queue space queue that the message resides on. Only unanswered *inquiry* messages, *return* messages, and *return/transfer control* messages can be materialized in this fashion. Other values result in a *scalar value invalid* (hex 3203) exception being signaled.

If the invocation identified by this offset does not exist in the stack, a *scalar value invalid* (hex 3203) exception will be signaled.

Originator invocation offset

Specifies a displacement from the invocation executing this instruction and must be zero (which indicates the current invocation) or negative (which indicates an older invocation). The invocation identified is used as the source for all authorization checks (environment authority to an invocation or authority to a process queue space). If the *originator invocation offset* is non-zero, then the invocation executing this instruction must be authorized to the originating invocation identified.

If the invocation identified by this offset does not exist in the stack or the value is greater than zero, a *scalar value invalid* (hex 3203) exception will be signaled.

Invocation Range

A signed numerical value indicating the number of invocations in the range in addition to the invocation identified by the Source invocation pointer. If a *Process Queue Space pointer* is provided, this value must be zero.

The sign of the *invocation range* determines the direction of the additional invocations. A positive number specifies a range encompassing newer invocations, while a negative number specifies a range encompassing older invocations.

It is not an error if this value specifies a range greater than the number of existing invocations in the specified direction. The materialization will stop after the last invocation is encountered.

Source invocation pointer

An *invocation pointer* to an invocation. If null, then the current invocation is indicated.

If the invocation identified does not exist in the stack or is invalid for this operation, an *invalid invocation* (hex 1603) exception will be signaled.

Process Queue Space pointer

A *system pointer* to a *Process Queue Space* object.

The template identified by operand 4 must be 16-byte aligned. Following is the format of the message selection template:

- Starting message reference index — Char(4)
- Ending message reference index — Char(4)
- Number of selection criteria — Bin(2)
- Reserved (binary 0) — Char(6)
- Selected message reference index — Char(4)[3]
- Selected message count — Bin(4)[3]
- Status change count — Bin(4)[2]
- Moved message count — Bin(4)[2]
- Selection criterion — Char(32)
  - Selection type — Char(1)

    hex 00 = Select based on message status

    hex 01 = Select based on message ID

    hex 02 = Select based on interrupt class

    hex 03 = Select based on invocation mark

    hex 04 = Select based on activation group mark
  - Reserved (binary 0) — Char(1)
  - Selection action — Char(2)
    - Reject criterion — Bit 0

      0 = Select message if criterion is statisfied
      1 = Reject message if criterion is statisfied
    - Reject message — Bit 1

[2] Ignored by the instruction

[3] Output by the instruction

0 = Do not reject message if criterion is not satisfied
1 = Reject message if criterion is not satisfied

Field	Type
- Reserved (binary 0)	Bits 2 - 15
— Message type mask	Char(4)
— Selection criterion information	Char(24)
If the *selection type* is *message status*	
- Message status mask	Char(8)
- Message status complement	Char(8)
- Reserved (binary 0)	Char(8)
If the *selection type* is *message ID*	
- Message ID	Char(7)
- Reserved (binary 0)	Char(17)
If the *selection type* is *interrupt class*	
- Interrupt class mask	Char(8)
- Interrupt class complement	Char(8)
- Reserved (binary 0)	Char(8)
If the *selection type* is *invocation mark*	
- Invocation mark	UBin(4)
- Reserved (binary 0)	Char(20)
If the *selection type* is *activation group mark*	
- Activation group mark	UBin(4)
- Reserved (binary 0)	Char(20)

Starting and ending message reference index

Messages in the specified range of index values are examined either until one of the selection criteria has been satisfied or all queues specified have been searched. The direction of search is determined by the relative values of Starting Message Reference Index and Ending Message Reference Index. If the former value is smaller, then the search direction is in numerically (and chronologically) increasing order, while if the latter value is smaller the search direction is in the opposite direction.

Messages are examined starting with the message identified by *starting message reference index*, or if no such message exists in the queue, starting with the closest existing message in the direction of the search.

If *operand 3* specifies a system pointer to a a queue space and a *source invocation offset* of -2, then the *starting and ending message reference indices* must be equal, otherwise a *scalar value invalid* (hex 3203) exception will be issued.

Number of selection criteria

A numerical value that specifies how many selection criteria fields are supplied. If *number of selection criteria* has a value of zero, then the first message in the index range will be materialized.

Selected message reference index

This value returns the message reference index of the message materialized. Zero is returned if no message satisfies the criteria.

Selected message count

This value indicates the number of messages selected: zero for no messages found, one if a message was found.

Selection criterion

This field contains the data used to select messages from a queue space. There is a variable number of criteria present in the template (the number present is in the number of selection criteria field). Each selection criterion may select a message, reject it, or take no action. Successive selection criteria are applied to each message until it is selected or rejected, or until selection criteria have been exhausted (in which case selection is the default).

Selection type

This field indicates the format of the selection criterion and what field in the message is compared.

Selection action

A bit-significant value indicating what actions to perform during the selection criteria processing.

- **Reject criterion** - If this bit is TRUE, a message is rejected if the selection criterion is satisfied. If this bit is FALSE, then a message is selected when the selection criterion is satisfied.
- **Reject message** - If this bit is TRUE, a message is rejected if the selection criteria is not satisfied. If this bit is FALSE, then a message is selected when the selection criteria not is satisfied.

Message type mask

A bit-significant value indicating types of messages that should be examined during selection criteria processing. The first 31 bits correspond to message types hex 00 through hex 1E respectively. The 32nd bit (bit 31) corresponds to all message types greater than hex 1E.

Message status mask and message status complement

These are bit-significant values indicating the message status attributes that will allow a message to be selected. These values are used to test a message as follows:

The *message status complement* is bit-wise exclusive-ORed with the message status value of a message. The result of this is then bit-wise ANDed with the *message status mask*. If the result is all FALSE bits, then the message does not satisfy this selection criterion. If the result is not all FALSE, bits then the message satisfies the selection criterion.

Message ID

A character value that is compared to the message ID of a message. If the values are equal the selection is satisfied.

Interrupt class mask and interrupt class complement

These are bit-significant values indicating the interrupt class attributes that will allow a message to be selected. These values are used to test a message as follows:

The *interrupt class complement* is bit-wise exclusive-ORed with the interrupt class value of a message. The result of this is then bit-wise ANDed with the *interrupt class mask*. If the result is all FALSE bits, then the message does not satisfy this selection criterion. If the result is not all FALSE bits, then the message satisfies the selection criterion.

Invocation mark

A binary number that is compared to the invocation mark of a message. If the values are equal the selection is satisfied.

Activation group mark
A binary number that is compared to the activation group mark of a message. If the values are equal the selection is satisfied.

Authorization Required:

The following algorithm is used to determine authorization.

1. The invocation which invoked the MATPRMSG instruction must have authority to the invocation identified as the Source Invocation.
2. The Originating Invocation must have authority to the invocation identified as the Source Invocation or to the invocation directly called by the Source invocation.

If any of the authority checks fail then a *activation group access protection violation* (hex 2C12) exception will be signaled.

- Operational
  - Operand 3
- Retrieve
  - Contexts referenced for address resolution

Lock Enforcement

- Materialize
  - Operand 3
  - Contexts referenced for address resolution

Exceptions

Exception		Operands 1	2	3	4	Other
06	Addressing					
	01 space addressing violation	X	X	X	X	
	02 boundary alignment	X	X	X	X	
	03 range	X	X	X	X	
	06 optimized addressability invalid	X	X	X	X	
08	Argument/parameter					
	01 parameter reference violation	X	X	X	X	
0A	Authorization					
	01 unauthorized for operation	X				X
10	Damage encountered					
	04 system object damage state	X	X	X	X	X
	05 authority verification terminated due to damaged object					X
	44 partial system object damage	X	X	X	X	X
1A	Lock state					
	01 invalid lock state	X				X
1C	Machine-dependent exception					
	03 machine storage limit exceeded	X				X
	04 object storage limit exceeded	X				

Exception		Operands 1	2	3	4	Other
20	Machine support					
	02 machine check					X
	03 function check					X
22	Object access					
	01 object not found	X	X	X	X	
	02 object destroyed	X	X	X	X	
	03 object suspended	X	X	X	X	
	07 authority verification terminated due to destroyed object					X
24	Pointer specification					
	01 pointer does not exist	X	X	X	X	
	02 pointer type invalid	X	X	X	X	
	03 pointer address invalid object	X				
2C	Program execution					
	12 activation group access protection violation				X	
2E	Resource control limit					
	01 user profile storage limit exceeded					X
36	Space management					
	01 space extension/truncation					X
46	Queue space management					
	01 queue space not associated with the process			X		

Extended Function Instructions

These instructions provide an extended set of functions which can be used to control and monitor the operation of the machine. Because of the more complicated nature of these instructions, they are more exposed to changes in their operation in different machine implementations than the basic function instructions. Therefore, it is recommended that, where possible, programs avoid using these extended function instructions to minimize the impacts which can arise in moving to different machine implementations.

© Copyright IBM Corp. 1991, 1993

Chapter 16. Context Management Instructions

This chapter describes the instructions used for context management. These instructions are in alphabetic order. See Appendix A, "Instruction Summary," for an alphabetic summary of all the instructions.

Materialize Context (MATCTX) 16-3

© Copyright IBM Corp. 1991, 1993

Materialize Context (MATCTX)

Op Code (Hex)	Operand 1	Operand 2	Operand 3
0133	Receiver	Permanent context, temporary context, or machine context	Materialization options

Operand 1: Space pointer.

Operand 2: System pointer or null.

Operand 3: Character scalar.

ILE access

```
MATCTX (
      receiver                   : space pointer
  var context                    : system pointer; OR
                                   null operand;
  var materialization_options    : aggregate
)
```

Description: Based on the contents of the materialization options specified by operand 3, the symbolic identification and/or system pointers to all or a selected set of the objects addressed by the context specified by operand 2 are materialized into the receiver specified by operand 1. If operand 2 is null, then the machine context is materialized.

The materialization options operand has the following format:

- Materialization control — Char(2)
  - Information requirements (1 = materialize) — Char(1)
    - Reserved (binary 0) — Bits 0-3
    - Extended context attributes — Bit 4
    - Validation — Bit 5

      0 = Validate system pointers
      1 = No validation
    - System pointers — Bit 6
    - Symbolic identification — Bit 7
  - Selection criteria — Char(1)
    - Reserved (binary 0) — Bits 0-2
    - Modification date/time selection — Bit 3

      0 = Do not select by *modification date/time*

      1 = Select by *modification date/time*
    - Object ID selection — Bits 4-7
      - Hex 0—All entries
      - Hex 1—Type code selection
      - Hex 2—Type code/subtype code selection

- Hex 4 — Name selection
- Hex 5 — Type code/name selection
- Hex 6 — Type code/subtype code/name selection
- Hex E — Context entries collating at and above the specified Type code/subtype code/name selection

• Length of name to be used for search argument	Bin(2)
• Type code	Char(1)
• Subtype code	Char(1)
• Name	Char(30)
• Timestamp	Char(8)

The materialization control **information requirements** field in the materialization options operand specifies the information to be materialized for each selected entry. Symbolic identification and system pointers identifying objects addressed by the context can be materialized based on the bit setting of this parameter.

If the *information requirements* field is binary 0, the context attributes are materialized with no context entries. In this case, the *selection criteria* field is meaningless.

If the *information requirements* field is set to just return the **extended context attributes**, the context attributes and extended attributes are materialized with no context entries. In this case, the *selection criteria* field is meaningless.

If the **validation attribute** indicates no validation is to be performed, no object validation occurs and a significant performance improvement results.

When *no validation* occurs, some of the following pointers may be erroneous:

- Pointers to destroyed objects
- Pointers to objects that are no longer in the context
- Multiple pointers to the same object

The materialization control **selection criteria** field specifies the context entries from which information is to be presented. The **type code**, **subtype code**, and **name** fields contain the selection criteria when a selective materialization is specified.

When *type code* or *type/subtype codes* are part of the selection criteria, only entries that have the specified codes are considered. When a **name** is specified as part of the selection criteria, the N characters in the search criteria are compared against the N characters of the context entry, where N is defined by the **length of name to be used for search argument** field in the materialization options. The remaining characters (if any) in the context entry are not used in the comparison.

Selection criteria value hex 00, when the number of bytes provided in the receiver does not allow for materialization of at least one context entry, requests that as much of the context attributes as will fit be materialized into the receiver and that an estimate of the the byte size correlating to the full list of context entries currently in the context be set into the number of bytes available for materialization field of the receiver. This capability of requesting an estimate of the size of a full materialization of the context provides a low overhead way of getting a close approximation of the amount of space that will be needed for an actual materialize of all context entries. This approximation may be either high or low by a few entries due to abnormal system terminations.

Selection criteria value hex 00, when the number of bytes provided in the receiver allow for materialization of at least one context entry, and values hex 01 through hex 06 request that all context entries matching the associated *type code/subtype code/name* criteria be materialized into the receiver. The number of bytes available for materialization field is set with the byte size correlating to the full list of context entries that matched the selection criteria whether or not the receiver provided enough room for the full list to be materialized.

Selection criteria value hex 0E requests that as many context entries as will fit which collate at or higher (are equal to or greater) than the specified *type code/subtype code/name* criteria be materialized into the receiver. The number of bytes available for materialization field is set with the byte size correlating to the list of context entries that were actually materialized into the receiver rather than the full list that may have been available in the context.

If **modification date/time** selection is specified, then entries are selected according to the time of last modification in addtiion to any object identification selection specified. The **timestamp** in the materialization control is used to determine which entries will be selected. Entries with modification timestamps greater than or equal to the *timestamp* specified in the control will be selected. Besides the additional selection done as above, the materialize will work the same as specified in the other controls.

Programming note: If the specified timestamp is for a date/time earlier than the date/time currently associated with the changed object list, all objects in the context will be inspected for their modification date. This may degrade system performance.

The format of the materialization (operand 1) is as follows:

• Materialization size specification	Char(8)
— Number of bytes provided for materialization	Bin(4)
— Number of bytes available for materialization	Bin(4)
• Context identification	Char(32)
— Object type	Char(1)
— Object subtype	Char(1)
— Object name	Char(30)
• Context options	Char(4)
— Existence attributes 0 = Temporary 1 = Permanent	Bit 0
— Space attribute 0 = Fixed-length 1 = Variable-length	Bit 1
— Reserved (binary 0)	Bit 2
— Access group 0 = Not a member of access group 1 = Member of access group	Bit 3
— Reserved (binary 0)	Bits 4-31
• Recovery options	Char(4)
— Automatic damaged context rebuild option 0 = Do not rebuild at IMPL	Bit 0

1 = Rebuild at IMPL

• Size of space	Bin(4)
• Initial value of space	Char(1)
• Performance class	Char(4)
— Space alignment	Bit 0

0 = The space associated with the object is allocated to allow proper alignment of pointers at 16-byte alignments within the space. If no space is specified for the object, this value must be specified for the performance class.
1 = The space associated with the object is allocated to allow proper alignment of pointers at 16-byte alignments within the space as well as to allow proper alignment of input/output buffers at 512-byte alignments within the the space.

— Reserved (binary 0)	Bits 1-4
— Main storage pool selection	Bit 5

0 = Process default main storage pool is used for object.
1 = Machine default main storage pool is used for object.

— Reserved (binary 0)	Bit 6
— Block transfer on implicit access state modification	Bit 7

0 = Transfer the minimum storage transfer size for this object. This value is 1 storage unit.
1 = Transfer the machine default storage transfer size. This value is 8 storage units.

— Reserved (binary 0)	Bits 8-31
• Reserved (binary 0)	Char(7)
• Reserved (binary 0)	Char(16)
• Access group	System pointer
• Extended Context Attributes (if requested)	Char(1)
— Changed object list	Bit 0

0 - A changed object list does not exist

1 - A changed object list does exist.

— Useable changed object list	Bit 1

0 - Changed object list is in a useable state

1 - Changed object list is not in a useable state

— Reserved (binary 0)	Bit 2-7
• Reserved (binary 0)	Char(7)
• Current timestamp	Char(8)
• Context entry (repeated for each selected entry)	Char(16-48)
— Object identification (if requested)	Char(32)
- Type code	Char(1)
- Subtype code	Char(1)
- Name	Char(30)
— Object pointer (if requested)	System pointer

The first 4 bytes of the materialization output identify the total **number of bytes provided** for use by the instruction. This value is supplied as input to the instruction and is not modified by the instruction. A value of less than 8 causes the *materialization length* (hex 3803) exception to be signaled. The instruction materializes as many bytes and pointers as can be contained in the receiver. If the byte area identified by the receiver is greater than that required to contain the information requested for materialization, the excess bytes are unchanged. No exceptions are signaled in the event that the receiver contains insufficient area for the materialization, other than the *materialization length* (hex 3803) exception described above.

See the Create Context (CRTCTX) for a description of the common object creation fields.

The **context entry object identification** information, if requested by the *materialization options* field, is present for each entry in the context that satisfies the search criteria. If both *system pointers* and *symbolic identification* are requested by the *materialization options* field, the *system pointer* immediately follows the *object identification* for each entry.

The order of the materialization of a context is by object type code, object subtype code, and object name, all in ascending sequence.

Authorization Required

- Retrieve
  - Operand 2

Lock Enforcement

- Materialization
  - Operand 2

Exceptions

Exception		Operands 1	2	3	Other
06	Addressing				
	01 Space addressing violation	X	X	X	
	02 Boundary alignment	X	X	X	
	03 Range	X	X	X	
	06 Optimized addressability invalid	X	X	X	
08	Argument/parameter				
	01 Parameter reference violation	X	X	X	
0A	Authorization				
	01 Unauthorized for operation		X		
10	Damage encountered				
	02 Machine context damage state				X
	04 System object damage state	X	X	X	X
	05 authority verification terminated due to damaged object				X
	44 Partial system object damage	X	X	X	X
1A	Lock state				
	01 Invalid lock state		X		

Exception		Operands 1	2	3	Other
1C	Machine-dependent exception				
	03 Machine storage limit exceeded		X		X
20	Machine support				
	02 Machine check				X
	03 Function check				X
22	Object access				
	01 Object not found	X	X	X	
	02 Object destroyed	X	X	X	
	03 Object suspended	X	X	X	
	07 authority verification terminated due to destroyed object				X
	08 object compressed				X
24	Pointer specification				
	01 Pointer does not exist	X	X	X	
	02 Pointer type invalid	X	X	X	
	03 Pointer addressing invalid object		X		
2E	Resource control limit				
	01 user profile storage limit exceeded				X
32	Scalar specification				
	02 Scalar attributes invalid			X	
	03 Scalar value invalid			X	
36	Space management				
	01 space extension/truncation				X
38	Template specification				
	03 Materialization length exception	X			

Chapter 17. Authorization Management Instructions

This chapter describes the instructions used for authorization management. These instructions are in alphabetic order. For an alphabetic summary of all the instructions, see Appendix A, "Instruction Summary."

Materialize Authority (MATAU) 17-3
Materialize Authority List (MATAL) 17-7
Materialize Authorized Objects (MATAUOBJ) 17-12
Materialize Authorized Users (MATAUU) 17-20
Materialize User Profile (MATUP) 17-25
Test Authority (TESTAU) 17-29
Test Extended Authorities (TESTEAU) 17-34

© Copyright IBM Corp. 1991, 1993

Materialize Authority (MATAU)

Op Code (Hex)	Operand 1	Operand 2	Operand 3
0153	Receiver	System object	User profile or Source Template

Operand 1: Space pointer.

Operand 2: System pointer.

Operand 3: System pointer or Space pointer data object or null.

ILE access

```
MATAU (
      receiver                          : space pointer;
  var system_object                     : system pointer;
  var user_profile_or_source_template : system pointer OR
                                        space pointer OR
                                        null operand
)
```

Description: This instruction materializes the specific types of authority for a system object available to the specified user profile. The private authorization that the user profile specified by operand 3 has to the permanent system object specified by operand 2, and the object's public authorization is materialized in operand 1. If operand 3 is null, then only the object's public authorization is materialized, and the *private authorization* field in the materialization is set to binary 0.

Except for certain special cases, the authority to be materialized is determined by first checking for direct authority to the object itself, then checking for indirect authority to the object through authority to an authorization list containing the object. The first source of authority found is materialized and the source is indicated in the materialization.

The special case of the operand 3 user profile having *all object special authority* overrides any explicit private authorities that the user profile might hold to the object or its containing authorization list and results in a materialization showing that the profile holds all private authorities directly to the object.

The special case of the operand 2 object being in an authorization list which has the *override specific object authority* attribute in effect results in the authorization or lack of authorization held to the authorizaiton list completely overriding the explicit private authorities that the user profile might hold to the object. This case results in a materialization showing that the profile has just the private authorities it holds or doesn't hold to the authorization list. That is, if the user profile has private authority to the object, but doesn't have private authority to the authorization list, the materialization will show that the user does not have any private authority to the object. Similarly, if the user profile has both private authority to the object and to the authorization list, the materialization will show that the user has only the private authority through the authorization list. If operand 3 is null, then only the object's public authorization is materialized, and the private authorization field in the materialization is set to binary zeros.

Operand 3 may be specified as a a system pointer which directly addresses the user profile to be checked as a source of authority or as a space pointer to a source template which identifies the source user profile. Specifying a template allows for additional controls over how the materialize operation is to be performed. The format of the source template is the following:

- Source flags — Char(2)

– Ignore all object special authority 0 = No 1 = Yes	Bit 0
– Reserved (binary 0)	Bit 1-15
• Reserved (binary 0)	Char(14)
• User profile	System pointer

The **ignore all object special authority** source flag specifies whether or not that special authority is to be ignored during the materialize operation. When *yes* is specified, just the explicitly held private authority that the specified user profile holds either directly to the object or indirectly to an authorization list containing the object will be materialized. When *no* is specified, the authority provided by all object special authority, if held by the source user profile, is included and results in a materialization showing that the profile holds all private authorities directly to the object. *No* is the default for this flag value when the source template is not specified.

The **user profile** field specifies the user profile to be checked as a source of authority.

The format of the materialization (operand 1) is as follows:

• Materialization size specification	Char(8)
– Number of bytes provided for materialization	Bin(4)
– Number of bytes available for materialization (contains a value of 16 for this instruction)	Bin(4)
• Private authorization (1 = authorized)	Char(2)
– Object control	Bit 0
– Object management	Bit 1
– Authorized pointer	Bit 2
– Space authority	Bit 3
– Retrieve	Bit 4
– Insert	Bit 5
– Delete	Bit 6
– Update	Bit 7
– Ownership (1 = yes)	Bit 8
– Excluded	Bit 9
– Authority List management	Bit 10
– Reserved (binary 0)	Bit 11-15
• Public authorization (1 = authorized)	Char(2)
– Object control	Bit 0
– Object management	Bit 1
– Authorized pointer	Bit 2
– Space authority	Bit 3
– Retrieve	Bit 4
– Insert	Bit 5
– Delete	Bit 6

— Update	Bit 7
— Reserved (binary 0)	Bit 8
— Excluded	Bit 9
— Authority List management	Bit 10
— Reserved (binary 0)	Bit 11-15
• Private authorization source	UBin(2)

0 = authority to object
1 = authority to authorization list

• Public authorization source	UBin(2)

0 = authority from object
1 = authority from authorization list

The first 4 bytes of the materialization identify the total **number of bytes provided** for use by the instruction. This value is supplied as input to the instruction and is not modified by the instruction. A value of less than 8 causes the *materialization length* (hex 3803) exception.

The second 4 bytes of the materialization identify the total **number of bytes available** to be materialized (16 for this instruction). The instruction materializes as many bytes as can be contained in the area specified as the receiver. If the byte area identified by the receiver is greater than that required to contain the information requested, then the excess bytes are unchanged. No exceptions (other than the materialization length exception) are signaled in the event that the receiver contains insufficient area for the materialization.

Any of the four authorizations- **retrieve**, **insert**, **delete**, or **update**-constitute operational authority.

If this instruction references a temporary object, all public authority states are materialized. Private authority states are not materialized.

Authorization Required

- Operational
  - Operand 3
- Retrieve
  - Contexts referenced for address resolution

Lock Enforcement

- Materialize
  - Operand 2
  - Operand 3
  - Contexts referenced for address resolution

Exceptions

Exception		Operands 1	2	3	Other
06	Addressing				
	01 space addressing violation	X	X	X	
	02 boundary alignment	X	X	X	

Exception		Operands 1	2	3	Other
	03 range	X	X	X	
	06 optimized addressability invalid	X	X	X	
08	Argument/parameter				
	01 parameter reference violation	X	X	X	
0A	Authorization				
	01 unauthorized for operation		X	X	
10	Damage encountered				
	02 machine context damage state				X
	04 system object damage state	X	X	X	X
	05 authority verification terminated due to damaged object				X
	44 partial system object damage	X	X	X	X
1A	Lock state				
	01 invalid lock state		X	X	
1C	Machine-dependent exception				
	03 machine storage limit exceeded				X
20	Machine support				
	02 machine check				X
	03 function check				X
22	Object access				
	01 object not found	X	X	X	
	02 object destroyed	X	X	X	
	03 object suspended	X	X	X	
	07 authority verification terminated due to destroyed object				X
	08 object compressed				X
24	Pointer specification				
	01 pointer does not exist	X	X	X	
	02 pointer type invalid	X	X	X	
	03 pointer addressing invalid object		X	X	
2E	Resource control limit				
	01 user profile storage limit exceeded				X
36	Space management				
	01 space extension/truncation				X
38	Template specification				
	03 materialization length exception	X			

Materialize Authority List (MATAL)

Op Code (Hex)	Operand 1	Operand 2	Operand 3
01B3	Receiver	Authorization List	Materialization Options

Operand 1: Space pointer.

Operand 2: System pointer.

Operand 3: Space pointer.

ILE access

```
MATAL (
      receiver                    : space pointer
  var authorization_list          : system pointer
      materialization_options     : space pointer
)
```

Description: Based on the contents of the materialization options specified by operand 3, the symbolic identification and/or system pointers to all or a selected set of the objects contained in the authorization list specified by operand 2 are materialized into the receiver specified by operand 1.

The materialization options operand has the following format:

- Materialization control — Char(2)
  - Information requirements — Char(1)
    - 12 = Materialize count of entries matching the criteria.
    - 22 = Materialize identification of entries matching the criteria and return information using short description format
    - 32 = Materialize identification of entries matching the criteria and return information using long description format
  - Selection criteria — Char(1)
    - 00 = All authorization list entries
    - 01 = Type code selection
    - 02 = Type code/subtype code selection
- Reserved (binary 0) — Bin(2)
- Type code — Char(1)
- Subtype code — Char(1)
- Reserved (binary 0) — Char(30)

The **information requirements** field specifies the type of materialization, just a *count of entries*, *short descriptions*, or *long descriptions*, which is being requested.

The **selection criteria** field specifies the criteria to be used in selecting the authorization list entries for which information is to be presented. The **type code** and **subtype code** fields contain the selection criteria when a selective materialization is specified.

When *type code* or *type/subtype codes* are part of the selection criteria, only entries that have the specified codes are considered.

The format of the materialization (operand 1) is as follows:

• Materialization size specification	Char(8)
– Number of bytes provided for materialization	Bin(4)
– Number of bytes available for materialization	Bin(4)
• Authorization List identification	Char(32)
– Object type	Char(1)
– Object subtype	Char(1)
– Object name	Char(30)
• Authorization List creation options	Char(4)
– Existence attributes 1 = Permanent (always permanent)	Bit 0
– Space attribute 0 = Fixed length 1 = Variable length	Bit 1
– Reserved (binary 0)	Bit 2-31
• Reserved (binary 0)	Char(4)
• Size of space	Bin(4)
• Initial value of space	Char(1)
• Performance class	Char(4)
• Reserved	Char(7)
• Context	System pointer
• Reserved	Char(16)
• Authorization List attributes	Char(4)
– Override specific object authority 0 = No 1 = Yes	Bit 0
– Reserved (binary 0)	Bit 1-31
• Reserved (binary 0)	Char(28)
• Entries header	Char(16)
– Number of entries available	UBin(4)
– Reserved	Char(12)

If no description (*information requirements* = hex 12) is requested in the materialization options parameter, the above constitutes the information available for materialization. If a description (short or long) is requested by the materialization options operand, a description entry is present (assuming a sufficient size receiver) for each object materialized into the receiver. Either of the following entry formats may be selected.

• Short description entry	Char(32)

— Type code	Char(1)
— Subtype code	Char(1)
— Reserved	Char(14)
— System object	System pointer
• Long description entry	Char(128)
— Type code	Char(1)
— Subtype code	Char(1)
— Object name	Char(30)
— Reserved	Char(16)
— System object	System pointer
— Object owning user profile	System pointer
— Context	Char(48)
- Type code	Char(1)
- Subtype code	Char(1)
- Context name	Char(30)
- Context pointer	System pointer

The first four bytes of the materialization output identify the total **number of bytes provided** for use by the instruction. This value is supplied as input to the instruction and is not modified by the instruction. A value of less than 8 causes the *materialization length* (hex 3803) exception to be signaled.

The second 4 bytes of the materialization identify the total **number of bytes available** to be materialized. The instruction materializes as many bytes and pointers as can be contained in the receiver. If the byte area identified by the receiver is greater than that required to contain the information requested for materialization, the excess bytes are unchanged. No exceptions are signaled in the event that the receiver contains insufficient area for the materialization, other than the *materialization length* (hex 3803) exception signaled above.

Refer to the Create Authorization List instruction for a discussion of the creation attributes materialized in the above template.

The **number of entries available** field specifies the number of authorization list entries which satisfied the selection criteria and were therefore materialized. A value of zero indicates no entries were available.

The object identification information (in the short and long description entries), if requested by the materialization options parameter, is present for each entry in the authorization list that satisfies the search criteria.

The object pointer information (in the long description entry only), if requested by the materialization options parameter, is present for each entry in the authorization list that satisfies the search criteria.

If the object addressed by the system pointer is not addressed by a context, the *context type* field is set to hex 00 or if the object is addressed by the machine context, the *context type* field is set to hex 81. Additionally, in either of these cases, the *context pointer* is set to the system default "pointer does not exist" value.

Authorization Required

- Retrieve

  Operand 2

Lock Enforcement

- Materialization

  Operand 2

Exceptions

Exception		Operands 1	2	3	Other
06	addressing				
	01 space addressing violation	X	X	X	
	02 boundary alignment violation	X	X	X	
	03 range	X	X	X	
	06 optimized addressability invalid	X	X	X	
08	Argument/parameter				
	01 parameter reference violation	X	X	X	
0A	Authorization				
	01 unauthorized for operation		X		
10	Damage encountered				
	04 system object damage		X		X
	05 authority verification terminated due to damaged object				X
	44 partial system object damage				X
1A	Lock state				
	01 invalid lock state		X		
1C	Machine dependent exception				
	03 machine storage limit exceeded		X		X
20	Machine support				
	02 machine check				X
	03 function check				X
22	Object access				
	01 object not found	X	X	X	
	02 object destroyed	X	X	X	
	03 object suspended	X	X	X	
	07 authority verification terminated due to destroyed object				X
	08 object compressed				X
24	Pointer specification				
	01 pointer does not exist	X	X	X	
	02 pointer type invalid	X	X	X	

Exception		Operands 1	2	3	Other
	03 pointer addressing invalid object		X		
2E	Resource control limit				
	01 user profile storage limit exceeded				X
32	Scalar specification				
	01 scalar type invalid	X	X	X	
	02 scalar attributes invalid			X	
	03 scalar value invalid			X	
36	Space management				
	01 space extension/truncation				X
38	Template specification				
	03 materialization length	X			

Materialize Authorized Objects (MATAUOBJ)

Op Code (Hex)	Operand 1	Operand 2	Operand 3
013B	Receiver	User profile	Materialization options/template

Operand 1: Space pointer.

Operand 2: System pointer.

Operand 3: Character scalar (fixed length).

ILE access

```
MATAUOBJ (
        receiver                 : space pointer;
    var user_profile             : system pointer;
    var materialization_options  : aggregate
)
```

Description: This instruction materializes the identification and the system pointers to all or selected system objects that are privately owned and/or authorized by a specified user profile. For the user profile (operand 2), the materialization options (operand 3) specify object selection criteria and the format and location of the object materialization data. The receiver space (operand 1) always indicates the number of objects materialized, and contains the object materialization data unless the materialization options specify an independent index to contain the data.

When the high-order bit of the materialization options is off, operand 3 is viewed as a Char(1) scalar. This option does not permit object selection by type and subtype, does not allow a continuation point to be specified, and returns all object materialization data in the receiver (operand 1). Following are the valid operand 3 values which may be used with the short template header format (operand 1):

Value (hex)	Meaning
11	Materialize count of owned objects.
12	Materialize count of authorized objects.
13	Materialize count of all authorized and owned objects.
21	Materialize identification of owned objects using short description entry format.
22	Materialize identification of authorized objects using short description entry format.
23	Materialize identification of all authorized and owned objects using short description entry format.
31	Materialize identification of owned objects using long description entry format.
32	Materialize identification of authorized objects using long description entry format.
33	Materialize identification of all authorized and owned objects using long description entry format.

Following are the valid operand 3 values which may be used with the long template header format (operand 1):

Value (hex)	Meaning
51-53	These long template header materialization options are the same as the short template header materialization options 11-13 (hex).
61-63	These long template header materialization options are the same as the short template header materialization options 21-23 (hex).
71-73	These long template header materialization options are the same as the short template header materialization options 31-33 (hex) except that the context extension is materialized for each object as well.

When the high-order bit of the materialization options is on, operand 3 is viewed as variable-length, must be 16-byte aligned in the space, and has the following format:

- Materialization options — Char(1)

  Valid values are hex 91-93, A1-A3, B1-B3, D1-D3, E1-E3, and F1-F3. They have the same meanings as the corresponding values hex 11 through 73.

- Materialization flags — Char(1)

  - Restrict information dcope — Bit 0

    This is an input bit which only has meaning when materialization data is being returned in the operand 1 receiver template. When there is more data to be materialized than can be contained in the template, then when this bit is *on*, the *number of bytes available for materialization*, the *number of objects owned by the user profile*, and the *number of objects privately authorized by the user profile* output fields are restricted to reflect only the information returned in the template; when *off*, the output fields reflect the total amount of materialization data available, even though the template may not be large enough to contain it all.

  - More materialization data available — Bit 1

    This output bit has meaning only when materialization data is being returned in the operand 1 receiver template. When *on*, it indicates that objects exist beyond those for which materialization data was returned in the template; when off it indicates the end of the objects was reached.

  - Continuation point specified — Bit 2

    This is an input bit. When *on*, it indicates that a continuation point is specified in the *continuation point* field; when off, continuation processing is ignored.

  - Reserved (binary 0) — Bits 3-7

- Reserved (binary 0) — Char(30)

- Independent Index pointer — System pointer

  If the pointer does not exist, the instruction returns all object materialization data in the receiver (operand 1). Otherwise it returns only the template header in the receiver and returns the object materialization data in the independent index.

- Continuation point — Char(16) or System pointer

  If the *continuation point specified* bit is *on*, when the instruction begins, if this field contains a system pointer or the storage form of a system pointer, then materialization data is returned for objects found in the profile following the object identified by the continuation point; otherwise, materialization data is returned beginning with the object which is logically first.

- Object type/subtype range array — Bin(2) dimension

Indicates the number of *object type/subtype ranges* specified in the array immediately following. If zero, objects of all types and subtypes are materialized. If larger than zero, only objects included in one or more of the *type/subtype ranges* specified in the array are materialized.

- Object type/subtype array — n * Char(4)

  An array of object type/subtype ranges qualifying the objects materialized. Each array element represents a range of *object type/subtypes* and has the following format:

  - Start of range — Char(2)
    - Object type code — Char(1)
    - Object subtype code — Char(1)
  - End of range — Char(2)
    - Object type code — Char(1)
    - Object subtype code — Char(1)

All materialization options for owned objects with descriptions (hex 21, 23, 31, 33, 61, 63, 71, 73, A1, A3, B1, B3, E1, E3, F1, and F3) also verify the user profile's storage utilization, unless the extended form of operand 3 is used and a valid continuation point is specified.

The order of materialization is owned objects (if requested by the materialization options operand) followed by objects privately authorized to the user profile (if requested by the materialization options operand). No authorizations are stored in the system pointers that are returned.

The template identified by operand 1 must be 16-byte aligned in the space. For options hex 11 through hex 33 and hex 91 through hex B3, the **short template header** is materialized. It has the following format:

- Materialization size specification — Char(8)
  - Number of bytes provided for materialization — Bin(4)
  - Number of bytes available for materialization — Bin(4)
- Number of objects owned by user profile — Bin(2)
- Number of objects privately authorized to user profile — Bin(2)
- Reserved (binary 0) — Char(4)

For options hex 51 through 77 and hex D1 through hex F7, the **long template header** is materialized. It has the following format:

- Materialization size specification — Char(8)
  - Number of bytes provided for materialization — Bin(4)
  - Number of bytes available for materialization — Bin(4)
- Number of objects owned by user profile — Bin(4)
- Number of objects privately authorized to user profile — Bin(4)

The first 4 bytes of the materialization identify the total **number of bytes provided** for use by the instruction. This value is supplied as input to the instruction and is not modified by the instruction. A value of less than 8 causes the *materialization length* (hex 3803) exception to be signaled.

The second 4 bytes of the materialization identify the total **number of bytes available** to be materialized. The instruction materializes as many bytes as can be contained in the area specified as the

receiver. If the byte area identified by the receiver is greater than that required to contain the information requested, then the excess bytes are unchanged. No exceptions (other than the *materialization length* (hex 3803) exception) are signaled in the event that the receiver contains insufficient area for the materialization. If the *restrict information scope* flag is 1, then the field contains the number of bytes materialized, rather than the number of bytes available to be materialized.

If the **restrict information scope** flag is 1, then the *number of objects owned by user profile* and the *number of objects privately authorized by user profile* fields reflect the number of objects for which complete materialization data is returned, rather than the total number of such objects.

If no description is requested in the *materialization options* field, the above constitutes the information available for materialization. If a description (short, long, or long with context extension) is requested by the *materialization options* field, a description entry is present for each object materialized into the receiver (assuming it is of sufficient size) or into the independent index. Object materialization data is in one of the following formats depending on the materialization options and the object into which it is materialized:

• Short description entry materialized into receiver	Char(32)
— Object type code	Char(1)
— Object subtype code	Char(1)
— Private authorization	Char(2)
— Reserved (binary 0)	Char(12)
— Object pointer	System pointer
• Long description entry materialized into receiver	Char(64)
— Object type code	Char(1)
— Object subtype code	Char(1)
— Object name	Char(30)
— Private authorization	Char(2)
— Public authorization	Char(2)
— Reserved (binary 0)	Char(12)
— Object pointer	System pointer
• Long description entry with context extension materialized into receiver	Char(112)
— Object type code	Char(1)
— Object subtype code	Char(1)
— Object name	Char(30)
— Private authorization	Char(2)
— Public authorization	Char(2)
— Reserved (binary 0)	Char(12)
— Object pointer	System pointer
— Context type code	Char(1)
— Context subtype code	Char(1)
— Context name	Char(30)

– Context pointer	System pointer
• Short description entry materialized into independent index	Char(32)
– Entry type code hex 40 = Owned object hex 80 = Authorized object	Char(1)
– Object type code	Char(1)
– Object subtype code	Char(1)
– Private authorization	Char(2)
– Reserved (binary 0)	Char(11)
– Object pointer	System pointer
• Long description entry materialized into independent index	Char(64)
– Entry type code hex 40 = Owned object hex 80 = Authorized object	Char(1)
– Object type code	Char(1)
– Object subtype code	Char(1)
– Object name	Char(30)
– Private authorization	Char(2)
– Reserved (binary 0)	Char(2)
– Public authorization	Char(2)
– Reserved (binary 0)	Char(9)
– Object pointer	System pointer
• Long description entry with context extension materialized into independent index	Char(112)
– Entry type code hex 40 = Owned object hex 80 = Authorized object	Char(1)
– Context type code	Char(1)
– Context subtype code	Char(1)
– Context name	Char(30)
– Object type code	Char(1)
– Object subtype code	Char(1)
– Object name	Char(30)
– Private authorization	Char(2)
– Reserved (binary 0)	Char(2)
– Public authorization	Char(2)
– Reserved (binary 0)	Char(9)
– Object pointer	System pointer

- Context pointer — System pointer

Following is the format of the authorization information:

- Private authorization (1 = authorized) — Char(2)
  - Object Control — Bit 0
  - Object Management — Bit 1
  - Authorized Pointer — Bit 2
  - Space Authority — Bit 3
  - Retrieve — Bit 4
  - Insert — Bit 5
  - Delete — Bit 6
  - Update — Bit 7
  - Ownership (1 = yes) — Bit 8
  - Excluded — Bit 9
  - Authority List management — Bit 10
  - Reserved (binary 0) — Bits 11-15
- Public authorization (1 = authorized) — Char(2)
  - Object Control — Bit 0
  - Object Management — Bit 1
  - Authorized Pointer — Bit 2
  - Space Authority — Bit 3
  - Retrieve — Bit 4
  - Insert — Bit 5
  - Delete — Bit 6
  - Update — Bit 7
  - Reserved (binary 0) — Bit 8
  - Excluded — Bit 9
  - Authority List management — Bit 10
  - Reserved (binary 0) — Bits 11-15

When context information is materialized, if the object addressed by the system pointer is not addressed by a context, the *context type* field is set to hex 00 or if the object is addressed by the machine context, the *context type* field is set to hex 81. Additionally, in either of these cases, the *context pointer* is set to the system default pointer does not exist value.

When the **more materialization data available** flag is 1, the pointer to the object within the last entry in the operand 1 receiver template may be specified as the continuation point on a subsequent invocation of this instruction, to cause materialization to continue, starting with the "logically next" object. To determine whether the continuation point is within the owned or authorized objects, the *ownership bit* in the private authorizations of the last materialized object may be tested. This instruction does not guarantee an atomic snapshot of the user profile across a continuation request.

The following considerations apply when object materialization data is returned in an independent index:

- System pointers returned in index entries are not set unless the index is created to contain both pointer and scalar data.
- Entry data may be truncated or padded on the right with hex zeroes to conform to the index's key and/or fixed entry lengths.
- An entry is added to the index for each qualifying object. Previously existing entries which are thereby duplicated are replaced.
- In order to ensure that index entries inserted within the same execution of this instruction are not duplicates of each other, the *index entry length* (if fixed) and *key length* (if keyed) must be sufficiently large to include the object pointer within the entry data.

Authorization Required

- Operational
  - Operand 2
- Retrieve
  - Contexts referenced for address resolution
  - Operand 2 if materializing owned objects
- Insert
  - Independent index if identified by operand 3

Lock Enforcement

- Materialize
  - Contexts referenced for address resolution
  - Operand 2 if materializing owned objects
- Modify
  - Independent index if identified by operand 3

Exceptions

Exception		Operands 1	2	3	Other
06	Addressing				
	01 space addressing violation	X	X	X	
	02 boundary alignment	X	X	X	
	03 range	X	X	X	
08	Argument/parameter				
	01 parameter reference violation	X	X	X	
0A	Authorization				
	01 unauthorized for operation		X	X	
10	Damage encountered				
	02 machine context damage state				X
	04 system object damage state	X	X	X	X
	05 authority verification terminated due to damaged object			X	X
	44 partial system object damage	X	X	X	X

Exception		Operands 1	2	3	Other
1A	Lock state				
	01 invalid lock state		X	X	
1C	Machine-dependent exception				
	03 machine storage limit exceeded				
	04 object storage limit exceeded			X	
20	Machine support				
	02 machine check				X
	03 function check				X
22	Object access				
	01 object not found	X	X	X	
	02 object destroyed	X	X	X	
	03 object suspended	X		X	
	04 object not eligible for operation	X	X		
	07 authority verification terminated due to destroyed object			X	X
	08 object compressed				X
24	Pointer specification				
	01 pointer does not exist	X	X	X	
	02 pointer type invalid	X	X	X	
	03 pointer addressing invalid object		X	X	
2E	Resource control limit				
	01 user profile storage limit exceeded			X	X
32	Scalar specification				
	03 scalar value invalid			X	
36	Space management				
	01 space extension/truncation				X
38	Template specification				
	01 template value invalid			X	
	03 materialization length exception	X			

Materialize Authorized Users (MATAUU)

Op Code (Hex)	Operand 1	Operand 2	Operand 3
0143	Receiver	System object	Materialization options

Operand 1: Space pointer.

Operand 2: System pointer.

Operand 3: Character(1) scalar (fixed-length).

ILE access

```
MATAUU (
       receiver                    : space pointer;
   var system_object               : system pointer;
   var materialization_options : aggregate
)
```

Description: The instruction materializes the authorization states and the identification of the user profile(s). The materialization options (operand 3) for the system object (operand 2) are returned in the receiver (operand 1). The materialization options for operand 3 have the following format:

Value (Hex)	Meaning
11	Materialize public authority.
12	Materialize public authority and number of privately authorized profiles.
21	Materialize identification of owning profile using short description entry format.
22	Materialize identification of privately authorized profiles using short description entry format.
23	Materialize identification of owning and privately authorized profiles using short description entry format.
31	Materialize identification of owning profile using long description entry format.
32	Materialize identification of privately authorized profiles using long description entry format.
33	Materialize identification of owning and privately authorized profiles using long description entry format.

The order of materialization is an entry for the owning user profile (if requested by the materialization options operand) followed by a list (0 to n entries) of entries for user profiles having private authorization to the object (if requested by the materialization options operand). The authorization field within the system pointers will not be set.

The template identified by operand 1 must be 16-byte aligned in the space and has the following format:

- Materialization size specification — Char(8)
  - Number of bytes provided for materialization — Bin(4)
  - Number of bytes available for materialization — Bin(4)
- Public authorization — Char(2)

(1 = authorized)

- Object control — Bit 0
- Object management — Bit 1
- Authorized pointer — Bit 2
- Space authority — Bit 3
- Retrieve — Bit 4
- Insert — Bit 5
- Delete — Bit 6
- Update — Bit 7
- Reserved (binary 0) — Bit 8
- Excluded — Bit 9
- Authority List management — Bit 10
- Reserved (binary 0) — Bits 11-15

• Number of privately authorized user profiles — Bin(2)

• Reserved (binary 0) — Char(4)

The first 4 bytes of the materialization identify the total **number of bytes provided** for use by the instruction. This value is supplied as input to the instruction and is not modified by the instruction. A value of less than 8 causes the *materialization length* (hex 3803) exception.

The second 4 bytes of the materialization identify the total **number of bytes available** to be materialized. The instruction materializes as many bytes as can be contained in the area specified as the receiver. If the byte area identified by the receiver is greater than that required to contain the information requested, then the excess bytes are unchanged. No exceptions (other than the *materialization length* (hex 3803) exception) are signaled in the event that the receiver contains insufficient area for the materialization.

If no description is requested by the *materialization options* field, the template identified by operand 1 constitutes the information available for materialization. If a description (short or long) is requested by the *materialization options* field, a description entry is present (assuming there is a sufficient sized receiver) for each user profile materialized or available to be materialized into the receiver. Either of the following entry types may be selected.

• Short description entry — Char(32)

- User profile type code — Char(1)
- User profile subtype code — Char(1)
- Private authorization (1 = authorized) — Char(2)
  - Object control — Bit 0
  - Object management — Bit 1
  - Authorized pointer — Bit 2
  - Space authority — Bit 3
  - Retrieve — Bit 4
  - Insert — Bit 5
  - Delete — Bit 6

- Update	Bit 7
- Ownership (1 = yes)	Bit 8
- Excluded	Bit 9
- Authority List management	Bit 10
- Reserved (binary 0)	Bits 11-15
— Reserved (binary 0)	Char(12)
— User profile	System pointer
• Long description entry	Char(64)
— User profile type code	Char(1)
— User profile subtype code	Char(1)
— User profile name	Char(30)
— Private authorization (1 = authorized)	Char(2)
- Object control	Bit 0
- Object management	Bit 1
- Authorized pointer	Bit 2
- Space authority	Bit 3
- Retrieve	Bit 4
- Insert	Bit 5
- Delete	Bit 6
- Update	Bit 7
- Ownership (1 = yes)	Bit 8
- Excluded	Bit 9
- Authority List Management	Bit 10
- Reserved (binary 0)	Bits 11-15
— Reserved (binary 0)	Char(14)
— User profile	System pointer

If this instruction references a temporary object, all public authority states are materialized. The privately authorized user and owner profile(s) descriptions are not materialized (binary 0).

Authorization Required

- Retrieve
  - Contexts referenced for address resolution
- Object management or ownership
  - Operand 2 object (when object is not an authorization list)
- Authorization list management or ownership
  - Operand 2 object (when object is an authorization list)

Lock Enforcement

- Materialize
  - Operand 2
  - Contexts referenced for address resolution

Exceptions

Exception		Operands 1	2	3	Other
06	Addressing				
	01 space addressing violation	X	X	X	
	02 boundary alignment	X	X	X	
	03 range	X	X	X	
	06 optimized addressability invalid	X	X	X	
08	Argument/parameter				
	01 parameter reference violation	X	X	X	
0A	Authorization				
	01 unauthorized for operation		X		
10	Damage encountered				
	04 system object damage state	X	X	X	X
	05 authority verification terminated due to damaged object				X
	44 partial system object damage	X	X	X	X
1A	Lock state				
	01 invalid lock state		X		
1C	Machine-dependent exception				
	03 machine storage limit exceeded				X
20	Machine support				
	02 machine check				X
	03 function check				X
22	Object access				
	01 object not found	X	X	X	
	02 object destroyed	X	X	X	
	03 object suspended	X	X	X	
	07 authority verification terminated due to destroyed object				X
	08 object compressed				X
24	Pointer specification				
	01 pointer does not exist	X	X	X	
	02 pointer type invalid	X	X	X	
2E	Resource control limit				
	01 user profile storage limit exceeded				X
32	Scalar specification				

Exception		Operands 1	2	3	Other
	03 scalar value invalid			X	
36	Space management				
	01 space extension/truncation				X
38	Template specification				
	03 materialization length exception	X			

Materialize User Profile (MATUP)

Op Code (Hex)	Operand 1	Operand 2
013E	Receiver	User profile

Operand 1: Space pointer.

Operand 2: System pointer.

ILE access

```
MATUP (
      receiver     : space pointer;
  var user_profile : system pointer
)
```

Description: The attributes of the user profile specified by operand 2 are materialized into the receiver specified by operand 1.

The receiver identified by operand 1 must be 16-byte aligned in the space. The following is the format of the materialized information:

- Materialization size specification — Char(8)
  - Number of bytes provided for materialization — Bin(4)
  - Number of bytes available for materialization — Bin(4)
- Object identification — Char(32)
  - Object type — Char(1)
  - Object subtype — Char(1)
  - Object name — Char(30)
- Object creation options — Char(4)
  - Existence attribute — Bit 0

    1 = Permanent
  - Space attribute — Bit 1

    0 = Fixed-length
    1 = Variable-length
  - Reserved (binary 1) — Bit 2
  - Reserved (binary 0) — Bits 3-12
  - Initialize space — Bit 13
  - Reserved (binary 0) — Bits 14-31
- Reserved (binary 0) — Char(4)
- Size of space — Bin(4)
- Initial value of space — Char(1)
- Performance class — Char(4)
- Reserved (binary 0) — Char(7)
- Reserved (binary 0) — Char(16)

• Reserved (binary 0)	Char(16)
• Privileged instructions (1 = authorized)	Char(4)
– Create Logical Unit Description	Bit 0
– Create Network Description	Bit 1
– Create Controller Description	Bit 2
– Create User Profile	Bit 3
– Modify User Profile	Bit 4
– Diagnose	Bit 5
– Terminate Machine Processing	Bit 6
– Initiate Process	Bit 7
– Modify Resource Management Controls	Bit 8
– Create Mode Description	Bit 9
– Create Class of Service Description	Bit 10
– Reserved (binary 0)	Bits 11-31
• Special authorizations (1 = authorized)	Char(4)
– All object authority	Bit 0
– Load (unrestricted)	Bit 1
– Dump (unrestricted)	Bit 2
– Suspend object (unrestricted)	Bit 3
– Load (restricted)	Bit 4
– Dump (restricted)	Bit 5
– Suspend object (restricted)	Bit 6
– Process control	Bit 7
– Reserved (binary 0)	Bit 8
– Service authority	Bit 9
– Auditor authority	Bit 10
– Spool control	Bit 11
– Reserved (binary 0)	Bit 12-23
– Modify machine attributes	Bits 24-31
- Group 2	Bit 24
- Group 3	Bit 25
- Group 4	Bit 26
- Group 5	Bit 27
- Group 6	Bit 28
- Group 7	Bit 29
- Group 8	Bit 30
- Group 9	Bit 31

Note: Group 1 requires no authorization.

- Storage authorization — Bin(4)
  The maximum amount of auxiliary storage (in units of 1024 bytes) that can be allocated for the storage of objects owned by this user profile
- Storage utilization — Bin(4)
  The current amount of auxiliary storage (in units of 1024 bytes) allocated for the storage of objects owned by this user profile
- User profile status — Char(2)
  - Verify storage utilization — Bit 0 1=Storage utilization has not been verified and may not be correct. The Materialize Authorized Objects instruction can be used to verify the storage utilization.
  - Reserved (binary 0) — Bits 1-15
- Reserved (binary 0) — Char(1)
- Object audit level — Char(1)
  - Reserved (binary 0) — Bits 0-5
  - Audit object changes for this user — Bit 6
  - Audit object reads for this user — Bit 7
- User Audit level 1 — Char(4)
- User Audit level 2 — Char(4)
  - Audit program adoption (1=audit) — Bit 0
  - Reserved (binary 0) — Bits 1-31
- Reserved (binary 0) — Char(4)

The first 4 bytes of the materialization identify the total **number of bytes provided** for use by the instruction. This value is supplied as input to the instruction and is not modified by the instruction. A value of less than 8 causes the *materialization length* (hex 3803) exception to be signaled.

The second 4 bytes of the materialization identify the total **number of bytes available** to be materialized. The instruction materializes as many bytes as can be contained in the area specified as the receiver. If the byte area identified by the receiver is greater than that required to contain the information requested, then the excess bytes are unchanged. No exceptions (other than the *materialization length* (hex 3803) exception) are signaled in the event that the receiver contains insufficient area for the materialization.

The attributes that the instruction can materialize are described in the Create User Profile instruction.

Authorization Required:

- Operational
  - Operand 2

Lock Enforcement

- Materialize
  - Operand 2

Exceptions

Exception		Operands 1	2	Other
06	Addressing			
	01 space addressing violation	X	X	
	02 boundary alignment	X	X	
	03 range	X	X	
	06 optimized addressability invalid	X	X	
08	Argument/parameter			
	01 parameter reference violation	X	X	
0A	Authorization			
	01 unauthorized for operation		X	
10	Damage encountered			
	02 machine context			X
	04 system object damage state	X	X	X
	05 authority verification terminated due to damaged object			X
	44 partial system object damage	X	X	X
1A	Lock state			
	01 invalid lock state		X	
1C	Machine-dependent exception			X
	03 machine storage limit exceeded			X
20	Machine support			
	02 machine check			X
	03 function check			X
22	Object access			
	01 object not found	X	X	
	02 object destroyed	X	X	
	03 object suspended	X	X	
	07 authority verification terminated due to destroyed object			X
	08 object compressed			X
24	Pointer specification			
	01 pointer does not exist	X	X	
	02 pointer type invalid	X	X	
	03 pointer addressing invalid object		X	
2E	Resource control limit			
	01 user profile storage limit exceeded			X
36	Space management			
	01 space extension/truncation			X
38	Template specification			
	03 materialization length exception	X		

Test Authority (TESTAU)

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand 3	Operand [4-5]
TESTAU 10F7		Available authority template receiver	System object or object template	Required authority template	
TESTAUB 1CF7	Branch options	Available authority template receiver	System object or object template	Required authority template	Branch targets
TESTAUI 18F7	Indicator options	Available authority template receiver	System object or object template	Required authority template	Indicator targets

Operand 1: Character(2) variable scalar (fixed length) or null.

Operand 2: System pointer or space pointer data object.

Operand 3: Character(2) scalar (fixed length).

Operand 4-5:

- *Branch Form*-Branch point, instruction pointer, relative instruction number, or absolute instruction number.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

ILE access

```
TESTAU (
 var receiver                   : aggregate; OR
                                  null operand;
 var system_object_or_template : pointer;
 var required_authority         : aggregate;
) : signed binary /* return_code */
```

The return code will be set as follows:

Return_Code	Meaning
1	Authorized.
0	Not Authorized.

Description: This instruction verifies that the object authorities and/or ownership rights specified by operand 3 are currently available to the process for the object specified by operand 2.

If operand 1 is not null, all of the authorities and/or ownership specified by operand 3 that are currently available to the process are returned in operand 1.

If an object template is not specified (i.e. operand 2 is a system pointer), then authority verification is performed relative to the invocation executing this instruction. If an object template is specified (i.e. operand 2 is a space pointer), then authority verification is performed relative to the invocation specified in the template. Specifying an invocation causes the invocations subsequent to it to be bypassed in the authority verification process. This has the influence of excluding the program adopted user profiles for any of these excluded invocations from acting as a source of authority to the authority verification process.

The required authorities and/or ownership are specified by the required authority template of operand 3. This template includes a test option that indicates whether all of the specified authorities are required or whether any one or more of the specified authorities is sufficient. This option can be used, for example, to test for operational authority by coding a template value of hex 0F01 in operand 3. Using the *any* option does not affect what is returned in operand 1. If operand 1 is not null and the *any* option is specified, all of the authorities specified by operand 3 that are available to the process are returned in operand 1.

If the required authority is available, one of the following occurs:

- Branch form indicated
  - Conditional transfer of control to the instruction indicated by the appropriate branch target operand.
- Indicator form specified
  - The leftmost byte of each of the indicator operands is assigned the following values.

    Hex F1- If the result of the test matches the corresponding indicator option
    Hex F0- If the result of the test does not match the corresponding indicator option

If no branch options are specified, instruction execution proceeds to the next instruction. If operand 1 is null and neither the branch or indicator form is used, an invalid operand type exception is signaled.

The format for the available authority template (operand 1) is as follows: (1 = authorized)

• Authorization template	Char(2)
— Object control	Bit 0
— Object management	Bit 1
— Authorized pointer	Bit 2
— Space authority	Bit 3
— Retrieve	Bit 4
— Insert	Bit 5
— Delete	Bit 6
— Update	Bit 7
— Ownership (1 = yes)	Bit 8
— Excluded	Bit 9
— Authority List management	Bit 10
— Reserved (binary 0)	Bits 11-15

If operand 2 is a system pointer, it identifies the object for which authority is to be tested. If operand 2 is a space pointer, it provides addressability to the object template. The format for the optional object template is as follows:

• Object template	Char(32)
— Relative invocation	Bin(2)
— Reserved (binary 0)	Char(14)
— System object	System pointer

The **relative invocation** field in the object template identifies an invocation relative to the current invocation at which the authority verification is to be performed. The value of the relative invocation field must be less than or equal to zero. A value of zero identifies the current invocation, -1 identifies the prior invocation, -2, the invocation prior to that, and so on. A value larger than the number of invoca-

tions currently on the invocation stack or a positive value results in the signaling of the *template value invalid* (hex 3801) exception. The program adopted and propagated user profiles for the identified invocation and older invocations will be included in the authority verification process. Program adopted user profiles for invocations newer than the identified invocation will not be included in the authority verification process. If the current invocation is specified, its program adopted user profile is included whether or not it is to be propagated.

The **system object** field specifies a system pointer which identifies the object for which authority is to be tested.

The format for the required authority template (operand 3) is as follows: (1 = authorized)

• Authorization template	Char(2)
– Object control	Bit 0
– Object management	Bit 1
– Authorized pointer	Bit 2
– Space authority	Bit 3
– Retrieve	Bit 4
– Insert	Bit 5
– Delete	Bit 6
– Update	Bit 7
– Ownership (1 = yes)	Bit 8
– Excluded	Bit 9
– Authority List management	Bit 10
– Reserved (binary 0)	Bits 11-14
– Test option	Bit 15

0 = All of the above authorities must be present.
1 = Any one or more of the above authorities must be present.

This instruction will tolerate a damaged object referenced by operand 2 when the reference is a resolved pointer. The instruction will not tolerate damaged contexts or programs when resolving pointers. Damaged user profiles encountered during the authority verification processing result in the signaling of the authority verification terminated due to damaged object exception.

Resultant Conditions

- Authorized - the required authority is available.
- Unauthorized - the required authority is not available.

Authorization Required:

- Retrieve
  - Contexts referenced for address resolution

Lock Enforcement

- Materialize
  - Contexts referenced for address resolution

Exceptions

Exception		Operands 1	2	3	Other
06	Addressing				
	01 space addressing violation	X	X	X	
	02 boundary alignment	X	X	X	
	03 range	X	X	X	
	06 optimized addressability invalid	X	X	X	
08	Argument/parameter				
	01 parameter reference violation	X	X	X	
0A	Authorization				
	01 unauthorized for operation		X		
10	Damage encountered				
	02 machine context damage state		X		
	04 system object damage state	X	X	X	X
	05 authority verification terminated due to damaged object				X
	44 partial system object damage	X	X	X	X
1A	Lock state				
	01 invalid lock state		X		
1C	Machine-dependent exception				
	03 machine storage limit exceeded				X
20	Machine support				
	02 machine check				X
	03 function check				X
22	Object access				
	01 object not found	X	X	X	
	02 object destroyed	X	X	X	
	07 authority verification terminated due to destroyed object				X
	08 object compressed				X
24	Pointer specification				
	01 pointer does not exist	X	X	X	
	02 pointer type invalid	X	X	X	
2C	Program execution				
	04 invalid branch target				X
2E	Resource control limit				
	01 user profile storage limit exceeded				X
32	Scalar specification				
	01 scalar type invalid	X	X	X	
	03 scalar value invalid			X	
36	Space management				
	01 space extension/truncation				X

Exception		Operands 1	2	3	Other
38	Template specification				
	01 template value invalid		X		

Test Extended Authorities (TESTEAU)

Op Code (Hex)	Extender	Operand 1	Operand 2	Operand 3	Operand [4-5]
TESTEAU 10FB		Available authority template receiver	Required authority template	Relative invocation	
TESTEAUB 1CFB	Branch options	Available authority template receiver	Required authority template	Relative invocation	Branch targets
TESTEAUI 18FB	Indicator options	Available authority template receiver	Required authority template	Relative invocation	Indicator targets

Operand 1: Character(8) variable scalar (fixed length) or null

Operand 2: Character(8) scalar (fixed length)

Operand 3: Binary(2) variable scalar (fixed length) or constant, or null

Operand 4-5:

- *Branch Form*-Branch point, instruction pointer, relative instruction number, or absolute instruction number.
- *Indicator Form*-Numeric variable scalar or character variable scalar.

ILE access

```
TESTEAU (
  var receiver            : aggregate; OR
                            null operand;
  var required_authority  : aggregate;
  var relative_invocation : signed binary; OR
                            null operand;
) : signed binary /* return_code */
```

The return code will be set as follws:

Return_Code	Meaning
1	Authorized.
0	Not Authorized.

Description: This instruction verifies that the privileged instructions and special authorities specified by operand 2 are currently available to the process.

If operand 1 is not null, all of the privileged instructions and special authorities specified by operand 2 that are currently available to the process are returned in operand 1.

Note: The term *authority verification* refers to the testing of the required privileged instruction and special authorities.

If operand 3 is null, the authority verification is performed relative to the invocation executing this instruction. If an operand 3 is specified, the authority verification is performed relative to the invocation specified. Specifying an invocation causes the invocations subsequent to it to be bypassed in the authority verification process. This has the influence of excluding the program adopted user profiles

for any of these excluded invocations from acting as a source of authority to the authority verification process.

The required privileged instructions and special authorities are specified by the required authority template of operand 2.

If the required authority is available, one of the following occurs:

- Branch form indicated
  - Conditional transfer of control to the instruction indicated by the appropriate branch target operand.
- Indicator form specified
  - The leftmost byte of each of the indicator operands is assigned the following values.

    Hex F1- If the result of the test matches the corresponding indicator option
    Hex F0- If the result of the test does not match the corresponding indicator option

If no branch options are specified, instruction execution proceeds to the next instruction. If operand 1 is null and neither the branch or indicator form is used, an invalid operand type exception is signaled.

The format for the available authority template (operand 1) is as follows: (1 = authorized)

• Authority template	Char(8)
— Privileged instruction template	Char(4)
- Create Logical Unit Description	Bit 0
- Create Network Description	Bit 1
- Create Controller Description	Bit 2
- Create User Profile	Bit 3
- Modify User Profile	Bit 4
- Diagnose	Bit 5
- Terminate Machine Processing	Bit 6
- Initiate Process	Bit 7
- Modify Resource Management Control	Bit 8
- Create Mode Description	Bit 9
- Create Class of Service Description	Bit 10
- Reserved (binary 0)	Bits 11-31
— Special authority template	Char(4)
- All object	Bit 0
- Load (unrestricted)	Bit 1
- Dump (unrestricted)	Bit 2
- Suspend (unrestricted)	Bit 3
- Load (restricted)	Bit 4
- Dump (restricted)	Bit 5
- Suspend (restricted)	Bit 6
- Process control	Bit 7
- Reserved (binary 0)	Bit 8

- Service	Bit 9
- Auditor authority	Bit 10
- Spool control	Bit 11
- Reserved (binary 0)	Bit 12-23
- Modify Machine Attributes	Bit 24-31
• Group 2	Bit 24
• Group 3	Bit 25
• Group 4	Bit 26
• Group 5	Bit 27
• Group 6	Bit 28
• Group 7	Bit 29
• Group 8	Bit 30
• Group 9	Bit 31

The format for the required authority template (operand 2) is as follows: (1 = authorized)

• Required authority	Char(8)
— Privileged instruction template	Char(4)
- Create Logical Unit Description	Bit 0
- Create Network Description	Bit 1
- Create Controller Description	Bit 2
- Create User Profile	Bit 3
- Modify User Profile	Bit 4
- Diagnose	Bit 5
- Terminate Machine Processing	Bit 6
- Initiate Process	Bit 7
- Modify Resource Management Control	Bit 8
- Create Mode Description	Bit 9
- Create Class of Service Description	Bit 10
- Reserved (binary 0)	Bits 11-31
— Special authority template	Char(4)
- All object	Bit 0
- Load (unrestricted)	Bit 1
- Dump (unrestricted)	Bit 2
- Suspend (unrestricted)	Bit 3
- Load (restricted)	Bit 4
- Dump (restricted)	Bit 5
- Suspend (restricted)	Bit 6
- Process control	Bit 7
- Reserved (binary 0)	Bit 8
- Service	Bit 9
- Auditor authority	Bit 10

- Spool control	Bit 11
- Reserved (binary 0)	Bit 12-23
- Modify Machine Attributes	Bit 24-31
• Group 2	Bit 24
• Group 3	Bit 25
• Group 4	Bit 26
• Group 5	Bit 27
• Group 6	Bit 28
• Group 7	Bit 29
• Group 8	Bit 30
• Group 9	Bit 31

The **relative invocation** operand (operand 3) identifies an invocation relative to the current invocation at which the authority verification is to be performed. The value of the *relative invocation* field must be less than or equal to zero. A value of zero identifies the current invocation, -1 identifies the prior invocation, -2, the invocation prior to that, and so on. A value larger than the number of invocations currently on the invocation stack or a positive value results in the signaling of the *scalar value invalid* (hex 3203) exception.

An immediate value is not allowed for operand 3.

The program adopted and propagated user profiles for the identified invocation and older invocations will be included in the authority verification process. Program adopted user profiles for invocations newer than the identified invocation will not be included in the authority verification process. If the current invocation is specified, its program adopted user profile is included whether or not it is to be propagated.

Damaged user profiles encountered during the authority verification processing result in the signaling of the authority verification terminated due to damaged object exception.

Resultant Conditions

- Authorized - the required authority is available.
- Unauthorized - the required authority is not available.

Authorization Required:

- Retrieve
  - Contexts referenced for address resolution

Lock Enforcement

- Materialize
  - Contexts referenced for address resolution

Exceptions

Exception		Operands 1	2	3	Other
06	Addressing				
	01 space addressing violation	X	X	X	
	02 boundary alignment	X	X	X	
	03 range	X	X	X	

Exception		Operands 1	2	3	Other
	06 optimized addressability invalid	X	X	X	
08	Argument/parameter				
	01 parameter reference violation	X	X	X	
0A	Authorization				
	01 unauthorized for operation		X		
10	Damage encountered				
	02 machine context damage state		X		
	04 system object damage state	X	X	X	X
	05 authority verification terminated due to damaged object				X
	44 partial system object damage	X	X	X	X
1A	Lock state				
	01 invalid lock state		X		
1C	Machine-dependent exception				
	03 machine storage limit exceeded				X
20	Machine support				
	02 machine check				X
	03 function check				X
22	Object access				
	01 object not found	X	X	X	
	02 object destroyed	X	X	X	
	07 authority verification terminated due to destroyed object				X
	08 object compressed				X
24	Pointer specification				
	01 pointer does not exist	X	X	X	
	02 pointer type invalid	X	X	X	
2C	Program execution				
	04 invalid branch target				X
2E	Resource control limit				
	01 user profile storage limit exceeded				X
32	Scalar specification				
	01 scalar type invalid	X	X	X	
	03 scalar value invalid			X	
36	Space management				
	01 space extension/truncation				X
38	Template specification				
	01 template value invalid		X		

Chapter 18. Process Management Instructions

This chapter describes instructions used for process management. These instructions are in alphabetic order. For an alphabetic summary of all the instructions, See Appendix A, "Instruction Summary."

| Materialize Process Activation Groups (MATPRAGP) 18-3
Materialize Process Attributes (MATPRATR) 18-5
Wait On Time (WAITTIME) 18-18

© Copyright IBM Corp. 1991, 1993

Materialize Process Activation Groups (MATPRAGP)

Op Code (Hex)	Operand 1
0331	Receiver

Operand 1: Space pointer

ILE access

```
MATPRAGP (
      receiver : space pointer
)
```

Warning: The following information is subject to change from release to release. Use it with caution and be prepared to adjust for changes with each new release.

Description: This instruction provides a list of the activation groups which exist in the current process. Operand 1 locates a template which receives information.

The materialization template identified by operand 1 must be 16-byte aligned in the space. This materialization template has the following format:

• Number of bytes provided for materialization	Bin(4)
• Number of bytes available for materialization	Bin(4)
• Activation group count	Bin(4)
• Activation group list	
array(1..*activation group count*) of	UBin(4)

The Materialize Activation Group Attributes instruction can be used to examine the attributes of an individual activation group.

The first 4 bytes of the materialization template specify the **number of bytes provided** for use by the instruction. In all cases if the number of bytes provided is less than 8 then a *materialization length* (hex 3803) exception will be signaled.

The second 4 bytes of the instruction indicate the actual **number of bytes available** to be returned. In no case does the instruction return more bytes of information than those available.

activation group count
This is the number of activation groups within the process. It is also the extent of the activation group list which follows.

activation group list
This is the list of activation groups which exist within the current process.

This is an array of activation group mark values. Each entry denotes an activation group currently existent within the process.

Authorization: n/a

Lock Enforcement: n/a

Exceptions

Exception		Operands 1	Other
06	Addressing		
	01 space addressing violation	X	
	03 range	X	
	06 optimized addressability invalid	X	
08	Argument/parameter	X	
	01 parameter reference violation	X	
10	Damage encountered		
	44 partial system object damage	X	X
1C	Machine-dependent exception		
	03 machine storage limit exceeded		X
	04 object storage limit exceeded		X
20	Machine support		
	02 machine check		X
	03 function check		X
38	Template specification		
	03 Materialization length	X	
44	Domain specification		
	01 Domain Violation	X	

Materialize Process Attributes (MATPRATR)

Op Code (Hex)	Operand 1	Operand 2	Operand 3
0333	Receiver	Process control space	Materialization options

Operand 1: Space pointer.

Operand 2: System pointer or null.

Operand 3: Character scalar(1).

ILE access

```
MATPRATR (
     receiver                 : space pointer;
 var process_control_space    : system pointer; OR
                                null operand;
 var materialization_options : aggregate
)
```

Description: The instruction causes either one specific attribute or all the attributes of the designated process to be materialized.

Operand 1 specifies a space that is to receive the materialized attribute values. The space pointer specified in operand 1 must address a 16-byte aligned area.

Operand 2 is a system pointer identifying the process control space associated with the process whose attributes are to be materialized. If operand 2 is null, the process issuing the instruction is the subject process. If the subject process's attributes are being materialized by another process, that process must be the original initiator of the subject process or the governing user profile(s) must have process control special authorization.

Operand 3 is a character scalar(1) specifying which process attribute is to be materialized. A value of hex 00 results in all the attributes of a process being materialized in the format described in the Initiate Process instruction for the process definition template. Other options allow materialization of specialized process attributes.

A summary of the allowable hex values for operand 3 follows.

00	Entire orginal PDT
01-0B	Entire Char(4) *process control attributes* field
0C	Signal event control mask
0D	Number of event monitors
0E	Priority
0F	Main storage pool id
10	Maximum temporary auxiliary storage allowed
11	Time slice interval
12	Default time-out interval
13	Maximum processor time allowed
14	Multiprogramming level class ID
15	Modification control indicators
16	User profile pointer
17	Process communication object (PCO) pointer
18	Process name resolution list space pointer

19	Initiation phase program pointer
1A	Termination phase program pointer
1B	Problem phase program pointer
1C	Process default exception handler (PDEH) program pointer
1F	Process access group pointer
20	Process status indicators
21	Process resource usage attributes
22	Subordinate processes identification attributes
23	Performance status attributes
24	Execution status attributes
25	Process control space system pointer
26	Adopted user profile list space pointer
27	Entire Char(4) *process control attributes* field
28	Process category
29	Queue space system pointer

Note that there exist differences in the operand 3 value definitions between this instruction and the Modify Process Attributes instruction.

The materialization template has the following general format when a process scalar attribute is materialized:

- Materialization size specification — Char(8)
  - Number of bytes provided for materialization — Bin(4)
  - Number of bytes available for materialization — Bin(4)
- Process scalar attributes — Char(*)

The materialization template has the following general format when a process pointer attribute is materialized:

- Materialization size specification — Char(8)
  - Number of bytes provided for materialization — Bin(4)
  - Number of bytes available for materialization — Bin(4)
- Reserved (binary 0) — Char(8)
- Process pointer attribute — System pointer or Space pointer

Note: The values of the entry associated with an asterisk (*) are ignored by this instruction.

The following attributes require materialization targets of varying lengths. The attributes to be materialized and their operand 3 materialization option values follow.

- Process control attributes — Char(4)

  Values hex 01 through hex 0B or hex 27 cause the 4-byte process control attributes value to be placed in the byte area identified by operand 1. The individual attributes and the corresponding values are as follows:

  - Process type — Bit 0

    0 = Dependent process
    1 = Independent process

  - Instruction wait access state control — Bit 1

    0 = Access state modification is not allowed
    1 = Access state modification is allowed if specified

- Time slice end access state control — Bit 2

  0 = Access state modification is not allowed
  1 = Access state modification is allowed if specified

- Time slice end event option — Bit 3

  0 = Time slice expired without entering instruction wait event is not signaled
  1 = Time slice expired without entering instruction wait event is signaled

- Reserved (binary 0) — Bit 4

- Initiation phase program option — Bit 5

  0 = No initiation phase program specified (do not enter initiation phase)
  1 = Initiation phase program specified (enter initiation phase)

- Problem phase program option — Bit 6

  0 = No problem phase program specified (do not enter problem phase)
  1 = Problem phase program specified (enter problem phase)

- Termination phase program option — Bit 7

  0 = No termination phase program specified (do not enter termination phase)
  1 = Termination phase program specified (enter termination phase)

- Process default exception handler option — Bit 8

  0 = No process default exception handler
  1 = Process default exception handler specified

- Process NRL (name resolution list) option — Bit 9

  0 = No process NRL specified
  1 = Process NRL specified

- Process access group option — Bit 10

  0 = No process access group specified
  1 = Process access group specified

- Process adopted user profile list option — Bit 11

  0 = No process adopted user profile list specified
  1 = Process adopted user profile list specified

- Process category specified — Bit 12

  0 = No process category specified when the process was initiated
  1 = A process category was specified when the process was initiated

- Recycling control for process storage addresses used by user state programs — Bit 13

  0 = Process storage addresses used by user state programs are not recycled within the process
  1 = Process storage addresses used by user state programs are recycled within the process

- Implicitly created activation group's automatic storage access group membership control — Bit 14

  0 = The machine is free to create automatic storage areas of implicitly created activation groups within the process access group

  Note: This in no way guarantees that the automatic areas will in fact be access group members. This is an advisory flag informing the machine that the MI user would prefer these areas to be access group members.

1 = Implicitly created activation group's automatic storage areas will not be created within the process access group

- Implicitly created activation group's static storage access group membership control — Bit 15

  0 = The machine is free to create static storage areas of implicitly created activation groups within the process access group

  Note: This in no way guarantees that the static areas will in fact be access group members. This is an advisory flag informing the machine that the MI user would prefer these areas to be access group members.

  1 = Implicitly created activation group's static storage areas will not be created within the process access group

- Implicitly created activation group's default heap storage access group membership control — Bit 16

  0 = The machine is free to create default heap storage areas of implicitly created activation groups within the process access group

  Note: This in no way guarantees that the activation group default heap areas will in fact be access group members. This is an advisory flag informing the machine that the MI user would prefer these areas to be access group members.

  1 = Implicitly created activation group's default heap storage areas will not be created within the process access group

- Reserved (binary 0) — Bits 17-31

• Signal event control mask

The materialization of the control mask is as follows:

- Hex 0C = Signal event control mask — Char(2)

• Number of event monitors

The materialization of this attribute is as follows:

- Hex 0D = Number of event monitors — Bin(2)

The resource management attributes and data types are as follows:

Attribute	Type
• Hex 0E = Process priority	Char(1)
• Hex 0F = Process storage pool ID	Char(1)
• Hex 10 = Maximum temporary auxiliary storage allowed	Bin(4)
• Hex 11 = Time slice interval	Char(8)
• Hex 12 = Default time-out interval	Char(8)
• Hex 13 = Maximum processor time allowed	Char(8)
• Hex 14 = Process multiprogramming level class ID	Char(1)
• Hex 15 = Modification control indicators	Char(8)

The *modification control indicators* are materialized when the operand 3 value is hex 15. Each indicator specifies the modification options allowed to a process upon itself by the initiating process. The possible values of each *modification control indicator* are as follows:

00 = Modification of the attribute is not allowed.

01 = Modification is allowed in the initiation or termination phases only.

11 = Modification is allowed in all phases (initiation, problem, and termination).

The bit assignments of the *modification control indicators* are as follows:

• Instruction wait access state control	Bits 0-1
• Time slice end access state control	Bits 2-3
• Time slice event option	Bits 4-5
• Reserved (binary 0)	Bits 6-7
• Problem phase program option	Bits 8-9
• Termination phase program option	Bits 10-11
• Process default exception handler option	Bits 12-13
• Process NRL option	Bits 14-15
• Signal event control mask	Bits 16-17
• Process priority	Bits 18-19
• Process storage pool identification	Bits 20-21
• Maximum temporary auxiliary storage allowed	Bits 22-23
• Time slice interval	Bits 24-25
• Default time-out interval	Bits 26-27
• Maximum processor time allowed	Bits 28-29
• Process MPL class ID	Bits 30-31
• User profile pointer	Bits 32-33
• Reserved	Bits 34-35
• Process NRL pointer	Bits 36-37
• Termination phase program pointer	Bits 38-39
• Problem phase program pointer	Bits 40-41
• Process default exception handler	Bits 42-43
• Process adopted user profile list	Bits 44-45
• Process adopted user profile list option	Bits 46-47
• Process category	Bits 48-49
• Recycling control for process storage addresses used by user state programs	Bits 50-51
• Reserved (binary 0)	Bits 52-63

The process pointer attributes which may be materialized are the following:

- Hex 16 = Process user profile pointer

  The system pointer with addressability to the user profile is placed into the space addressed by operand 1. If the materialization option (hex 00) is specified in operand 3, a reserved Char(9) field is included at this point. This user profile is the process user profile assigned by the Initiate Process or Modify Process Attribute instruction.

- Hex 17 = Process communication object (PCO) pointer

  The PCO pointer is placed in the space addressed by operand 1.

- Hex 18 = Process name resolution list

  The space pointer to the NRL is placed in the space addressed by operand 1.

- Hex 19 = Initiation phase program pointer

  The system pointer to the program is placed in the space addressed by operand 1.

- Hex 1A = Termination phase program pointer

  The system pointer to the program is placed in the space addressed by operand 1.

- Hex 1B = Problem phase program pointer

  The system pointer to the program is placed in the space addressed by operand 1.

- Hex 1C = PDEH (process default exception handler program)

  The system pointer to the program is placed in the space addressed by operand 1.

- Hex 1F = Process access group

  The system pointer with addressability to the PAG is placed in the space addressed by operand 1.

- Hex 20 = Process status indicators

  Process status indicators are materialized when the value of operand 3 is hex 20. The format and associated values of this attribute are as follows:

Field	Type
— Process states	Char(2)
- External existence state	Bits 0-2
000 = Suspended 010 = Suspended, in instruction wait 100 = Active, in ineligible wait 101 = Active, in current MPL 110 = Active, in instruction wait	
- Invocation exit active	Bit 3
- Reserved (binary 0)	Bits 4-7
- Internal processing phase	Bits 8-10
001 = Initiation phase 010 = Problem phase 100 = Termination phase	
- Reserved (binary 0)	Bits 11-15
— Process interrupt status (1 = pending)	Char(2)
- Time slice end pending	Bit 0
- Transfer lock pending	Bit 1
- Asynchronous lock retry pending	Bit 2
- Suspend process pending	Bit 3
- Resume process pending	Bit 4
- Resource management attribute modify pending	Bit 5
- Process attribute modify pending	Bit 6
- Terminate machine processing pending	Bit 7
- Terminate process pending	Bit 8
- Wait time-out pending	Bit 9
- Event schedule pending	Bit 10
- Machine service pending	Bit 11
- Cancel Long Running MI Instruction	Bit 11

- Reserved (binary 0) — Bits 13-15

— Process initial internal termination status — Char(3)

- Initial internal termination reason — Bits 0-7

  Hex 80 = Return from first invocation in problem phase.
  Hex 40 = Return from first invocation in initiation phase, and no problem phase program specified.
  Hex 20 = Terminate Process instruction issued by this process to itself.
  Hex 10 = Exception was not handled by the process.
  Hex 00 = Process terminated externally.

- Initial internal termination code — Bits 8-23

  The code is assigned in one of the following ways:

  1. If the termination is caused by a Return External instruction from the first invocation, then this code is binary 0's.
  2. The code is assigned by operand 2 of the Terminate Process instruction. This code is also given to subordinate processes involved in the termination.
  3. The code is assigned by the original exception code that caused process termination to commence. This code is also given to subordinate processes involved in the termination.

— Process initial external termination status — Char(3)

- Initial external termination reason: — Bits 0-7

  Hex 80 = Terminate Process instruction issued explicitly to this process from another process.
  Hex 40 = A superordinate process has been terminated.
  Hex 00 = Process terminated internally.

- Initial external termination code: — Bits 8-23

  This code is supplied by the termination code in operand 2 of the Terminate Process instruction.

— Process final termination status — Char(3)

- Final termination reason: — Bits 0-7

  Hex 80 = Return instruction from first invocation.
  Hex 40 = Terminate Process instruction issued by the process being materialized.
  Hex 20 = Terminate Process instruction issued to the process being materialized by another process.
  Hex 10 = Exception not handled by this process.
  Hex 08 = Terminate Process instruction issued to superordinate of the process being materialized.
  Hex 04 = Superordinate process of the process being materialized completed termination phase.

- Final termination code is assigned in one of the following ways: — Bits 8-23

  1. If the termination is caused by a Return External instruction from first invocation, then this code is binary 0's.
  2. The termination code is assigned by the Terminate Process instruction.
  3. The termination code is assigned by the original exception code that caused process termination.

The process final termination status is presented as event-related data in the terminate process event. Usually the event is the only source of the process final termination status since the process will cease to exist before its attributes can be materialized.

- Hex 21 = Process resource usage attributes

  Process resource usage attributes are materialized when the value of operand 3 is hex 21. The format and associated values of this attribute are as follows:

  - Total temporary auxiliary storage used — Bin(4)
  - Total processor time used — Char(8)
  - Number of locks currently held by the process (including implicit locks) — Bin(2)

- Hex 22 Subordinate processes identification attributes *Subordinate processes identification attributes* are materialized when the value of operand 3 is hex 22. The format and associated values of this attribute are as follows:

  - Materialization size specification — Char(8)
    - Number of bytes provided for matialization — Bin(4)
    - Number of bytes available for materialization — Bin(4)
  - Number of immediately subordinate processes — Bin(2)
  - Reserved (binary 0) — Char(6)
  - System pointer to the process control space — System pointer(s) for each subordinate process (repeated for each immediately subordinate process)

- Hex 23 = Process performance attributes *Process performance attributes* are materialized when the value of operand 3 is hex 23. The format and associated values of this attribute are as follows:

  - Materialization size specification — Char(8)
    - Number of bytes provided for materialization — Bin(4)
    - Number of bytes available for materialization — Bin(4)
  - Number of synchronous page reads into main storage associated with data base — Bin(4)
  - Number of synchronous page reads into main storage not associated with data base — Bin(4)
  - Total number of synchronous page writes from main storage. This includes writes associated with and not associated with data base. — Bin(4)
  - Number of transitions into ineligible wait state — UBin(2)
  - Number of transitions into an instruction wait — UBin(2)
  - Number of transitions into ineligible wait state from an instruction wait — UBin(2)
  - Timestamp of materialization — Char(8)
  - Number of asynchronous reads into main storage associated with data base — Bin(4)
  - Number of asynchronous reads into main storage not associated with data base — Bin(4)
  - Number of synchronous writes from main — Bin(4)

storage associated with data base

- Number of synchronous writes from main storage not associated with data base — Bin(4)
- Number of asynchronous writes from main storage associated with data base — Bin(4)
- Number of asynchronous writes from main storage not associated with data base — Bin(4)
- Total number of writes from main storage of permanent objects — Bin(4)
- Total reads and writes performed for checksum updating due to writes of checksum protected objects — Bin(4)
- Number of page faults on process access group objects — Bin(4)
- Number of internal effective address overflow exceptions — Bin(4)
- Number of internal binary overflow exceptions — Bin(4)
- Number of internal decimal overflow exceptions — Bin(4)
- Number of internal floating point overflow exceptions — .Bin(4)
- Number of times a page fault occurred on an address that was currently part of an auxiliary storage I/O operation — Bin(4)
- Number of times the process explicitly waited for outstanding asynchronous I/O operations to complete — Bin(4)

Each of the Bin(2) counters has a limit of 32 767. If this limit is exceeded, the count is set to 0, and no exception is signaled.

The process performance attributes are not supplied with materialization option hex 00.

The first 4 bytes of the materialization identify the total **number of bytes provided** for use by the instruction. This value is supplied as input to the instruction and is not modified by the instruction. A value of less than 8 causes the *materialization length* (hex 3803) exception to be signaled.

The second 4 bytes of the materialization identify the total **number of bytes available** to be materialized. The instruction materializes as many bytes as can be contained in the area specified as the receiver. If the byte area identified by the receiver is greater than that required to contain the information requested, then the excess bytes are unchanged. No exceptions (other than the *materialization length* (hex 3803) exception described previously) are signaled in the event that the receiver contains insufficient area for the materialization.

- Hex 24 = Process execution status attributes

  Process execution status attributes are materialized when the value of operand 3 is hex 24. The format and associated values of this attribute are as follows:

  - Process priority — Char(2)
    - Machine interface priority — Char(1)
    - Machine adjusted priority — Char(1)

      Normal value is hex 80. This value is dynamically modified by the machine.
  - Pending interruptions — Char(2)

- Time slice end	Bit 0
- Transfer lock	Bit 1
- Asynchronous lock retry	Bit 2
- Suspend process	Bit 3
- Resume process	Bit 4
- Modify resource management attribute	Bit 5
- Modify process attribute	Bit 6
- Terminate machine processing	Bit 7
- Terminate process	Bit 8
- Wait time-out	Bit 9
- Event	Bit 10
- Machine service pending	Bit 11
- Cancel Long Running MI Instruction	Bit 11
- Reserved (binary 0)	Bits 13-15
— Execution status	Char(2)
- Suspended	Bit 0
- Instruction wait	Bit 1
- In MPL	Bit 2
- Ineligible wait	Bit 3
- Reserved (binary 0)	Bits 4-15
— Wait status	Char(2)
- Wait on event	Bit 0
- Dequeue	Bit 1
- Lock	Bit 2
- Wait on time	Bit 3
- Wait to start a commit cycle	Bit 4
- Reserved (binary 0)	Bits 5-15
— Process class identification	Char(2)
- Storage pool class	Char(1)
- MPL class	Char(1)
— Processor time used	Char(8)
— Performance attributes	Char(78)
- Number of synchronous reads into main storage associated with data base	Bin(4)
- Number of synchronous reads into main storage not associated with data base	Bin(4)
- Total number of synchronous page writes from main storage. This includes writes associated with and not associated with data base.	Bin(4)
- Transitions to ineligible wait	UBin(2)

- Transitions to instruction wait — UBin(2)
- Transitions to ineligible from instruction wait — UBin(2)
- Number of asynchronous reads into main storage associated with data base — Bin(4)
- Number of asynchronous reads into main storage not associated with data base — Bin(4)
- Number of synchronous writes from main storage associated with data base — Bin(4)
- Number of synchronous writes from main storage not associated with data base — Bin(4)
- Number of asynchronous writes from main storage associated with data base — Bin(4)
- Number of asynchronous writes from main storage not associated with data base — Bin(4)
- Total number of writes from main storage of permanent objects — Bin(4)
- Total reads and writes performed for checksum updating due to writes of checksum protected objects — Bin(4)
- Number of page faults on process access group objects — Bin(4)
- Number of internal effective address overflow exceptions — Bin(4)
- Number of internal binary overflow exceptions — Bin(4)
- Number of internal decimal overflow exceptions — Bin(4)
- Number of internal floating point overflow exceptions — Bin(4)
- Number of times a page fault occurred on an address that was currently part of an auxiliary storage I/O operation — Bin(4)
- Number of times the process explicitly waited for outstanding asynchronous I/O operations to complete — Bin(4)

- Hex 25 = Process control space pointer

  A system pointer to the *process control space* is materialized when the value of operand 3 is hex 25. If a process control space pointer is supplied in operand 2, it is ignored. A pointer to the process that is executing the MATPRATR instruction is always materialized.

- Hex 26 = Adopted user profile list attributes

  A materialization option's value of hex 26 causes the *adopted user profile list attributes* to be materialized as follows:

  - Materialization size specification — Char(8)
    - Number of bytes provided for materialization — Bin(4)
    - Number of bytes available for materialization — Bin(4)
  - Reserved (binary 0) — Char(8)
  - Pointer to the adopted user profile list — Space pointer

last used to set this attribute

- Number of user profiles in the encapsulated adopted user profile list — Bin(2)
- Reserved — Char(14)
- List of user profiles in the encapsulated adopted user profile list (one system pointer to each user profile in the list) — System pointers

Due to verifications performed on the user profiles specified in an adopted process user profile list input to either the Initiate Process or Modify Process instructions, the encapsulated adopted user profile list may differ from the input list. When verification of an input user profile fails, it is not included in the encapsulated list.

The adopted user profile list attributes are not supplied with materialization option hex 00.

- Hex 27 = Process control attributes

  A materialization option's value of hex 27 causes the process control attributes to be materialized. Refer to the description of this materialization provided in prior text for this instruction.

- Hex 28 = Process category — Char(2)
- Hex 29 = Queue Space system pointer

  The system pointer with addressability to the Queue Space is placed in the space addressed by operand 1.

Authorization Required

- Process control special authorization
  - For materializing a process other than the one executing this instruction
- Retrieve
  - Contexts referenced for address resolution

Lock Enforcement

- Materialize
  - Contexts referenced for address resolution

Exceptions

Exception		Operands 1	2	3	4	Other
06	Addressing					
	01 space addressing violation	X	X	X		
	02 boundary alignment	X	X	X		
	03 range	X	X	X		
	06 optimized addressability invalid	X	X	X		
08	Argument/parameter					
	01 parameter reference violation	X	X	X		
0A	Authorization					
	01 unauthorized for operation		X			
	04 unauthorized for process control		X			

Exception		Operands 1	2	3	4	Other
10	Damage encountered					
	04 system object damage state	X	X	X	X	X
	05 authority verification terminated due to damaged object					X
	44 partial system object damage	X	X	X	X	X
1A	Lock state					
	01 invalid lock state				X	
1C	Machine-dependent exception					
	03 machine storage limit exceeded					X
20	Machine support					
	02 machine check					X
	03 function check					X
22	Object access					
	01 object not found	X	X	X		
	02 object destroyed	X	X	X		
	03 object suspended	X	X	X		
	07 authority verification terminated due to destroyed object					X
	08 object compressed					X
24	Pointer specification					
	01 pointer does not exist	X	X	X		
	02 pointer type invalid	X	X	X		
	03 pointer addressing invalid object		X			
28	Process state					
	02 process control space not associated with a process		X			
2E	Resource control limit					
	01 user profile storage limit exceeded					X
32	Scalar specification					
	03 scalar value invalid		X			
36	Space management					
	01 space extension/truncation					X
38	Template specification					
	03 materialization length exception	X				

Wait On Time (WAITTIME)

Op Code (Hex)	Operand 1
0349	Wait template

Operand 1: Character(16) scalar (fixed-length)

ILE access

```
WAITTIME (
  var wait_template : aggregate
)
```

Description: This instruction causes the process to wait for a specified time interval. The current process is placed in wait state for the amount of time specified by the wait template in accordance with the specified wait options.

The format of the wait template for operand 1 is:

- Wait time interval — Char(8)
- Wait options — Char(2)
  - Access state control for entering wait — Bit 0
    - 0 = Do not modify access state
    - 1 = Modify access state
  - Access state control for leaving wait — Bit 1
    - 0 = Do not modify access state
    - 1 = Modify access state
  - MPL (multiprogramming level) control during wait — Bit 2
    - 0 = Do not remain in current MPL set
    - 1 = Remain in current MPL set
  - Reserved — Bits 3-15
- Reserved — Char(6)

The format of the **wait time interval** value is that of a 64-bit unsigned binary value where bit 41 is equal to 1024 microseconds, assuming the bits are numbered from 0 to 63.

The **access state control** options control whether the process access group (PAG) will be explicitly transferred between main and auxiliary storage when entering and leaving a wait as a result of execution of this instruction. Specification of *modify access state* requests that the PAG be purged from main to auxiliary storage for entering a wait and requests that the PAG be transferred from auxiliary to main storage for leaving a wait. Specification of *do not modify access state* requests that the PAG not be explicitly transferred between main and auxiliary storage as a result of executing this instruction.

The access state of the PAG is modified when entering the wait if the process' *instruction wait initiation access state* control attribute specifies *allow access state modification*, if the *access state control for entering wait* option specifies *modify access state*, and if the *MPL control during wait* option specifies *do not remain in current MPL set*.

The **MPL control during wait** option controls whether the process will be removed from the current MPL (multiprogramming level) set or remain in the current MPL set when the process enters a wait as a result of executing this instruction.

When the *MPL control during wait option* specifies *remain in current MPL set* and the *access state control for entering wait* option specifies *do not modify access state*, the process will remain in the current MPL set for a maximum of 2 seconds. After 2 seconds, the process will automatically be removed from the current MPL set. The automatic removal does not change or affect the total wait time specified for the process in the wait time interval.

Exceptions

Exception		Operands 1	2	3	4	Other
06	Addressing					
	01 space addressing violation	X				
	02 boundary alignment	X				
	03 range	X				
	06 optimized addressability invalid	X				
08	Argument/parameter					
	01 parameter reference violation	X				
10	Damage encountered					
	04 system object damage state					X
	05 authority verification terminated due to damaged object					X
	44 partial system object damage					X
20	Machine support					
	02 machine check					X
	03 function check					X
22	Object access					
	02 object destroyed	X				
	03 object suspended	X				
	07 authority verification terminated due to destroyed object					X
	08 object compressed					X
24	Pointer specification					
	01 pointer does not exist	X				
	02 pointer type invalid	X				
2E	Resource control limit					
	01 user profile storage limit exceeded					X
32	Scalar specification					
	01 scalar type invalid	X				
	02 scalar attributes invalid	X				
	03 scalar value invalid	X				
36	Space management					
	01 space extension/truncation					X

Chapter 19. Resource Management Instructions

This chapter describes the storage and resource management instructions. These instructions are in alphabetic order. For an alphabetic summary of all the instructions, see Appendix A, "Instruction Summary."

Ensure Object (ENSOBJ) 19-3
Materialize Access Group Attributes (MATAGAT) 19-5
Materialize Resource Management Data (MATRMD) 19-9
Set Access State (SETACST) 19-31

© Copyright IBM Corp. 1991, 1993

Ensure Object (ENSOBJ)

Op Code (Hex)	Operand 1
0381	System pointer

Operand 1: System pointer.

ILE access

```
ENSOBJ (
 var system_pointer : system pointer
)
```

Description: The object identified by operand 1 is protected from volatile storage loss. The machine ensures that any changes made to the specified object are recorded on nonvolatile storage media. The access state of the object is not changed by this instruction. If operand 1 addresses a temporary object, no operation is performed because temporary objects are not preserved during a machine failure. No exception is signaled if temporary objects are referenced.

Authorization Required

- Retrieve
  - Contexts referenced for address resolution

Lock Enforcement

- Materialize
  - Contexts referenced for address resolution

Exceptions

Exception		Operands 1	Other
06	Addressing		
	01 space addressing violation	X	
	02 boundary alignment	X	
	03 range	X	
	06 optimized addressability invalid	X	
08	Argument/parameter		
	01 parameter reference violation	X	
0A	Authorization		
	01 unauthorized for operation	X	
10	Damage encountered		
	04 system object damage state	X	
	05 authority verification terminated due to damaged object		X
	44 partial system object damage	X	X
1A	Lock state		

Exception		Operands 1	Other
	01 invalid lock state	X	
1C	Machine-dependent exception		
	03 machine storage limit exceeded		X
20	Machine support		
	02 machine check		X
	03 function check		X
22	Object access		
	01 object not found	X	
	02 object destroyed	X	
	03 object suspended	X	
	04 object not eligible for operation	X	
	07 authority verification terminated due to destroyed object		X
	08 object compressed		X
24	Pointer specification		
	01 pointer does not exist	X	
	02 pointer type invalid	X	
	03 pointer addressing invalid object	X	
2E	Resource control limit		
	01 user profile storage limit exceeded		X
30	Journal management		
	02 entry not journaled	X	
32	Scalar specification		
	01 scalar type invalid	X	
36	Space management		
	01 space extension/truncation		X

Materialize Access Group Attributes (MATAGAT)

Op Code (Hex)	Operand 1	Operand 2
03A2	Receiver	Access group

Operand 1: Space pointer.

Operand 2: System pointer.

ILE access

```
MATAGAT
     receiver     : space pointer;
 var access_group : system pointer
)
```

Description: The attributes of the access group and the identification of objects currently contained in the access group are materialized into the receiving object specified by operand 1.

Objects requested to be in the access group may:

- exist entirely in the access group,
- exist partially in the access group and partially outside the access group,
- or exist entirely outside the access group.

The machine may also use the access group for enabling programs to run within a process. In this case, the Process Control Space (PCS) object is considered to exist partially in the access group, even if the access group membership was not requested when the PCS was created.

Only objects which exist wholly or partially in the access group will be materialized.

The materialization must be aligned on a 16-byte boundary. The format is:

- Materialization size specification — Char(8)
  - Number of bytes provided for materialization — Bin(4)
  - Number of bytes available for materialization — Bin(4)
- Object identification — Char(32)
  - Object type — Char(1)
  - Object subtype — Char(1)
  - Object name — Char(30)
- Object creation options — Char(4)
  - Existence attributes — Bit 0

    0 = Temporary
    1 = Reserved
  - Space attribute — Bit 1

    0 = Fixed-length
    1 = Variable-length
  - Context — Bit 2

    0 = Addressability not in context

1 = Addressability in context

– Reserved (binary 0)	Bits 3-12
– Initialize space	Bit 13
– Reserved (binary 0)	Bits 14-31
• Reserved (binary 0)	Char(4)
• Size of space	Bin(4)
• Initial value of space	Char(1)
• Performance class	Char(4)
– Space alignment	Bit 0

0 = The space associated with the object is allocated to allow proper alignment of pointers at 16-byte alignments within the space. If no space is specified for the object, this value must be specified for the performance class.
1 = The space associated with the object is allocated to allow proper alignment of pointers at 16-byte alignments within the space as well as to allow proper alignment of input/output buffers at 512-byte alignments within the space.

– Reserved (binary 0)	Bits 1-4
– Default main storage pool	Bit 5

0 = Process main storage pool is used for this object.
1 = Machine default main storage pool is used for this object.

– Reserved (binary 0)	Bit 6
– Block transfer on implicit access state modification	Bit 7

0 = Minimum storage transfer size for this object is transferred. This value is 1 storage unit.
1 = Machine default storage transfer size is transferred. This value is 8 storage units.

– Reserved (binary 0)	Bits 8-31
• Reserved (binary 0)	Char(7)
• Context	System pointer
• Reserved (binary 0)	Char(16)
• Access group size	UBin(4)
• Available space in the access group	UBin(4)
• Number of objects in the access group	UBin(4)
• Reserved (binary 0)	Char(4)
• Access group object system pointer (repeated for each object currently contained in the access group)	System pointer

The receiver space contains the access group's attributes (as defined by the Create Access Group instruction), the current status of the access group, and a system pointer to each object assigned to the access group.

The **access group size** represents the total amount of space that has been allocated to the access group.

The **amount of available space** represents the amount of space that is available in the access group for additional objects.

The **number of objects in the access group** is a count of the number of objects that are currently contained in the access group. This value is also the number of times that the *access group object system pointer* below is repeated.

There is one **access group object system pointer** for each object currently assigned to the access group. The authorization field within each system pointer is not set.

Authorization Required

- Retrieve
  - Operand 2
  - Contexts referenced for address resolution

Lock Enforcement

- Materialize
  - Operand 2
  - Contexts referenced for address resolution

Exceptions

Exception		Operands 1	2	Other
06	Addressing			
	01 space addressing violation	X	X	
	02 boundary alignment	X	X	
	03 range	X	X	
	06 optimized addressability invalid	X	X	
08	Argument/parameter			
	01 parameter reference violation	X	X	
0A	Authorization			
	01 unauthorized for operation		X	
10	Damage encountered			
	04 system object damage state	X	X	X
	05 authority verification terminated due to damaged object			X
	44 partial system object damage	X	X	X
1A	Lock state			
	01 invalid lock state		X	X
1C	Machine-dependent exception			
	03 machine storage limit exceeded			X
20	Machine support			
	02 machine check			X
	03 function check			X
22	Object access			
	01 object not found	X	X	

Exception		Operands 1	Operands 2	Other
	02 object destroyed	X	X	
	03 object suspended	X	X	
	07 authority verification terminated due to destroyed object			X
	08 object compressed			X
24	Pointer specification			
	01 pointer does not exist	X	X	
	02 pointer type invalid	X	X	
	03 pointer addressing invalid object		X	
2E	Resource control limit			
	01 user profile storage limit exceeded			X
36	Space management			
	01 space extension/truncation			X
38	Template specification			
	03 materialization length exception	X		

Materialize Resource Management Data (MATRMD)

Op Code (Hex)	Operand 1	Operand 2
0352	Receiver	Control data

Operand 1: Space pointer.

Operand 2: Character(8) scalar

ILE access

```
MATRMD (
     receiver     : space pointer;
 var control_data : aggregate
)
```

Description: The data items requested by operand 2 are materialized into the receiving object specified by operand 1. Operand 2 is an 8-byte character scalar. The first byte identifies the generic type of information being materialized, and the remaining 7 bytes further qualify the information desired.

Operand 1 contains the materialization and has the following format:

- Materialization size specification — Char(8)
  - Number of bytes provided for materialization — Bin(4)
  - Number of bytes available for materialization — Bin(4)
- Time of day — Char(8)
- Resource management data — Char(*)

The remainder of the materialization depends on operand 2 and on the machine implementation. The following values are allowed for operand 2:

- Selection option — Char(1)

  Hex 01 = Materialize processor utilization data
  Hex 03 = Materialize storage management counters
  Hex 04 = Materialize storage transient pool information
  Hex 08 = Materialize machine address threshold data
  Hex 09 = Materialize main storage pool information
  Hex 0A = Materialize MPL control information
  Hex 0C = Materialize machine reserved storage pool information
  Hex 11 = Materialize user storage area 1
  Hex 12 = Materialize auxiliary storage information
  Hex 13 = Materialize multiprocessor utilizations
  Hex 14 = Materialize Storage pool tuning

- Reserved (binary 0) — Char(7)

The following defines the formats and values associated with each of the above materializations of resource management data.

Processor Utilization (Hex 01):

- Processor time since IPL (initial program load) — Char(8)

Processor time since IPL is the total amount of processor time used, both by instruction processes and internal machine functions, since IPL. The significance of bits within the field is the same as that defined for the time-of-day clock.

On a machine with more than one active processor, the value returned will be the average of the processor time used since IPL by all active processors.

Storage Management Counters (Hex 03):

- Access pending Bin(2)
- Storage pool delays Bin(2)
- Directory look-up operations Bin(4)
- Directory page faults Bin(4)
- Access group member page faults Bin(4)
- Microcode page faults Bin(4)
- Microtask read operations Bin(4)
- Microtask write operations Bin(4)
- Reserved Bin(4)

Access pending is a count of the number of times that a paging request must wait for the completion of a different request for the same page.

Storage pool delays is a count of the number of times that processes have been momentarily delayed by the unavailability of a main storage frame in the proper pool.

Directory look-up operations is a count of the number of times that auxiliary storage directories were interrogated, exclusive of storage allocation or deallocation.

Directory page faults is a count of the number of times that a page of the auxiliary storage directory was transferred to main storage, to perform either a look-up or an allocation operation.

Access group member page faults is a count of the number of times that a page of an object contained in an access group was transferred to main storage independently of the containing access group. This occurs when the containing access group has been purged or because portions of the containing access group have been displaced from main storage.

Microcode page faults is a count of the number of times a page of microcode was transferred to main storage.

Microtask read operations is a count of the number of transfers of one or more pages of data from auxiliary main storage on behalf of a microtask rather than a process.

Microtask write operations is a count of the number of transfers of one or more pages of data from main storage to auxiliary storage on behalf of a microtask, rather than a process.

Storage Transient Pool Information (Hex 04):

- Storage pool to be used for the transient pool Bin(2)

The pool number materialized is the number of the main storage pool, which is being used as the transient storage pool. A value of 0 indicates that the transient pool attribute is being ignored.

Machine Address Threshold Data (Hex 08):

- Total permanent addresses possible Char(8)
- Total temporary addresses possible Char(8)
- Permanent addresses remaining Char(8)
- Temporary addresses remaining Char(8)

• Permanent addresses threshold	Char(8)
• Temporary addresses threshold	Char(8)

Total permanent addresses possible is the maximum number of permanent addresses that can exist on the machine.

Total temporary addresses possible is the maximum number of temporary addresses that can exist on the machine.

Permanent addresses remaining is the number of permanent addresses that can still be created before address regeneration must be run.

Temporary addresses remaining is the number of temporary addresses that can still be created before address regeneration must be run.

Permanent addresses threshold is a number that, when it exceeds the number of permanent addresses remaining, causes the *machine address threshold exceeded* (hex 000C,05,01) event to be signaled. When the event is signaled, the threshold is reset to 0.

Temporary addresses threshold is a number that, when it exceeds the number of temporary addresses remaining, causes the *machine address threshold exceeded* (hex 000C,05,01) event to be signaled. When the event is signaled, the threshold is reset to 0.

Main Storage Pool Information (Hex 09):

• Machine minimum transfer size	Bin(2)
• Maximum number of pools	Bin(2)
• Current number of pools	Bin(2)
• Main storage size	Bin(4)
• Reserved (binary 0)	Char(2)
• Pool 1 minimum size	Bin(4)
• Individual main storage pool information (repeated once for each pool, up to the current number of pools)	Char(32)
– Pool size	Bin(4)
– Pool maintenance	Bin(4)
– Process interruptions (data base)	Bin(4)
– Process interruptions (nondata base)	Bin(4)
– Data transferred to pool (data base)	Bin(4)
– Data transferred to pool (nondata base)	Bin(4)
– Amount of pool not assigned to virtual addresses	Bin(4)
– Reserved (binary 0)	Char(4)

Machine minimum transfer size is the smallest number of bytes that may be transferred as a block to or from main storage.

Maximum number of pools is the maximum number of storage pools into which main storage may be partitioned. These pools will be assigned the logical identification beginning with 1 and continuing to the *maximum number of pools*.

Current number of pools is a user-specified value for the number of storage pools the user wishes to utilize. These are assumed to be numbered from 1 to the number specified. This number is fixed by the machine to be equal to the maximum number of pools.

Main storage size is the amount of main storage, in units equal to the *machine minimum transfer size*, which may be apportioned among main storage pools.

Pool 1 minimum size is the amount of main storage, in units equal to the *machine minimum transfer size*, which must remain in pool 1. This amount is machine and configuration dependent.

Individual main storage pool information is data in an array that is associated with a main storage pool by virtue of its ordinal position within the array. In the descriptions below, data base refers to all other data, including internal machine fields. *Pool size, pool maintenance, amount of pool not assigned to virtual addresses* and *data transferred information* is expressed in units equal to the *machine minimum transfer size* described above.

Pool size is the amount of main storage assigned to the pool.

Pool maintenance is the amount of data written from a pool to secondary storage by the machine to satisfy demand for resources from the pool. It does not represent total transfers from the pool to secondary storage, but rather is an indication of machine overhead required to provide primary storage within a pool to requesting processes.

Process interruptions (data base and nondata base) is the total number of interruptions to processes (not necessarily assigned to this pool) which were required to transfer data into the pool to permit instruction execution.

Data transferred to pool (data base and nondata base) is the amount of data transferred from auxiliary storage to the pool to permit instruction execution and as a consequence of set access state, implicit access group movement, and internal machine actions.

The **amount of the pool not assigned to virtual addresses** represents the storage available to be used for new transfers into the main storage pool without displacing any virtual data already in the pool. After a pool's size has been modified (via the Modify Resource Management Controls instruction), this value will be inaccurate until a Modify Resource Management Controls instruction, option hex 0D, is issued to flush the modified pool. After which time, this value will be accurate until the pool size is again modified. The value returned will not include any storage that has been reserved for load/dump sessions active in the pool.

Multiprogramming Level Control Information (Hex 0A):

Field	Type
• Machine-wide MPL control	Char(16)
— Machine maximum number of MPL classes	Bin(2)
— Machine current number of MPL classes	Bin(2)
— MPL (max)	Bin(2)
— Ineligible event threshold	Bin(2)
— MPL (current)	Bin(2)
— Number of processes in ineligible state	Bin(2)
— Reserved	Char(4)
• MPL class information (repeated for each MPL class, from 1 to the current number of MPL classes)	Char(16)
— MPL (max)	Bin(2)

— Ineligible event threshold	Bin(2)
— Current MPL	Bin(2)
— Number of processes ineligible state	Bin(2)
— Number of processes assigned to class	Bin(2)
— Transitions (active to ineligible)	Bin(2)
— Transitions (active to MI wait)	Bin(2)
— Transitions (MI wait to ineligible)	Bin(2)

Machine-Wide MPL Control:

Maximum number of MPL classes is the largest number of MPL classes allowed in the machine. These are assumed to be numbered from 1 to the maximum.

Current number of MPL classes is a user-specified value for the number of MPL classes in use. They are assumed to be numbered from 1 to the current number.

MPL (max) is the maximum number of processes which may concurrently be in the active state in the machine.

Ineligible event threshold is a number which, if exceeded by the *machine number of ineligible processes* defined below, will cause the *machine ineligible threshold exceeded* (hex 000C,04,01) event to be signaled. When the event is signaled, this value is set by the machine to 65,535.

MPL (current) is the current number of processes in the active state.

Number of processes in the ineligible state is the number of processes not currently active because of enforcement of both the machine and class MPL rules.

MPL Class Information

MPL class information is data in an array that is associated with an MPL class by virtue of its ordinal position within the array.

MPL (max) is the number of processes assigned to the class which may be concurrently active.

Ineligible event threshold, **MPL (current)**, and **number of processes in ineligible state** are as defined above but apply only to processes assigned to the class.

Number of processes assigned to class is the total number of processes, in any state, assigned to the pool.

Transitions count is the total number of transitions by processes assigned to a class as follows:

1. Active state to ineligible state
2. Active state to wait
3. Wait state to ineligible state

Note that transitions from wait state to active state can be derived as (2 - 3) and transitions from ineligible state to active state as (1 + 3). These numbers are Bin(2) and are maintained by the machine without regard to overflow conditions.

Machine Reserved Storage Pool Information (Hex 0C):

• Current number of pools	Bin(2)
• Reserved	Char(6)
• Individual main storage pool information (repeated once for each pool, up to the current number of pools)	Char(16)
— Pool size	Bin(4)
— Machine portion of the pool	Bin(4)

- Number of load/dump sessions — Bin(2)
- Reserved — Char(6)

Pool size is the amount of main storage assigned to the pool (including the machine reserved portion).

Machine portion of the pool specifies the amount of storage from the pool that is dedicated to machine functions.

User storage area 1 (Hex 11):

- User data — Char(*)

The **user data** previously stored internally in the machine through usage of the corresponding option on the Modify Resource Management Controls instruction is materialized into the receiver. The operand 1 template, for this option, must start on a 16 byte boundary and any pointers contained in the user data are preserved in the materialization.

The length value materialized in the number of bytes available for materialization field of operand 1 specifies the length of the entire operand 1 template and is limited, through checks performed on the modify operation, to a maximum value of 65,504 (64K-32) bytes. The actual length of the user data materialized is calculated by subtracting 16 from the length value for the total template length.

Auxiliary Storage Information (Hex 12):

The *auxiliary storage information* describes the ASPs (auxiliary storage pools) which are configured within the machine and the units of auxiliary storage currently allocated to an ASP or known to the machine but not allocated to an ASP.

Note that contrary to the normal case of being able to modify the values materialized by this option through use of the Modify Resource Management Controls instruction, modification of most of the auxiliary storage configuration is performed using functions available in the Dedicated Service Tool (DST).

Also note that through appropriate setting of the number of bytes provided field for operand 1, the amount of information to be materialized for this option can be reduced thus avoiding the processing for unneeded information. As an example, by setting this field to only provide enough bytes for the common 16 byte header, plus the option Hex 12 control information, plus the system ASP entry of the ASP information, you can get just the information up through the system ASP entry returned and avoid the overhead for the user ASPs and unit information.

Control information (occurs just once) — Char(64)

- Number of ASPs — Bin(2)
- Number of allocated auxiliary storage units — Bin(2)

  Note: Number of configured, non-mirrored units + number of mirrored pairs
- Number of unallocated auxiliary storage units — Bin(2)
- Control flags — Char(1)
  - Main storage dump area unavailable — Bit 0
  - Reserved (binary 0) — Bits 1-7
- Reserved (binary 0) — Char(1)
- Maximum unprotected space used — Char(8)
- Current unprotected space in use — Char(8)
- Checksum main storage — Bin(4)

• Unit information offset	Bin(4)
• Number of pairs of mirrored units	Bin(2)
• Mirroring main storage	Bin(4)
• Reserved (binary 0)	Char(2)
• Total temporary space including LS	Char(8)
• Number of bytes in a page	Bin(4)
• Reserved (binary 0)	Char(12)
ASP information	Char(160)

(Repeated once for each ASP. Located immediately after the control information above. ASP 1, always configured, is first. Configured user ASPs follow in ascending numerical order.)

• ASP number	Char(2)
• ASP control flags	Char(1)
– Suppress threshold exceeded event	Bit 0
– User ASP overflow	Bit 1
– Checksum protection	Bit 2
– Unprotected space overflow	Bit 3
– ASP mirrored	Bit 4
– User ASP MI State	Bit 5
– ASP overflow storage available	Bit 6
– Reserved (binary 0)	Bit 7
• ASP overflow recovery result	Char(1)
– Successful	Bit 0
– Failed due to insufficient free space	Bit 1
– Cancelled	Bit 2
– Reserved (binary 0)	Bits 3-7
• Reserved (binary 0)	Char(4)
• ASP media capacity	Char(8)
• Reserved	Char(8)
• ASP space available	Char(8)
• ASP event threshold	Char(8)
• ASP event threshold percentage	Bin(2)
• Reserved (binary 0)	Char(6)
• ASP system storage	Char(8)
• ASP overflow storage	Char(8)
• Space allocated to the Error log	Bin(4)
• Space allocated to the machine log	Bin(4)
• Space allocated to the machine trace	Bin(4)
• Space allocated for main store dump	Bin(4)
• Space allocated to the microcode	Bin(4)

• Reserved	Char(12)
• ASP checksum information	Char(64)
— Protected space capacity	Char(8)
— Unprotected space capacity	Char(8)
— Protected space available	Char(8)
— Unprotected space available	Char(8)
— Unprotected space on each checksummed unit	Bin(4)
— Reserved	Char(28)

Unit information Char(208)
(Consists of one entry each for the configured, non-mirrored units and one unit of the mirrored pairs, the non-configured units, and the other unit of the mirrored pairs.

An allocated storage unit (ASU) is either an allocated, non-mirrored unit or a mirrored pair. Note that the mirrored pair counts only as one ASU. When used without qualification, the term unit refers to an ASU.

Unit information start may be located by the Unit Information Offset in the control information.)

• Device type	Char(8)
— Disk Type	Char(4)
— Disk Model	Char(4)
• Device identification	Char(8)
— Unit number	Char(2)
— Serial number	Char(4)
— Reserved	Char(2)
• Unit relationship	Char(4)
— Reserved	Char(1)
— Bus information	Char(1)
- Bus number	Bits 0-2
- Bus unit (IOP)	Bits 3-7
— Controller identification	Char(1)
— Actuator identification	Char(1)
• Unit ASP number	Char(2)
• Logical mirrored pair status	Char(1)
— Unit mirrored	Bit 0
— Mirrored unit protected	Bit 1
— Mirrored pair reported	Bits 2
— Reserved	Bits 3-7
• Mirrored unit status	Char(1)
• Unit media capacity	Char(8)
• Unit storage capacity	Char(8)
• Unit space available	Char(8)
• Unit reserved system space	Char(8)
• Unit relationship	Char(6)

– Bus information	Char(2)
- Bus number	Char(1)
- Bus unit (IOP)	Char(1)
– Unit address	Char(4)
- Controller identification	Char(1)
- Actuator identification	Char(1)
- Reserved	Char(2)
• Unit control flags	Char(2)
– Unit in checksummed ASP	Bit 0
– Unit is device parity protected	Bit 1
– Subsystem is active	Bit 2
– Unit in subsystem has failed	Bit 3
– Other unit in subsystem has failed	Bit 4
– Subsystem runs in degraded mode	Bit 5
– Hardware failure	Bit 6
– Device parity protection is being rebuilt	Bit 7
– Unit is not ready	Bit 8
– Unit is write protect	Bit 9
– Unit is busy	Bit 10
– Unit is not operational	Bit 11
– Status is not recognizable	Bit 12
– Status is not available	Bit 13
– Unit is Read/Write protected	Bit 14
– Reserved (binary 0)	Bit 15
Bits 2 to 14 are mutually exclusive.	
• Reserved	Char(16)
• Unit checksum information	Char(64)
– Unit redundancy space	Char(8)
– Unit protected space capacity	Char(8)
– Unit protected space available	Char(8)
– Unit unprotected space capacity	Char(8)
– Unit unprotected space available	Char(8)
– Unit checksum set number	Char(2)
– Reserved (binary 0)	Char(22)
• Unit usage information	Char(64)
– Blocks transferred to main storage	Bin(4)
– Blocks transferred from main storage	Bin(4)
– Requests for data transfer to main storage	Bin(4)
– Requests for data transfer from main storage	Bin(4)
– Permanent blocks transferred from main storage	Bin(4)

— Requests for permanent data transfer from main storage	Bin(4)
— Redundancy blocks transferred from main storage	Bin(4)
— Requests for redundancy data transfer from main storage	Bin(4)
— Reserved (binary 0)	Char(32)

Number of ASPs is the number of ASPs configured within the machine. One, the minimum value, indicates just the system ASP exists and that there are no user ASPs configured. Up to 15 user ASPs can be configured. Values greater than one indicate how many user ASPs are configured in addition to the system ASP. The system ASP always exists.

Number of allocated auxiliary storage units is the number of configured units logically addressable by the system as units. This is the number of configured, non-mirrored units plus the number of mirrored pairs allocated to the ASPs. The total number of units (actuator arms) on the system is the sum of the allocated auxiliary storage units plus the number of unallocated auxiliary storage units plus the number of pairs of mirrored units. For example, each 9335 enclosure represents two units. Information on these units is materialized as part of the unit information. Any two units of the same type and size may be associated to form a mirrored pair. Association of two units as a mirrored pair reduces the amount of logically available storage by the number of bytes contained on one of the mirrored units in the mirrored pair.

Number of unallocated auxiliary storage units is the number of auxiliary storage units that are currently not allocated to an ASP. Information on these units is materialized as part of the unit information.

The **main storage dump area unavailable** flag indicates whether or not the main storage dump area on disk is unavailable. A value of binary 1 indicates it is unavailable; binary 0 indicates it is available. The condition where it is unavailable can arise when main storage is added to the machine, but during subsequent IPL processing the machine can not free up space on the load source disk unit for the additional dump area needed. This occurs when there is insufficient space available on the other disk units in the system ASP to allow for movement of object allocations off of the load source unit. The corrective action is to free up space in the system ASP and reIPL the machine so the allocation of additional space to the dump area can be completed.

The main storage dump area is important for recovery and diagnostic purposes. It is used by the machine during certain hardware and power failures to capture a main storage dump which is used to minimize the object damage which can result. It is also used by the machine during certain software logic failures to capture a main storage dump which is used to determine the cause of the failure.

Maximum unprotected space used (Checksum field) is the largest number of bytes of unprotected storage used at any one time since the last IPL of the machine. When checksum protection is not in effect for the system ASP, this field describes the amount of unprotected storage that would have been used if checksum protection had been in effect.

Current unprotected space used (Checksum field) is the current number of bytes of unprotected storage in use. When checksum protection is not in effect for the system ASP, this field describes the amount of unprotected storage that would be in use if checksum protection was in effect.

Checksum main storage (Checksum field) is the number of bytes of main storage reserved in the machine storage pool for checksum usage.

Unit information offset is the offset, in bytes, from the start of the operand 1 materialization template to the start of the unit information. This value can be added to a space pointer addressing the start of operand 1 to address the start of the unit information.

Number of pairs of mirrored units represents the number of mirrored pairs in the system. Each mirrored pair consists of two mirrored units; however, only one of the two mirrored units is guaranteed to be operational.

Mirroring main storage is the number of bytes of main storage in the machine storage pool used by mirroring. This increases when mirror synchronization is active. This amount of storage is directly related to the number of mirrored pairs.

Total temporary space including LS is the number of bytes of temporary storage allocated on the system. This includes the temporary storage allocated on the load source unit.

Number of bytes in a page is the number of bytes in a single page. This can be used to convert fields that are given in pages into the correct number of bytes.

ASP information is repeated once for each ASP configured within the machine. The number of ASPs configured is specified by the *number of ASPs* field. ASP 1, the system ASP is materialized first. Because the system ASP always exists, its materialization is always available. The user ASPs which are configured are materialized after the system ASP in ascending numerical order. There may be gaps in the numerical order. That is, if just user ASPs 3 and 5 are configured, only information for them is materialized producing information on just ASP 1, ASP 3 and ASP 5 in that order.

ASP number uniquely identifies the auxiliary storage pool. The ASP number may have a value from 1 through 16. A value of 1 indicates the system ASP. A value of 2 through 16 indicates a user ASP.

Suppress threshold exceeded event flag indicates whether or not the machine is suppressing signaling of the related event when the event threshold in effect for this ASP has been exceeded. A value of binary 1 indicates that the signaling is being suppressed; binary 0 indicates that the signaling is not being suppressed. The default after each IPL of the machine is that the signaling is not suppressed; i.e. default is binary 0. For the system ASP, this flag is implicitly set to binary 1 by the machine when the *machine auxiliary storage threshold exceeded* (hex 000C,02,01) event is signaled. For a user ASP, this flag is implicitly set to binary 1 by the machine when the *user auxiliary storage threshold exceeded* (hex 000C,02,02) event is signaled. This is done to avoid repetitive signaling of the event when the threshold exceeded condition occurs. Option Hex 12 of the Modify Resource Management Controls instruction can be used to explicitly reset the suppression of the threshold exceeded event when it is desirable to again have the machine detect the threshold exceeded condition and signal the related event.

User ASP overflow flag (Checksum field) indicates whether or not object allocations directed into the user ASP have overflowed into the system ASP. A value of binary 1 indicates overflow; binary 0 indicates no overflow. This flag does not apply to the system ASP, and is always set to a binary 0 for it.

Checksum protection flag specifies whether or not the ASP is under checksum protection. A value of binary 1 indicates that checksum protection is in effect; a value of binary 0 indicates it is not.

Unprotected space overflow flag (Checksum field) specifies whether or not allocations for unprotected data in the ASP have exceeded the unprotected space capacity and overflowed into the area normally used for allocation of protected data. A value of binary 1 indicates that such overflow has occurred; a value of binary 0 indicates it has not. This status is set when the *ASP unprotected space overflow* (hex 000C,02,04) event is signaled; it is reset automatically on the subsequent IMPL of the machine. Because unprotected storage is used primarily for allocation of temporary objects which are automatically deallocated as part of the IPL process, the overflowed allocations are freed up at IPL, providing for the automatic reset of the overflow condition. Because unprotected storage is only allowed for the system ASP, this flag is only applicable to the system ASP.

ASP mirrored flag specifies whether or not the ASP is configured to be mirror protected. A value of binary 1 indicates that ASP mirror protection is configured. Refer to the mirror unit protected flag to

determine if mirror protection is active for each mirrored pair. A value of binary 0 indicates that none of the units associated with the ASP are mirrored.

User ASP MI State indicates the state of the User ASP. A value of binary 1 indicates that the User ASP is in the 'new' state. This means that a context may be allocated in this User ASP. A value of binary 0 indicates that the User ASP is in the 'old' state. This means that there are no contexts allocated in this User ASP. This flag has no meaning for the System ASP and will always be set to binary 0 for it.

ASP overflow storage available flag indicates whether or not the amount of auxiliary storage that has overflowed from the user ASP into the system ASP is available. A value of binary 1 indicates that the amount is maintained by the machine and available in the *ASP overflow storage* field. A value of binary 0 indicates that the amount is not available.

ASP overflow recovery result flags indicate the result of the ASP overflow recovery operation which is performed during an IPL upon request by the user. When this operation is requested, the machine attempts to recover the user ASP from an overflow condition by moving overflowed auxiliary storage from the system ASP back to the user ASP during the Storage Management recovery step of an IPL. The successful flag is set to a value of binary 1 when all the overflowed storage was successfully moved. The failed due to insufficient free space flag is set to a value of binary 1 when there is not sufficient free space in the user ASP to move all the overflowed storage. The cancelled flag is set to a value of binary 1 when the operation was cancelled prior to completion (e.g., system power loss, user initiated IPL).

ASP media capacity specifies the total space, in number of bytes of auxiliary storage, on the storage media allocated to the ASP. This is just the sum of the unit media capacity fields for (1) the units allocated to the ASP or (2) the mirrored pairs in the ASP.

ASP space available is the number of bytes of auxiliary storage that is not currently assigned to objects or internal machine functions, and therefore, is available for allocation in the ASP when the ASP is not under checksum protection. Note that a mirrored pair counts for only one unit. When the ASP is under checksum protection, this value is meaningless and the ASP checksum information describes the space available values.

ASP event threshold specifies the minimum value for the number of bytes of auxiliary storage available in the ASP prior to the signaling of the appropriate threshold exceeded event. The threshold exceeded condition occurs when either the protected space available value or the ASP space available value, depending upon whether checksum protection is or isn't in effect for the ASP, becomes equal to or less than the ASP event threshold value. This condition causes either the *auxiliary storage threshold exceeded* (hex 000C,02,01) event, for the system ASP, or the *user ASP threshold exceeded event* (hex 000C,02,02), for a user ASP, to be signaled. Redundant signaling of these events is suppressed as indicated by the setting of the suppress threshold exceeded event flag. Refer to the definition of the suppress threshold exceeded event flag for more information.

The *ASP event threshold* value is calculated from the the ASP event threshold percentage value by multiplying either the protected space capacity value or the ASP media capacity value, depending upon whether checksum protection is or isn't in effect for the ASP, by the ASP event threshold percentage and subtracting the product from that same capacity value.

ASP event threshold percentage specifies the auxiliary storage space utilization threshold as a percentage of either the protected space capacity or the ASP media capacity, depending upon whether checksum protection is or isn't in effect for the ASP. This value is used, as described above, to calculate the ASP event threshold value. This value can be modified through use of Dedicated Service Tool DASD configuration options.

ASP system storage specifies the amount of system storage currently allocated in the ASP in bytes. This storage will not be calculated when determining if the user ASP MI state can be changed.

ASP overflow storage indicates the number of bytes of auxiliary storage that have overflowed from the user ASP into the system ASP. This value is valid only if the ASP overflow storage available flag is set to a value of binary 1.

Space allocated to the Error log this is the number of pages of auxiliary storage that are allocated to the error log. This field only applies to the System ASP.

Space allocated to the machine log this is the number of pages of auxiliary storage that are allocated to the machine log. This field only applies to the System ASP.

Space allocated to the machine trace this is the number of pages of auxiliary storage that are allocated to the machine trace. This field only applies to the System ASP.

Space allocated for Main Store Dump this is the number of pages of auxiliary storage that are allocated to the main store dump space. The contents of main store are written to this location for some system terminations. This field only applies to the System ASP.

Space allocated to microcode this is the number of pages of auxiliary storage that are allocated for microcode and space used by the microcode. The space allocated to the error log, machine log, machine trace, and main store dump space is not included in this field. This field only applies to the System ASP.

ASP checksum information (Checksum field) specifies capacity and space available values that apply when the ASP is under checksum protection. In this case, the units of auxiliary storage allocated to ASP are formatted with areas for protected data, unprotected data, and redundancy data. Information on the protected and unprotected space is provided both here in these fields on an ASP basis and under unit information on a per unit basis. Information on redundancy space is only provided under unit information on a per unit basis. When the ASP is not under checksum protection, the values of these fields are meaningless.

Protected space capacity (Checksum field) specifies the total number of bytes of auxiliary storage formatted for the storage of protected data in the ASP.

Unprotected space capacity (Checksum field) specifies the total number of bytes of auxiliary storage formatted for storage of unprotected data in the ASP. Since unprotected space is only allowed in the system ASP, this information is only applicable to the system ASP.

Protected space available (Checksum field) specifies the number of bytes of auxiliary storage formatted for storage of protected data that are not currently assigned to objects or internal machine functions, and therefore, are available for allocation in the ASP.

Unprotected space available (Checksum field) specifies the number of bytes of auxiliary storage formatted for storage of unprotected data that are not currently assigned to objects or internal machine functions, and therefore, are available for allocation in the ASP. Since unprotected space is only allowed in the system ASP, this information is only applicable to the system ASP.

Unprotected space on each checksummed unit (Checksum field) specifies the number of megabytes (millions of bytes) of auxiliary storage formatted for storage of unprotected data on each unit allocated to a checksum set in the ASP. Using the Dedicated Service Tool to modify this value provides for altering the relation of the protected versus unprotected space capacity values. Since unprotected space is only allowed in the system ASP, this information is only applicable to the system ASP.

Unit information is materialized in the following order:

Group 1: Configured units consisting of non-mirrored units and mirrored units.

Group 2: Non-configured units

Group 3: Configured units consisting of mirrored units.

The **unit information** is located by the **unit information offset** field which specifies the offset from the beginning of the operand 1 template to the start of the unit information. The number of entries for each of the three groups listed above is defined as follows:

Group 1: Number of non-mirrored, configured units + number of mirrored pairs

Group 2: Number of non-configured storage units

Group 3: Number of mirrored pairs

For unallocated units, the device type, device identification, unit relationship, and unit media capacity fields contain meaningful information. The remaining fields have no meaning for unallocated units because the units are not currently in use by the system. Mirrored unit entries contain either current or last known information. The last known data consists of the mirrored unit status, disk type, disk model, unit ASP number, disk serial number, and unit address. Last known information is provided when the Mirrored Unit Reported field is a binary zero.

Disk type identifies the type of disk enclosure containing this auxiliary storage unit. This is the four byte character field from the vital product data for the disk device which identifies the type of drive. For example, the value is character string '9332' for a 9332 device and '9335' for a 9335 device.

Disk model identifies the model of the type of disk enclosure containing this auxiliary storage unit. This is the four byte character field from the vital product data for the disk device which identifies the model of the drive. For example, the value is character string '0200' for a model 200 9332 device and '0400' for a model 400 9332 device.

Unit number uniquely identifies each non-mirrored unit or mirrored pair among the configured units. Both mirrored units of a mirrored pair have the same unit number. The value of the unit number is assigned by the system when the unit is allocated to an ASP. For unallocated units, the unit number is set to binary zero.

Serial number specifies the serial number of the device containing this auxiliary storage unit. This is the four byte serial number field from the vital product data for the disk device.

Bus number identifies the I/O Bus to which the disk device containing this auxiliary storage unit is connected.

Bus unit (IOP) identifies the I/O Processor used to access the controller for the disk device containing this auxiliary storage unit.

Controller identification specifies the controller for the disk device containing this auxiliary storage unit.

Actuator identification specifies the actuator associated with this auxiliary storage unit in the disk device containing it.

Unit ASP number specifies the ASP to which this unit is currently allocated. A value of 1 specifies the system ASP. A value from 2 through 16 specifies a user ASP and correlates to the user ASP number field in the user ASP information entries. A value of 0 indicates that this unit is currently unallocated.

Unit mirrored flag indicates that this unit number represents a mirrored pair. This bit is materialized with both mirrored units of a mirrored pair.

Mirrored unit protected flag indicates the mirror status of a mirrored pair. A value of 1 indicates that both mirrored units of a mirrored pair are active. A 0 indicates that one mirrored unit of a mirrored pair is not active. Active means that both units are on line and fully synchronized (ie. the data is identical on both mirrored units).

Mirrored unit reported flag indicates that a mirrored unit reported as present. The mirrored unit reported present during or following IMPL. Current attachment of a mirrored unit to the system **cannot** be inferred from this bit. A 0 indicates that the mirrored unit being materialized is missing. The last known information pertaining to the missing mirrored unit is materialized. A 1 indicates that the mirrored unit being materialized has reported. The information for this reported unit is current to the last time it reported status to the system.

Mirrored unit status indicates mirrored unit status.

A value of 1 indicates that this mirrored unit of a mirrored pair is active (ie. on-line with current data).

A value of 2 indicates that this mirrored unit is being synchronized.

A value of 3 indicates that this mirrored unit is suspended.

Mirrored unit status is stored as binary data and is valid only when the unit mirrored flag is on.

Unit media capacity is the space, in number of bytes of auxiliary storage, on the non-mirrored unit or mirrored pair, that is, the capacity of the unit prior to any formatting or allocation of space by the system it is attached to. For a mirrored pair, this space is the number of bytes of auxiliary storage on either one of the mirrored units. The space is identical on both of the mirrored units. Caution, do not attempt to add the capacities of the two units of a mirrored pair together.

Unit space available is the number of bytes of secondary storage space that is not currently assigned to objects or internal machine functions, and therefore, is available for allocation on the unit (or the mirrored pair) when the ASP containing it is not under checksum protection. When the ASP containing the unit is under checksum protection, this value is meaningless and the Unit checksum information describes the space available values. For a mirrored pair, this space is the number of bytes of auxiliary storage available on either one of the mirrored units. The space is identical on both of the mirrored units. Caution, do not attempt to add the capacities of the two units of a mirrored pair together.

Unit reserved system space is the total number of bytes of auxiliary storage on the unit reserved for use by the machine. This storage is not available for storing objects, redundancy data, and other internal machine data. When the unit is not in a checksum set, the unit checksum set number contains a value of zero, this reserved space is included in the ASP and unit media capacity fields and reduces the corresponding space available values. When the unit is in a checksum set, the unit checksum set number is nonzero, this reserved space is not included in the ASP and unit checksum information fields which provide capacity and space information and, therefore, does not reduce the corresponding space available values.

Bus number identifies the I/O Bus to which the disk device containing this auxiliary storage unit is connected.

Bus unit (IOP) identifies the I/O Processor used to access the controller for the disk device containing this auxiliary storage unit.

Controller identification specifies the controller for the disk device containing this auxiliary storage unit.

Actuator identification specifies the actuator associated with this auxiliary storage unit in the disk device containing it.

Unit in checksummed ASP - a value of 1 indicates that this unit is configured in an ASP that is checksummed. It does not indicate whether or not the unit is in a checksum set.

Unit is device parity protected - a value of 1 indicates that this unit is device parity protected.

| **Subsystem is active** indicates whether the array subsystem is active.

If the **unit in subsystem has failed** field is 1, the unit in an array subsystem being addressed has failed. Data protection for this subsystem is no longer in effect.

If the **other unit in subsystem has failed** field is 1, the unit being addressed is operational, but another unit in the array subsystem has failed. Data protection for this subsystem is no longer in effect.

If the **subsystem runs in degraded mode** field is 1, the array subsystem is operational and data protection for this subsystem is in effect, but a failure that may affect performance has occurred. It must be fixed.

If the **hardware failure** field is 1, the array subsystem is operational and data protection for this subsystem is in effect, but hardware failure has occurred. It must be fixed.

If the **device parity protection is being rebuilt** field is 1, the device parity protection for this device is being rebuilt following a repair action.

If the **unit is not ready** field is 1, the unit being addressed is not ready for I/O operation.

If the **unit is write protected** field is 1, the write operation is not allowed on the unit being addressed.

If the **unit is busy** field is 1, the unit being addressed is busy.

If the **unit is not operational** field is 1, the unit being addressed is not operational. The status of the device is not known.

If the **unit is not recognizable** field is 1, the unit being addressed has an unexpected status. I.e. the unit is operational, but its status returned to Storage Management from IOP is not one of those previously described.

If the **status is not available** field is 1, the machine is not able to communicate with I/O processor. The status of the device is not known.

If the **unit is Read/Write protected** is 1, a DASD array may be in the read/write protected state when there is problem, such as cache problem, configuration problem, or some other array problems that could create a data integrity exposure.

Unit checksum information (Checksum field) specifies capacity and space available values that apply when the ASP containing the unit is under checksum protection. In this case, when the unit is in a checksum set, the unit checksum set number is nonzero, the unit is formatted with areas for protected data, unprotected data, and redundancy data and these fields provide information relating to those areas. If the unit is not allocated to a checksum set, the unit checksum set number contains a value of zero, it is only formatted for the storage of unprotected data and the other capacity values will be zero. When the ASP containing the unit is not under checksum protection, the values of these fields are meaningless, except that the unit checksum set number field will contain a zero value.

Unit redundancy space (Checksum field) is the total number of bytes of auxiliary storage on the unit formatted for use for redundancy data. This storage is not available for storing objects and other internal machine data.

Unit protected space capacity (Checksum field) is the number of bytes of auxiliary storage formatted for storage of protected data on the unit. This field is only nonzero if this unit is allocated to a checksum set. Units not allocated to a checksum set contain no protected storage area. This value does not include the size of any data redundancy area which may have been formatted on the unit as well.

Unit protected space available (Checksum field) is the number of bytes of protected space on secondary storage available for allocation on the unit; that is, not currently assigned to objects or internal machine functions. This field is only nonzero if this unit is allocated to a checksum set. Units not

allocated to a checksum set contain no protected storage area, unless they are mirrored. All space of a mirrored pair is protected.

:Unit unprotected space capacity (Checksum field) is the number of bytes of auxiliary storage formatted for storage of unprotected data on the unit. This value does not include the size of any data redundancy area which may have been formatted on the unit as well. Since unprotected space is only allowed in the system ASP, this information is only applicable to units allocated to the system ASP.

Unit unprotected space available (Checksum field) is the number of bytes of unprotected space on secondary storage that are not currently assigned to objects or internal machine functions, and therefore, are available for allocation on the unit. Since unprotected space is only allowed in the system ASP, this information is only applicable to units allocated to the system ASP.

Unit checksum set number (Checksum field) specifies the checksum set to which this unit is currently allocated. A nonzero value specifies the number of the checksum set. A zero value specifies that the unit is currently not assigned to a checksum set.

Unit usage information specifies statistics relating to usage of the unit. For unallocated units, these fields are meaningless.

Blocks transferred to/from main storage fields specify the number of 512-byte blocks transferred for the unit since the last IMPL. These values wrap around to zero and continue counting in the case of an overflow of the field with no indication of the overflow having occurred.

Requests for data transfer to/from main storage fields specify the number of data transfer (I/O) requests processed for the unit since the last IMPL. These values wrap around to zero and continue counting in the case of an overflow of the field with no indication of the overflow having occurred. These values are not directly related to the number of blocks transferred for the unit because the number of blocks to be transferred for a given transfer request can vary greatly.

Permanent blocks transferred from main storage specifies the number of 512-byte blocks of permanent data transferred from main storage to auxiliary storage for the unit since the last IMPL. In the case of an overflow of the field, this value wraps around back to zero and continues counting, with no indication of the overflow condition having occurred.

Requests for permanent data transfer from main storage specifies the number of transfer (I/O) requests for transfers of permanent data from main storage to auxiliary storage that have been processed for the unit since the last IMPL. In the case of an overflow of the field, this value wraps around back to zero and continues counting, with no indication of the overflow condition having occurred. This value is not directly related to the permanent blocks transferred from main storage value for the unit ASP because the number of blocks to be transferred for any particular transfer request can vary greatly.

Redundancy blocks transferred from main storage (Checksum field) specifies the number of 512-byte blocks of redundancy data transferred from main storage to auxiliary storage for the unit since the last IMPL. In the case of an overflow of the field, this value wraps around back to zero and continues counting, with no indication of the overflow condition having occurred. This field is only meaningful for a unit in a checksum set.

Requests for redundancy data transfer from main storage (Checksum field) specifies the number of transfer (I/O) requests for transfers of redundancy data from main storage to auxiliary storage that have been processed for the unit since the last IMPL. In the case of an overflow of the field, this value wraps around back to zero and continues counting, with no indication of the overflow condition having occurred. This value is not directly related to the redundancy blocks transferred from main storage value for the unit because the number of blocks to be transferred for any particular transfer request can vary greatly. This field is only meaningful for a unit in a checksum set.

Multiprocessor utilizations (Hex 13):

• Number of processors configured on the machine	Bin(2)
• Number of configured processors currently active on the machine	Bin(2)
• Bit map of processors currently active on the machine	Char(2)
– Processor 1 is active	Bit 0
– Processor 2 is active	Bit 1
– Processor 3 is active	Bit 2
– Processor 4 is active	Bit 3
– Processor 5 is active	Bit 4
– Processor 6 is active	Bit 5
– Processor 7 is active	Bit 6
– Processor 8 is active	Bit 7
– Processor 9 is active	Bit 8
– Processor 10 is active	Bit 9
– Processor 11 is active	Bit 10
– Processor 12 is active	Bit 11
– Processor 13 is active	Bit 12
– Processor 14 is active	Bit 13
– Processor 15 is active	Bit 14
– Processor 16 is active	Bit 15
• Reserved	Char(2)
• Array of CHAR(8) processor time used since IPL values. Repeated once for each active processor.	Char(128)
– Processor 1 time busy since IPL	Char(8)
– Processor 2 time busy since IPL	Char(8)
– Processor 3 time busy since IPL	Char(8)
– Processor 4 time busy since IPL	Char(8)
– Processor 5 time busy since IPL	Char(8)
– Processor 6 time busy since IPL	Char(8)
– Processor 7 time busy since IPL	Char(8)
– Processor 8 time busy since IPL	Char(8)
– Processor 9 time busy since IPL	Char(8)
– Processor 10 time busy since IPL	Char(8)
– Processor 11 time busy since IPL	Char(8)
– Processor 12 time busy since IPL	Char(8)
– Processor 13 time busy since IPL	Char(8)
– Processor 14 time busy since IPL	Char(8)

- Processor 15 time busy since IPL — Char(8)
- Processor 16 time busy since IPL — Char(8)

The **number of active processors** will always be less than or equal to the **number of configured processors**. The **active processor bit map** will only indicate active processors within the first (leftmost) *number of configured processors* number of bits. The significance of bits within the time busy fields are the same as that defined for the time-of-day clock. This option always returns a *number of bytes available for materialization* equal to the length of the entire structure detailed above (it does not vary with the number of configured or active processors).

Storage pool tuning (Hex 14):

- Control information (occurs just once) — Char(16)
- Current number of pools — Bin(2)
- Reserved (binary 0) — Char(14)

Pool information (repeated once for each pool) — Char(104)

- Type of pool tuning — Char(1)
  - Hex 00 = No tuning is being done for the pool
  - Hex 10 = Static tuning
  - Hex 20 = Dynamic tuning of transfers to main storage
  - Hex 30 = Dynamic tuning of transfers to main storage and to auxiliary storage
- Changed page handling — Char(1)
  - Hex 00 = System page replacement algorithm handles changed pages
  - Hex 10 = Periodically transfer changed pages to auxiliary storage
- Reserved (binary 0) — Char(14)
- Nondatabase objects — Char(8)
  - Blocking factor — Char(2)
    - Hex 0000 = Use the default system value
    - Hex 0008 = Transfer data between main storage and auxiliary in blocks of 4K.
    - Hex 0010 = Transfer data between main storage and auxiliary in blocks of 8K.
    - Hex 0020 = Transfer data between main storage and auxiliary in blocks of 16K.
    - Hex 0040 = Transfer data between main storage and auxiliary in blocks of 32K.
  - Reserved (binary 0) — Char(6)
- Reserved (binary 0) — Char(16)
- Handling of database objects by class (repeat for each of the four classes). — Char(8)
  - Blocking factor — Char(2)
    - Hex 0000 = Use the default system value
    - Hex 0008 = Transfer data between main storage and auxiliary in blocks of 4K.
    - Hex 0010 = Transfer data between main storage and auxiliary in blocks of 8K.
    - Hex 0020 = Transfer data between main storage and auxiliary in blocks of 16K.

  - Hex 0040 = Transfer data between main storage and auxiliary in blocks of 32K.
  - Hex 0080 = Transfer data between main storage and auxiliary in blocks of 64K.
  - Hex 0100 = Transfer data between main storage and auxiliary in blocks of 128K.
- Allow exchange operations — Char(1)
  - Hex 00 = Use the default system value
  - Hex C5 = Allow exchange operations
  - Hex D5 = Disable exchange operations
  - Hex D9 = Indicate that objects are good candidates for replacement
- Handling of requests to transfer object to auxiliary storage — Char(1)
  - Hex 00 = Use the default system value
  - Hex D5 = Use the system page replacement algorithm
  - Hex D7 = Purge the objects from main storage
  - Hex D9 = Indicate the objects are good candidates for replacement
  - Hex E6 = Write the objects to auxiliary storage
- Reserved (binary 0) — Char(4)

• Reserved (binary 0) — Char(32)

Current number of pools is a user-specified value for the number of storage pools the user wishes to utilize. These are assumed to be numbered from 1 to the number specified. This number is fixed by the machine to be equal to the maximum number of pools.

Type of pool tuning determines what the system is doing to tune the performance of a storage pool.

When *tuning is not being done for a pool* (hex 00), the system tries to minimize the amount of main storage that is used by each of the jobs in the system independent of the amount of main storage that exists in a pool. The values returned for nondatabase objects and database objects by class will be all zeros to represent that the default values are being used.

If *static tuning* is being done. (hex 10), the system will use the values specified for pool information to determine the amount of data to transfer to main storage and auxiliary storage. The values returned for database objects by class and nondatabase objects will be the values previously specified on the MODRMC instructions.

When *dynamic tuning of transfers to main storage* is being done (hex 20), the system bases the amount of data to transfer to main storage based on the demand for storage in the storage pool, the size of the pool, the number of active users in the pool and other performance attributes. The values returned for database objects by class and nondatabase objects is the current value being used by the system to handle the objects.

When *dynamic tuning of transfers to main storage and auxiliary storage* is being done (hex 30), the system bases the amount of data to transfer to main storage and to auxiliary storage based on the demand for storage in the storage pool, the size of the pool, the number of active users in the pool and other performance attributes. The values returned for database objects by class and nondatabase objects is the current value being used by the system to handle the objects.

When tuning is requested (hex 10, 20 or 30), the system periodically categorizes database objects into four different performance classes. The class are:

Class 1 Object access appears to be very random - a disk access is required for nearly each record that is accessed

Class 2 Some locality of reference detected, several records are being accessed per disk access

Class 3 High locality of reference detected, object is being processed in a sequential manner. references are highly clustered, large portions of the object are resident in memory.

Class 4 See following explanation.

The class of a database object is adjusted if the objects size is small in comparision to the available storage in the storage pool. This class adjustment involves adding 1 to the class number, so a class 3 database object (as defined above) would be treated as a class 4 if it is small in comparision to the available storage in the storage pool.

Reference information for determining a object's class is collected periodically and by storage pool so an object's class will vary over time and by storage pool.

Change page handling effects when the system will write changed pages to auxiliary storage. When the *system page replacement algorithm* (hex 00) is specified as the change page handing mechanism, the system will transfer changed pages to auxiliary storage when:

- Explictly requested to transfer the page (for example, Set Access State (SETACST) instruction)
- There is a demand for pages in the pool

When the *periodically transfer changed pages* option (hex 10) is specified as the change page handing mechanism, the system will transfer changed pages to auxiliary storage when:

- Explictly requested to transfer the page (for example, Set Access State (SETACST) instruction)
- There is a demand for pages in the pool
- Periodically look for changed pages in a pool and transfer the changed pages to auxiliary storage

Blocking factor determines how much data should be brought into main storage when the objects is needed in main storage.

Allow exchange operations controls which method the system should use to find main storage to hold data. With the *exchange method* (hex C5), the system uses the page frames associated with a specific object to satisfy the request. If *exchange operations are disabled* (hex D5), the system will use the normal page replacement algorithm to find page frames for the request. If *objects should be treated as good candidates for replacement* (hex D9), the system makes the page frames associated with the object being exchanged a good replacement candidate but uses the normal page replacement algorithm to find page frames for the request.

Handling of requests to transfer objects to auxiliary storage determines when the data is transferred to auxiliary storage and when the page frames containing the object are available to contain other data. If *purging is active* (hex D7) and a request is made to purge the object to auxiliary storage, the system will immediately schedule the request to transfer the data and when the transfer is completed, the page frames containing the data just written will be made available to hold other objects. If *writing is active* (hex E6) and a request is made to purge the object to auxiliary storage, the system will immediately schedule the request to transfer the data and the page frames are not made good candidates to be reused. If *objects are good candiates for replacement* (hex D9), the objects are likely to be removed from main storage by transfering the objects to auxiliary storage when the system needs to transfer other objects into main storage. If the *system page replacement algorithm* is used (hex D5), the system decides when the object should be transferred from main storage to auxiliary storage.

Exceptions

Exception		Operands 1	2	Other
06	Addressing			

Exception		Operands 1	Operands 2	Other
	01 space addressing violation	X	X	
	02 boundary alignment	X	X	
	03 range	X	X	
	06 optimized addressability invalid	X	X	
08	Argument/parameter			
	01 parameter reference violation	X	X	
10	Damage encountered			
	04 system object damage state	X		X
	44 partial system object damage	X	X	X
1C	Machine-dependent exception			
	03 machine storage limit exceeded			X
20	Machine support			
	02 machine check			X
	03 function check			X
22	Object access			
	01 object not found	X	X	
	02 object destroyed	X	X	
	03 object suspended	X	X	
	08 object compressed			X
24	Pointer specification			
	01 pointer does not exist	X	X	
	02 pointer type invalid	X	X	
2E	Resource control limit			
	01 user profile storage limit exceeded			X
32	Scalar specification			
	02 scalar attribute invalid		X	
	03 scalar value invalid	X		
36	Space management			
	01 space extension/truncation			X
38	Template specification			
	03 materialization length exception	X		

Set Access State (SETACST)

Op Code (Hex)	Operand 1
0341	Access state template

Operand 1: Space pointer.

ILE access

```
SETACST (
    access_state : space pointer
)
```

Description: The instruction specifies the access state (which specifies the desired speed of access) that the issuing process has for a set of objects or subobject elements in the execution interval following the execution of the instruction. The specification of an access state for an object momentarily preempts the machine's normal management of an object.

Note: This instruction should be used with caution when the pointer to object entry in the template below points to a process space (i.e. static storage, automatic storage, and heap space storage). These process spaces may be shared by other programs in the activation group, so explicit access management may affect those other programs.

The Set Access State instruction template must be aligned on a 16-byte boundary. The format is:

- Number of objects to be acted upon — Bin(4)
- Reserved (binary 0) — Char(12)
- Access state specifications (repeated as many times as necessary) — Char(32)
  - Pointer to object whose access state is to be changed — Space pointer or system pointer
  - Access state code — Char(1)
  - Reserved (binary 0) — Char(3)
  - Access state parameter — Char(12)
    - Access pool ID — Char(4)
    - Space length — Bin(4)
    - Operational object size — Bin(4)

    Note: This value is returned for some of the access state code options.

The **number of objects** field specifies how many objects are potential candidates for access state modification. An **access state specification** is included for each object to be acted upon.

The **pointer to object** field identifies the object or space which is to be acted upon. For the space associated with a system object, the space pointer may address any byte in the space. This pointer is followed by parameters that define in detail the action to be applied to the object.

The **access state code** designates the desired access state. The allowed values are as follows:

Access State Code (Hex)	Function and Required Parameter
00	No operations are performed.

Access State Code (Hex)	Function and Required Parameter
01	Associated object is moved into main storage (if not already there) synchronously with the execution of the instruction.
02	Associated object is moved into main storage (if not already there) asynchronously with the execution of the instruction.
03	Associated object is placed in main storage without regard to the current contents of the object. This causes access to secondary storage to be reduced or eliminated. For this access state code, a space pointer must be provided.
04	Associated object is removed from main storage in a manner which reduces or eliminates access to secondary storage. Content of the object is unpredictable after this operation. For this access state code, a space pointer must be provided.
10	The object is synchronously ensured (changes written to auxiliary storage) and then removed from main storage. This option returns a number in the *operational object size* field. The unit assumed is the machine minimum transfer size (page size). The value returned is the total size of the opertational parts of the object examined/processed, including the associated space (if there is one). **Note:** This number is not the number of pages written or removed, but rather, is the total size of the object being processed. Some, all or none of the object may be in mainstore prior to the execution of the instruction. The *space length* field must be zero for this operation. The entire associated space, if any, will be processed with the rest of the object's storage. The *access pool ID* field is ignored for this operation. The associated pointer to the object must be a system pointer.
18	This operation essentially combines the functions of a 10 code followed by asynchronously bringing the operational parts of the object into main storage. The object is brought into the main storage pool identified by the *access pool ID* field). **Note:** Because this function first removes the object from main storage and then brings it into main storage, this can be used to "move" an object from one main storage pool to another. This option returns a number in the *operational object size* field. The unit assumed is the machine minimum transfer size (page size). The value returned is the total size of the object processed. **Note:** If this value is larger than the size of the main storage pool being used, unpredictable parts of the object will be resident in the main storage pool following processing. A a preceding access code of 40 is ignored for this operation. The *space length* field must be zero for this operation. The entire associated space, if any, will be processed with the rest of the object's storage. The *access pool ID* field must be specified for this access code (it must be value 1 though 16, decimal) . The associated pointer to the object must be a system pointer.
20	Associated object attributes are moved into main storage synchronous with the instruction's execution. The associated attributes are the attributes that are common to all system objects. The associated pointer to object must be a resolved system pointer. The "space length" field is ignored for this access code.
21	Associated object attributes are moved into main storage asynchronous with the instruction's execution. The associated attributes are the attributes that are common to all system objects. The associated pointer to object must be a resolved system pointer. The "space length" field is ignored for this access code.

Access State Code (Hex)	Function and Required Parameter
22	Common associated object attributes plus some specified amount of object-specific attributes are moved into main storage synchronous with the instruction's execution. The common associated attributes are the attributes that are common to all system objects. The object-specific attributes are attributes that vary from one object type to another. The amount of these attributes brought into main storage is controlled by the *space length* field. **Note:** This use of *space length* is not consistent with the name of the field. For this code, the *space length* field does not control the size of any associated space processing, it controls the length of object-specific attributes processed. The *space length* field works in the following manner: it specifies the amount of storage above and beyond the common ojbect attributes which will be synchronously brought into storage. Therefore, a *space length* of 0 is valid, and results in an operation identical to access code 20. The associated pointer to object must be a resolved system pointer.
23	Common associated object attributes plus some specified amount of object-specific attributes are moved into main storage asynchronous with the instruction's execution. The common associated attributes are the attributes that are common to all system objects. The object-specific attributes are attributes that vary from one object type to another. The amount of these attributes brought into main storage is controlled by the *space length* field. **Note:** This use of *space length* is not consistent with the name of the field. For this code, the *space length* field does not control the size of any associated space processing, it controls the length of object-specific attributes processed. The *space length* field works in the following manner: it specifies the amount of storage above and beyond the common ojbect attributes which will be asynchronously brought into storage. Therefore, a *space length* of 0 is valid, and results in an operation identical to access code 21. The associated pointer to object must be a resolved system pointer.
30	The associated space of the object is moved into main storage (if not already there) synchronously with the execution of the instruction. The *space length* field is honored for this operation. The associated pointer to the object must be a system pointer.
31	The associated space of the object is moved into main storage (if not already there) asynchronously with the execution of the instruction. The *space length* field is honored for this operation. The associated pointer to the object must be a system pointer.
40	Perform no operation on the associated object. The main storage occupied by this object is to be used, if possible, to satisfy the request in the next access state specification entry. Either a space or system pointer may be provided for this access state code.
41	Wait for any previously issued but incomplete hex 81 or hex 91' access state code operations to complete. This includes all previous hex 81 and hex 91 operations that may have been performed on previous Set Access State instructions within the current process as well as those that may have been issued in previous access state specification entries in the current instruction. The pointer is ignored for this access state code entry.
80	Object should be written and it is not needed in main storage by issuing process. Object is written to nonvolatile storage synchronously with the execution of the instruction. Any main storage that the object occupied is then marked as to make it quickly available for replacement.
81	Object should be written and it is not needed in main storage by issuing process. Object is written to nonvolatile storage asynchronously with the execution of the instruction. Any main storage that the object occupied is then marked as to make it quickly available for replacement. If desired, the process can synchronize with any outstanding hex 81 access state operations by issuing a hex 41 access state operations either within the current instruction or during a subsequent Set Access State instruction.

Access State Code (Hex)	Function and Required Parameter
90	Associated object must be insured, but is still needed in main storage. Object is written to nonvolatile storage synchronously with the execution of the instruction. Unlike access state codes hex 80 and hex 81, this access state code does not mark any main storage occupied by the object as to make it quickly available for replacement.
91	Associated object must be insured, but is still needed in main storage. Object is written to nonvolatile storage asynchronously with the execution of the instruction. Unlike access state codes hex 80 and hex 81, this access state code does not mark any main storage occupied by the object as to make it quickly available for replacement. If desired, the process can synchronize with any outstanding hex 91 access state operations by issuing a hex 41 access state operations either within the current instruction or during a subsequent Set Access State instruction.

Access state codes hex 03 and hex 04 may be used for spaces only. The pointer to the object in the access state specification must be a space pointer. Otherwise, the *pointer type invalid* (hex 2402) exception is signaled.

Access state code hex 40 may be used in conjunction with access state codes hex 01, hex 02, or hex 03. The access state specification entry with access state code hex 40 must immediately precede the access state specification entry with access state code hex 01, hex 02, or hex 03 with which it is to be combined. The pointer to the object in both entries must be a space pointer. Otherwise, the *pointer type invalid* (hex 2402) exception is signaled. The *access state parameter* field in the *access state specification entry* with code hex 40 is ignored. The *access pool ID* and the *space length* in the entry with access state code hex 01, hex 02, or hex 03 are used.

The **access pool ID** field indicates the desired main storage pool in which the object is to be placed (0-16). The storage pool ID entry is treated as a 4-byte logical binary value. When a 0 storage pool ID is specified, the storage pool associated with the issuing process is used.

The **space length** field designates the part of the space associated with the object to be operated on. If the pointer to the object entry is a system pointer, the operation begins with the first byte of the space. If the pointer to the object entry is a space pointer that specifies a location, the operation proceeds for the number of storage units that are designated. No exception is signaled when the number of referenced bytes of the space are not allocated. When operations on objects are designated by system pointers, this operation is performed in addition to the access state modification of the object. This entry is ignored for *access state codes* hex 20 and hex 21. This entry will be truncated to a maximum of 65536 for *access state codes* immediately following access state code 40.

The **operational object size** field is a value which is ignored upon input to the instruction and is set by the instruction for access codes 10 and 18. It represents, in units of minimum machine transfer size, the total size of the object which could/did participate in the operation. The parts of an object which are considered "operational" are decided by the machine and does include the associated space, if any.

Authorization Required

- Retrieve
  - Contexts referenced for address resolution

Lock Enforcement

- Materialize
  - Contexts referenced for address resolution

Exceptions

Exception		Operands 1	Other
04	Access state		
	01 access state specification invalid	X	
06	Addressing		
	01 space addressing violation	X	
	02 boundary alignment	X	
	03 range	X	
	06 optimized addressability invalid	X	
08	Argument/parameter		
	01 parameter reference violation	X	
0A	Authorization		
	01 unauthorized for operation	X	
10	Damage encountered		
	04 system object damage state	X	
	05 authority verification terminated due to damaged object		X
	44 partial system object damage	X	X
1A	Lock state		
	01 invalid lock state	X	
1C	Machine-dependent exception		
	03 machine storage limit exceeded		X
20	Machine support		
	02 machine check		X
	03 function check		X
22	Object access		
	01 object not found	X	
	02 object destroyed	X	
	03 object suspended	X	
	07 authority verification terminated due to destroyed object		X
	08 object compressed		X
24	Pointer specification		
	01 pointer does not exist	X	
	02 pointer type invalid	X	
	03 pointer addressing invalid object	X	
	04 pointer not resolved	X	
2E	Resource control limit		
	01 user profile storage limit exceeded		X
36	Space management		
	01 space extension/truncation		X

Exception		Operands 1	Other
38	Template specification		
	01 template value invalid	X	

Chapter 20. Dump Space Management Instructions

This chapter describes all the instructions used for dump space management. These instructions are arranged in alphabetical order. For an alphabetic summary of all the instructions, see Appendix A, "Instruction Summary."

Materialize Dump Space (MATDMPS) 20-3

© Copyright IBM Corp. 1991, 1993

Materialize Dump Space (MATDMPS)

Op Code (Hex)	Operand 1	Operand 2
04DA	Receiver	Dump space

Operand 1: Space pointer.

Operand 2: System pointer.

ILE access

```
MATDMPS(
      receiver    : space pointer
  var dump_space : system pointer
)
```

Description: The current attributes of the dump space specified by operand 2 are materialized into the receiver specified by operand 1.

The template identified by operand 1 must be 16-byte aligned in the space. The format of the materialization is as follows:

• Materialization size specification	Char(8)
– Number of bytes provided for materialization	Bin(4)
– Number of bytes available for materialization (always 128 for this instruction)	Bin(4)
• Object identification	Char(32)
– Object type	Char(1)
– Object subtype	Char(1)
– Object name	Char(30)
• Object creation options	Char(4)
– Existence attributes 0 = Temporary 1 = Permanent	Bit 0
– Space attribute 0 = Fixed length 1 = Variable length	Bit 1
– Context 0 = Addressability not in context 1 = Addressability in context	Bit 2
– Reserved (binary 0)	Bit 3-12
– Initialize space	Bit 13
– Reserved (binary 0)	Bit 14-31
• Recovery options	Char(4)
• Size of space	Bin(4)
• Initial value of space	Char(1)

• Performance class	Char(4)
• Reserved	Char(7)
• Context	System pointer
• Reserved	Char(16)
• Dump Space size	Char(4)
• Dump data size	Char(4)
• Dump data size limit	Char(4)
• Reserved	Char(20)

The first 4 bytes of the materialization identify the total **number of bytes provided** for use by the instruction. This value is supplied as input to the instruction and is not modified by the instruction. A value of less then eight causes the *materialization length* (hex 3803) exception to be signaled.

The second 4 bytes of the materialization identify the total *number of bytes available* to be materialized. The instruction materializes as many bytes as can be contained in the area specified as the receiver. If the byte area identified by the receiver is greater than that required to contain the information requested, then the excess bytes are unchanged. No exceptions are signaled in the event that the receiver contains insufficient area for the materialization, other than the *materialization length* (hex 3803) exception described previously.

The **dump space size** field is set with the current size value for the number of 512-byte blocks of space allocated for storage of dump data within the dump space.

The **dump data size** field is set with the current size value for the number of 512-byte blocks of dump data contained in the dump space. This value specifies the number of blocks from the start of the dump space through the block of dump data which has been placed into the dump space at the largest dump space offset value. A value of zero indicates that the dump space currently contains no dump data.

The **dump data size limit** field is set with the current size limit for the number of 512-byte blocks of dump data which may be stored in the dump space. A value of zero indicates that no explicit limitation is placed on the amount of dump data which may be stored in the dump space. The machine implicitly places a limit on the maximum size of a dump space. This value of this limitation is dependent upon the specific implementation of the machine.

Authorization Required

- Operational
  - Operand 2
- Retrieve
  - Contexts referenced for address resolution

Lock Enforcement

- Materialize
  - Operand 2
  - Contexts referenced for address resolution

Exceptions

Exception		Operands 1	2	Other
06	Addressing			
	01 space addressing violation	X	X	
	02 boundary alignment violation	X	X	
	03 range	X	X	
	06 optimized addressability invalid	X	X	
08	Argument/parameter			
	01 parameter reference violation	X	X	
0A	Authorization			
	01 unauthorized for operation		X	
10	Damage encountered			
	04 system object damage state		X	X
	05 authority verification terminated due to damaged object			X
	44 partial system object damage			X
1A	Lock state			
	01 invalid lock state		X	
1C	Machine-dependent exception			
	03 machine storage limit exceeded			X
20	Machine support			
	02 machine check			X
	03 function check			X
22	Object access			
	01 object not found	X	X	
	02 object destroyed	X	X	
	03 object suspended	X	X	
	07 authority verification terminated due to destroyed object			X
	08 object compressed			X
24	Pointer specification			
	01 pointer does not exist	X	X	
	02 pointer type invalid	X	X	
	03 pointer addressing invalid object		X	
2E	Resource control limit			
	01 user profile storage limit exceeded			X
32	Scalar specification			
	01 scalar type invalid	X	X	
36	Space management			
	01 space extension/truncation			X
38	Template specification			
	01 template value invalid	X		

Exception	Operands 1	2	Other
03 materialization length exception	X		

Chapter 21. Machine Observation Instructions

This chapter describes all instructions used for machine observation. These instructions are arranged alphabetically. For an alphabetic summary of all the instructions, see Appendix A, "Instruction Summary."

| Find Relative Invocation Number (FNDRINVN) 21-3
Materialize Instruction Attributes (MATINAT) 21-8
Materialize Invocation (MATINV) 21-14
| Materialize Invocation Attributes (MATINVAT) 21-18
Materialize Invocation Entry (MATINVE) 21-28
Materialize Invocation Stack (MATINVS) 21-32
Materialize Pointer (MATPTR) 21-37
Materialize Pointer Locations (MATPTRL) 21-46
Materialize System Object (MATSOBJ) 21-48

© Copyright IBM Corp. 1991, 1993

Find Relative Invocation Number (FNDRINVN)

Op Code (Hex)	Operand 1	Operand 2	Operand 3
0543	Relative invocation number	Search range	Search criterion template

Operand 1: Signed binary(4) variable scalar

Operand 2: Character(48) scalar (fixed length) or null

Operand 3: Space pointer

ILE access

```
FNDRINVN (
 var relative_invocation_number : signed binary;
 var search_range               : aggregate; OR
                                  null operand;
     search_criterion_template  : space pointer
)
```

Description: The invocations identified by operand 2 are searched in the order specified by operand 2 until an invocation is found which satisfies the search criterion specified in the operand 3 template. The identity of the first invocation (in search order) to satisfy the search criterion is returned in operand 1. If no invocation in the specified range satisfies the search criterion, then either an exception is signaled, or a value of zero is returned in operand 1, depending on the modifiers specified in the operand 3 template.

Operand 1 is returned as a signed binary(4) value identifying the first invocation found that satisfies the specified search criterion. It is specified relative to the starting invocation identified by operand 2. A positive number indicates a displacement in the direction of newer invocations, while a negative indicates a displacement in the direction of older invocations. A zero value indicates that no invocation in the specified range matched the specified criterion. Operand 1 is not modified in the event that the instruction terminates with an exception.

Note that a modifier in the operand 3 template determines if the starting invocation identified by operand 2 is to be skipped. If the starting invocation is skipped during the search then a result of zero in operand 1 always indicates failure to find an invocation that satisfies the criterion. If the starting invocation is not skipped, then a failure to find an invocation that satisfies the criterion results in an exception.

Operand 2 identifies the starting invocation and the range of the search. If operand 2 is specified as a null operand, then operand 2 is assumed to identify a range starting with the current invocation and proceeding through all existing older invocations.

Operand 3 is a space pointer to a template that identifies the search criterion and search modifiers for the find operation.

Operand 2: The value specified by *operand 2* identifies the range of invocations to be searched. This operand can be null (which indicates the range which starts with the current invocation and proceeds through all existing older invocations), or it can contain either an invocation pointer to an invocation or a null pointer (which indicates a range starting with the current invocation).

Operand 2 has the following format:

• Starting invocation offset	Bin(4)
• Originating invocation offset (ignored)	Bin(4)
• Invocation range	Bin(4)
• Reserved (binary 0)	Char(4)
• Starting invocation pointer	Invocation pointer
• Reserved (binary 0)	Char(16)

If a non-null pointer is specified for *starting invocation pointer*, then *operand 2* must be 16-byte aligned in the space.

Terminology:

Requesting invocation
The invocation executing the **FNDRINVN** instruction. Note that, in many cases, this invocation belongs to a system or language run-time procedure/program, and the instruction is actually being executed on behalf of another procedure or program.

Starting invocation
The invocation which serves as the starting point for the search.

Field descriptions

Starting invocation offset
A signed numerical value indicating an invocation relative to the invocation located by the *starting invocation pointer*. A value of zero denotes the invocation addressed by the *starting invocation pointer*, with increasingly positive numbers denoting increasingly later invocations in the stack, and increasingly negative numbers denoting increasingly earlier invocations in the stack.

If the *starting invocation pointer* is valid or null, but the invocation identified by this offset does not exist in the stack, an *invocation offset outside of current stack* (hex 2C1A) exception will be signaled.

Originating invocation offset
This field is used by other instructions but is ignored by **FNDRINVN**.

Invocation range
Invocation range is a signed numerical value which specifies the direction of the search and the maximum number of invocations to be examined. The magnitude of *invocation range* specifies the maximum number of invocations to be searched *exclusive* of the starting invocation. It is not an error if this magnitude is greater than the number of existing invocations in the specified direction. If the sign of *invocation range* is positive (and non-zero), the search is performed in the direction of newer invocations, while if the sign is negative, the search is performed in the direction of older invocations.

Note that the *bypass starting invocation* modifier in *operand 3* affects how the starting invocation is treated. If this modifier is false, then the starting invocation is the first invocation examined. If *invocation range* is zero in this case then *only* the starting invocation is examined. If, on the other hand, *bypass starting invocation* is true, then the starting invocation *does not* participate in the search, and, if *invocation range* is zero, no invocations are searched and a value of zero is returned for operand 1.

Starting invocation pointer
An invocation pointer to an invocation. If null, then the current invocation is indicated. If not null, then *operand 2* must be 16-byte aligned in the space.

If the pointer identifies an invocation in another process, a *process object access invalid* (hex 2C11) exception will be signaled. If the invocation identified by this pointer does not exist in the stack, an *object destroyed* (hex 2202) exception will be signaled.

Usage note: In cases where *starting invocation pointer* is null, *operand 2* may be a constant.

Operand 3: The **search criterion template** identified by *operand 3* must be aligned on a 16-byte boundary. The template is a 32-byte value with the following format:

• Reserved (binary 0)	Char(8)
• Search option	Bin(4)
• Search modifiers	Char(4)
– Bypass starting invocation	Bit 0

0 = The starting invocation identified by *operand 2* is the first invocation tested. A *invocation not found* (hex 1E02) exception is signaled if the search criterion is not satisfied.
1 = The starting invocation identified by *operand 2* is skipped and no exception is signaled if the search criterion is not satisfied.

– Compare for mismatch	Bit 1

0 = The instruction identifies the first invocation (in specified search order) which *matches* the specified search criterion
1 = The instruction identifies the first invocation (in specified search order) which *does not match* the specified search criterion

– Reserved (binary 0)	Bit 1
• Search argument	Char(16)

Search option
Specifies the invocation attribute to be examined:

1 Routine type. *Search argument* is a one-byte routine type, left aligned. Allowed *search argument* values are:

Hex 01 OPM Program

Hex 02 NPM Program Entry Procedure (PEP)

Hex 03 NPM Procedure

2 Invocation type. *Search argument* is a one-byte invocation type, left aligned. Allowed *search argument* values are:

Hex 01 Call external

Hex 02 Transfer control

Hex 03 Event handler

Hex 04 External exception handler (for OPM program)

Hex 05 Initial program in process problem state

Hex 06 Initial program in process initiation state

Hex 07 Initial program in process termination state

Hex 08 Invocation exit (for OPM program)

Hex 09 Return or return/XCTL trap handler

Hex 0A Call program

Hex 0B Cancel handler (NPM only)

Hex 0C Exception handler (NPM only)

Hex 0D Call bound procedure/call with procedure pointer

Hex 0E Process Default Exception Handler

3 Invocation status. *Search argument* consists of two four-byte fields, left aligned. The invocation status of each examined invocation is ANDed with the first field and then compared to the second field.

4 Invocation mark. *Search argument* is a four-byte invocation mark, left aligned. If the search is in the direction of older invocations, the result identifies the first invocation found with an invocation mark less than or equal to the search argument. If the search is in the direction of newer invocations, the result identifies the first invocation found with an invocation mark greater than or equal to the search argument. If *invocation range* is zero, then the search is satisfied only if the invocation mark of the *starting invocation* exactly matches the search argument, and this can occur only if *bypass starting invocation* is false.

For this option *compare for mismatch* is ignored.

5 Activation mark. *Search argument* is a four-byte activation mark, left aligned. The activation mark of the program or module activation corresponding to each examined invocation is compared to *search argument.* Invocations with no activation (ie, the invocations of OPM reentrant programs, and the invocation stack base entry) are considered to have an activation mark of binary zero.

6 Activation group mark. *Search argument* is a four-byte activation group mark, left aligned. The activation group mark of each examined invocation is compared to *search argument.*

7 Program pointer. *Search argument* is a system pointer to a program. The program corresponding to each examined invocation is compared to the program identified by the pointer.

Bypass starting invocation

If *bypass starting invocation* is false, then the starting invocation specified by *operand 2* is the first invocation examined. In this case, if the *invocation range* of *operand 2* is exhausted without satisfying the search criterion then a *template value invalid* (hex 3801) exception is signaled, with the *search argument* field of *operand 3* identified as the erroneous field.

If *bypass starting invocation* is true, then the starting invocation specified by *operand 2* is skipped, and a failure to satisfy the search criterion is indicated by returning a binary zero value in *operand 1*.

Compare for mismatch

If *compare for mismatch* is false, then the search criterion is satisfied when an invocation is found whose attribute *matches* the *search argument*.
If *compare for mismatch* is true, however, then the search criterion is satisfied when an invocation is found whose attribute *does not match* the *search argument*.

Search argument

A value of between one and 16 bytes as described above. Unused bytes are ignored.

Authorization Required: None

Lock Enforcement: None

Exceptions

Exception		Operands 1	2	3	Other
06	Addressing				
	01 space addressing violation	X	X	X	
	02 boundary alignment		X	X	
	03 range	X	X	X	

Exception		Operands 1	2	3	Other
	06 optimized addressability invalid	X	X	X	
08	Argument/parameter				
	01 parameter reference violation	X	X	X	
10	Damage encountered				
	04 system object damage state				X
	44 partial system object damage				X
16	Exception management				
	03 invalid invocation addrsss		X		
1C	Machine-dependent exception				
	03 machine storage limit exceeded				X
1E	Machine observation				
	02 invocation not found		X		
20	Machine support				
	02 machine check				X
	03 function check				X
22	Object access				
	02 object destroyed	X	X	X	
	03 object suspended	X	X	X	
	08 object compressed				X
24	Pointer specification				
	01 pointer does not exist	X	X	X	
	02 pointer type invalid	X	X	X	
2C	Program execution				
	11 Process object access invalid		X		
	1A Invocation offset outside range of current stack		X		
2E	Resource control limit				
	01 user profile storage limit exceeded				X
32	Scalar specification				
	03 scalar value invalid		X		
36	Space management				
	01 space extension/truncation				X
38	Template specification				
	01 template value invalid			X	

Materialize Instruction Attributes (MATINAT)

Op Code (Hex)	Operand 1	Operand 2
0526	Receiver	Selection information

Operand 1: Space pointer.

Operand 2: Character scalar (fixed length).

Description: This instruction materializes the attributes of the instruction that are selected in operand 2 and places them in the receiver (operand 1).

Operand 2 is a 16-byte template. Only the first 16 bytes are used. Any excess bytes are ignored. Operand 2 has the following format:

- Selection template — Char(16)
  - Invocation number — Bin(2)
  - Instruction number — Bin(4)
  - Reserved (binary 0) — Char(10)

The **invocation number** is a specific identifier for the target invocation, in the process, that is to be materialized. This program must be observable or the *program not observable* (hex 1E01) exception is signaled.

The **instruction number** specifies the instruction in the specified program invocation that is to be materialized.

Operand 1 is a space pointer that addresses a 16-byte aligned template where the materialized data is placed. The format of the data is as follows:

- Materialization size specification — Char(8)
  - Number of bytes provided by the user — Bin(4)
  - Number of bytes available to be materialization — Bin(4)
- Object identification — Char(32)
  - Program type — Char(1)
  - Program subtype — Char(1)
  - Program name — Char(30)
- Offset to instruction attributes — Bin(4)
- Reserved (binary 0) — Char(8)
- Instruction attributes — Char(*)
  - Instruction type — Char(2)
    - Instruction version — Bits 0-3

      Hex 0000 = 2-byte operand references
      Hex 0001 = 3-byte operand references
    - Reserved (binary 0) — Bits 4-15
  - Instruction length as input — Bin(2)

to Create Program

— Offset to instruction form specified as input to Create Program	Bin(4)
— Reserved (binary 0)	Char(4)
— Number of instruction operands	Bin(2)
— Operand attributes offsets	Char(*)
- An offset is materialized for each of the the operands of the instruction specifying the offset to the attributes for the operand	Bin(4)
— Instruction form specified as input to Create Program	Char(*)
- Instruction operation code	Char(2)
- Optional extender field and operand fields	Char(*)
— Operand attributes	Char(*)

A set of attributes following this format is materialized for each of the operands of the instruction. Compound operand references result in materialization of only one set of attributes for the operand which describe the substring or array element as is appropriate. See the specific format described below for each operand type.

- Operand type	Bin(2)

1 = Data object
2 = Constant data object
3 = Instruction number reference
4 = Argument list
5 = Exception description
6 = Null operand
7 = Space pointer machine object

- Operand specific attributes	Char(*)

See descriptions below for detailed formats. Nothing is provided for null operands.

• Data object	Char(32)

For a data object, the following operand attributes are materialized.

— Operand type = 1	Bin(2)
— Data object specific attributes	Char(7)
- Element type	Char(1)

Hex 00 = Binary
Hex 01 = Floating-point
Hex 02 = Zoned decimal
Hex 03 = Packed decimal
Hex 04 = Character
Hex 08 = Pointer

- Element length	Char(2)

• If binary, or character, or floating-point:

— Length	Bits 0-15

• If zoned decimal or packed decimal:

— Fractional digits	Bits 0-7
— Total digits	Bits 8-15

• If pointer:

— Length = 16 | Bits 0-15

- Array size | Bin(4)
  - If scalar, then value of 0.
  - If array, then number of elements.

- Reserved (binary 0) | Char(6)
- Data object addressability | Char(17)
  - Addressability indicator | Char(1)

    Hex 00 = Addressability was not established
    Hex 01 = Addressability was established

  - Space pointer to the object if addressability could be established | Space pointer

- Constant data object | Char(*)

  For a constant data object, the following operand attributes are materialized (immediate operands as constants, signed immediates as binary, and unsigned immediates as character).

  — Operand type = 2 | Bin(2)

  — Constant specific attributes | Char(7)

    - Element type | Char(1)
      Hex 00 = Binary
      Hex 01 = Floating-point
      Hex 02 = Zoned decimal
      Hex 03 = Packed decimal
      Hex 04 = Character
    - Element length | Char(2)
      - If binary, or character, or floating-point:
        — Length | Bits 0-15
      - If zoned decimal or packed decimal:
        — Fractional digits | Bits 0-7
        — Total digits | Bits 8-15
      - Reserved (binary 0) | Bin(4)
    - Reserved (binary 0) | Char(7)
    - Constant value | Char(*)

- Instruction references | Char(*)

  For instruction references, either through instruction definition lists or immediate operands, the following operand attributes are materialized.

  — Operand type = 3 | Bin(2)

  — Number of instruction reference elements | Bin(2)

    1 = Single instruction reference
    >1 = Instruction definition list

  — Reserved (binary 0) | Char(12)

  — Reference list | Char(*)

    The instruction number of each instruction reference is materialized in the order in which they are defined.

- Argument list | Char(*)

For an argument list, the following operand attributes are materialized.

- Operand type = 4 — Bin(2)
- Argument list specific attributes — Char(4)
  - Actual number of list entries — Bin(2)
  - Maximum number of list entries — Bin(2)
- Reserved (binary 0) — Char(10)
- Addressability to list entries — Char(*)

  Space pointer to each list entry for the number of actual list entries. A value of all zeros is materialized if addressability could not be established. — Space pointer

• Exception description — Char(48)

For an exception description, the following operand attributes are materialized.

- Operand type = 5 — Bin(2)
- Reserved (binary 0) — Char(10
- Control flags — Char(2)
  - Exception handling action — Bits 0-2

    000 = Ignore occurrence of exception and continue processing
    001 = Disabled exception description
    010 = Continue search for an exception description by resignaling the exception to the immediately preceding invocation
    100 = Defer handling
    101 = Pass control to the specified exception handler

  - Reserved (binary 0) — Bits 3-15
- Compare value length — Bin(2)
- Compare value — Char(32)

• Space pointer machine object — Char(32)

For a space pointer machine object, the following operand attributes are materialized.

- Operand type = 7 — Bin(2)
- Reserved (binary 0) — Char(13)
- Pointer addressability — Char(17)
  - Pointer value indicator — Char(1)

    Hex 00 = Addressability value is not valid
    Hex 01 = Addressability value is valid

- Space pointer data object containing the space pointer machine object value if addressability value is valid. — Space pointer

The first 4 bytes of the materialization identify the total **number of bytes provided** for use by the instruction. This value is supplied as input to the instruction and is not modified by the instruction. A value of less than 8 causes the *mateialization length* (hex 3803) exception to be signaled.

The second 4 bytes of the materialization identify the total **number of bytes available** to be materialized. The instruction materializes as many bytes as can be contained in the area specified as the receiver. If the byte area identified by the receiver is greater than that required to contain the information requested, then excess bytes are unchanged.

The materialization available for an instruction depends on the execution status of the program that the instruction is in. If the program has not executed to the point of the instruction, little or no meaningful information about the instruction can be materialized. If the program executes the instruction multiple times, the materialization will vary with each execution.

No exceptions are signaled in the event that the receiver contains insufficient area for the materialization, other than the *materialization length* (hex 3803) exception described previously.

This instruction is valid only when the program to be materialized is an OPM program. If the invocation indicated by *operand 2* is an invocation of an NPM program or procedure, then an *instruction not valid for invocation type* (hex 2C1C) exception is signaled.

Exceptions

Exception		Operands 1	2	Other
06	Addressing			
	01 Space addressing violation	X	X	
	02 Boundary alignment	X	X	
	03 Range	X	X	
	06 Optimized addressability invalid	X	X	
08	Argument/Parameter			
	01 Parameter reference violation	X	X	
10	Damage Encountered			
	04 System object damage state			X
	44 Partial system object damage			X
1C	Machine-Dependent Exception			
	03 Machine storage limit exceeded			X
1E	Machine Observation			
	01 Program not observable		X	
20	Machine Support			
	02 Machine check			X
	03 Function check			X
22	Object Access			
	01 Object not found		X	
	02 Object destroyed	X	X	
	03 Object suspended	X	X	
	08 object compressed			X
24	Pointer Specification			
	01 Pointer does not exist	X	X	
	02 Pointer type invalid	X	X	
2C	Program Execution			
	1C Instruction not valid for invocation type		X	
2E	Resource Control Limit			

Exception		Operands 1	2	Other
	01 User Profile storage limit exceeded			X
32	Scalar Specification			
	01 Scalar type invalid	X	X	
	02 Scalar attributes invalid	X	X	
	03 Scalar value invalid	X	X	
36	Space Management			
	01 Space Extension/Truncation			X
38	Template Specification			
	01 Template value invalid		X	
	03 Materialization length exception	X		

Materialize Invocation (MATINV)

Op Code (Hex)	Operand 1	Operand 2
0516	Receiver	Selection information

Operand 1: Space pointer.

Operand 2: Space pointer.

ILE access

```
MATINV (
     receiver              : space pointer;
     selection_information : space pointer
)
```

Description: The attributes of the invocation selected through operand 2 are materialized into the receiver designated by operand 1.

Operand 2 is a space pointer that addresses a template that has the following format:

- Control information — Char(2)
  - Template extension — Bit 0*
    - 0 = Template extension is not present.
    - 1 = Template extension is present.
  - Invocation number — Bits 1-15
- Offset to list of parameters — Bin(4)*
- Number of parameter ODV numbers — Char(2)*
- Offset to list of exception descriptions — Bin(4)*
- Number of exception description ODV numbers* — Char(2)
- Template extension (optional) — Char(14)*
  - Offset to list of space pointer machine objects* — Bin(4)
  - Number of space pointer machine object ODV numbers* — Char(2)
  - Reserved (binary 0) — Char(8)

Note: Fields annotated with an (*) must be set to all binary zeroes if the invocation is for a bound program. The *template extension* field must also be set to zero. A *template value invalid* (hex 3801) exception is signaled if the invocation is a bound program invocation and the fields are not set to binarary zeroes.

The **offset to the list of space pointer machine objects**, **offset to the list of parameters**, and the **offset to the list of exception descriptions** are relative to the start of the operand 2 template. Each list is an array of Char(2) ODV numbers. The **number of space pointer machine object ODV numbers**, **number of parameter ODV numbers**, and **the number of exception description ODV numbers** define the sizes of the arrays.

Operand 1 is a space pointer that addresses a 16-byte aligned template into which the materialized data is placed. The format of the data is:

- Materialization size specification — Char(8)
  - Number of bytes provided by the user — Bin(4)
  - Number of bytes available for materialization — Bin(4)
- Object identification — Char(32)
  - Program type — Char(1)
  - Program subtype — Char(1)
  - Program name — Char(30)
- Trace specification — Char(2)
  - Invocation trace status — Bit 0

    0 = Not tracing new invocations
    1 = Tracing new invocations
  - Return trace — Bit 1

    0 = Not tracing returns
    1 = Tracing returns
  - Invocation trace propagation — Bit 2

    0 = Not propagating invocation trace
    1 = Propagating invocation trace
  - Return trace propagation — Bit 3

    0 = Not propagating return trace
    1 = Propagating return trace
  - Reserved (binary 0) — Bits 4-15

| The following fields are returned only for non-bound program invocations.

- Instruction number — UBin(2)
- Offset to parameter values — Bin(4)
- Offset to exception description value — Bin(4)
- Offset to space pointer machine object values — Bin(4)
  (Optional-This data is present only if the template extension is present in the selection information.)
- Space pointer machine objects — Char(*)
  (Optional-This data is present only if the template extension is present in the selection information.)
  - For each ODV number specified for a space pointer machine object, the value of the space pointer machine object is materialized as follows: — Char(32)
    - Reserved (binary 0) — Char(15)
    - Pointer value indicator — Char(1)

      00 = Addressability value is not valid
      01 = Addressability value is valid
    - Space pointer data object containing the space pointer machine object value if addressability value is valid. — Space pointer
- Parameters — Char(*)
  - For each parameter ODT number specified, — Space pointer

the address of the parameter data is materialized (If no parameter ODT numbers are materialized, this parameter is binary 0.)

- Exception description — Char(*)
  - For each exception description ODT number specified, the following is materialized: — Char(36)
  - Control flags — Char(2)
    - Exception handling action — Bits 0-2

      000 = Ignore occurrence of exception and continue processing
      001 = Disabled exception description
      010 = Continue search for an exception description by resignaling the exception to the immediately preceding invocation
      100 = Defer handling
      101 = Pass control to the specified exception handler
    - Reserved (binary 0) — Bits 3-15
  - Compare value length — Bin(2)
  - Compare value — Char(32)

The first 4 bytes of the materialization identify the total **number of bytes provided** for use by the instruction. This value is supplied as input to the instruction and is not modified by the instruction. A value of less than 8 causes the *materialization length* (hex 3803) exception to be signaled.

The second 4 bytes of the materialization identify the total **number of bytes available** to be materialized. The instruction materializes as many bytes as can be contained in the area specified as the receiver. If the byte area identified by the receiver is greater than that required to contain the information requested, then excess bytes are unchanged.

No exceptions (other than the *materialization length* (hex 3803) exception) are signaled in the event that the receiver contains insufficient area for the materialization.

The instruction number returned depends on how control was passed from the invocation:

Exit Type	Instruction Number
Call External	Locates the Call External instruction
Event	Locates the next instruction to execute
Exception	Locates the instruction that caused the exception

The space pointers that address parameter values are returned in the same order as the corresponding ODT numbers in the input array. The same is true for the exception description values.

If the *offset to the list of parameters* or the *number of parameter ODT numbers* is 0, no parameters are returned and the *offset to parameters* value is 0. If any parameters are returned, they are 16-byte aligned. If the *offset to list of exception descriptions* or the *number of exception description ODT numbers* is 0, no exception descriptions are returned and the *offset to exception description values* are 0.

Exceptions

Exception		Operands 1	Operands 2	Other
06	Addressing			
	01 space addressing violation	X	X	

Exception		Operands 1	Operands 2	Other
	02 boundary alignment	X	X	
	03 range	X	X	
	06 optimized addressability invalid	X	X	
08	Argument/parameter			
	01 parameter reference violation	X	X	
10	Damage encountered			
	04 system object damage state	X	X	X
	44 partial system object damage	X	X	X
1C	Machine-dependent exception			
	03 machine storage limit exceeded		X	
1E	Machine observation			
	01 program not observable			X
20	Machine support			
	02 machine check			X
	03 function check			X
22	Object access			
	01 object not found		X	
	02 object destroyed	X	X	
	03 object suspended	X	X	
	08 object compressed			X
24	Pointer specification			
	01 pointer does not exist	X	X	
	02 pointer type invalid	X	X	
2E	Resource control limit			
	01 user profile storage limit exceeded			X
32	Scalar specification			
	01 scalar type invalid	X	X	
	02 scalar attributes invalid	X	X	
36	Space management			
	01 space extension/truncation			X
38	Template specification			
	01 template value invalid		X	
	03 materialization length exception	X		

Materialize Invocation Attributes (MATINVAT)

Op Code (Hex)	Operand 1	Operand 2	Operand 3
0533	Receiver	Invocation identification	Attribute selection template

Operand 1: Space pointer.

Operand 2: Character(48) scalar (fixed length) or null.

Operand 3: Space pointer.

ILE access

```
MATINVAT (
      receiver                       : space pointer
  var invocation_identification      : aggregate; OR
                                       null operand;
      attribute_selection_template : space pointer
)
```

Description: The attributes specified by operand 3 of the invocation specified by operand 2 are materialized into the receiver specified by operand 1. In addition to specifying the attributes to be materialized, operand 3 controls how they are arranged in the operand 1 receiver.

Operand 1 is a space pointer to an area that is to receive the materialized attribute values. The format of this area is determined by the value of the attribute selection template.

Operand 2 identifies the source invocation whose attributes are to be materialized. It also identifies the originating invocation whose activation group access right to the source invocation's activation group is to be verified. If operand 2 is null, the invocation issuing the instruction is both the source invocation and the originating invocation.

Operand 3 is a space pointer to a template that selects the invocation attributes to be materialized and specifies how they are to be arranged in the receiver template.

Operand 2

The value specified by *operand 2* identifies the source and originating invocations. This operand can be null (which indicates the current invocation is to be used for the source and originating invocations) or it can contain either a invocation pointer to an invocation or a null pointer (which indicates the current invocation).

Operand 2 has the following format:

• Source invocation offset	Bin(4)
• Originating invocation offset	Bin(4)
• Invocation range (ignored)	Bin(4)
• Reserved (binary 0)	Char(4)
• Source invocation pointer	Invocation pointer
• Reserved (binary 0)	Char(16)

If a non-null pointer is specified for *source invocation pointer*, then *operand 2* must be 16-byte aligned in the space.

Terminology:

Requesting invocation
The invocation executing the **MATINVAT** instruction. Note that, in many cases, this invocation belongs to a system or language run-time procedure/program, and the instruction is actually being executed on behalf of another procedure or program.

Originating invocation
The invocation on whose behalf the instruction is being executed. It may be necessary to identify this invocation since its "activation group access rights" may need to be checked. This allows, for example, the requesting invocation to be a system state invocation with the instruction still performing an "activation group access rights" check that reflects the rights of the user.

Source invocation
The invocation whose attributes are to be materialized.

Activation group access rights
The rights that invocations executing in one activation group may have to access and modify the resources of another activation group.

Field descriptions:

Source invocation offset
A signed numerical value indicating an invocation relative to the invocation located by the source invocation pointer. A value of zero denotes the invocation addressed by the source invocation pointer, with increasingly positive numbers denoting increasingly later invocations in the stack, and increasingly negative numbers denoting increasingly earlier invocations in the stack.

If the source invocation pointer is not valid or the invocation identified by this offset does not exist in the stack, an *invocation offset outside range of current stack* (hex 2C1A) exception will be signaled.

Originating invocation offset
A signed numerical value identifying the originating invocation relative to the current invocation. Since this is an offset relative to the current invocation, only zero or negative values are allowed.

If the invocation identified by this offset does not exist in the stack, an *invocation offset outside range of current stack* (hex 2C1A) exception will be signaled.

Invocation range
This field is used by **FNDRINVN** and is ignored by this instruction.

Source invocation pointer
An *invocation pointer* to an invocation. If null, then the current invocation is indicated.

If the pointer identifies an invocation in another process, a *process object access invalid* (hex 2C11) exception will be signaled. If the invocation identified by this pointer does not exist in the stack, an *object destroyed* (hex 2202) exception will be signaled.

Activation group access rights checking: This instruction sometimes (depending on the attributes materialized) requires that activation group access rights to the activation group of the source invocation be verified. In such cases, the *originator offset* field of *operand 2* identifies the invocation whose right of access is to be checked. (That is, it identifies the invocation which is considered to have originated the request and on whose behalf the instruction is being executed.)

If *originator offset* is not zero, then the activation group of the requesting invocation must have the right to access the activation group of the invocation identified by *originator offset*. This check is made whether or not access rights to the source invocation need to be checked.

In the event that appropriate access rights are not found, an *activation group access violation* (hex 2C12) exception is signaled.

Note: The originating invocation identified by the originating invocation offset must be equal to or "newer" than the invocation identified as the source invocation. Otherwise, an *invalid origin invocation* (hex 2C19) exception will be signaled.

Usage note: In cases where *source invocation pointer* is null, *operand 2* may be a constant.

Operand 3: The attribute selection template has the following format:

- Selection template header — Char(16)
  - Number of attributes — Bin(4)
  - Control flags — Char(1)
    - Attribute index indirect — Bit 0

      0 = *Offset to attribute index* specifies directly the location of the attribute index value
      1 = *Offset to attribute index* specifies the location of a space pointer which in turn specifies the location of the attribute index value

    - Reserved (binary 0) — Bits 1-7
  - Reserved (binary 0) — Char(3)
  - Offset to attribute index — Bin(4)
  - Length of attribute index — Bin(4)
- Attribute selection entries — Char(*)

The attribute selection entries are each 16 bytes long and have the following format:

- Attribute ID — Bin(4)
- Control flags — Char(1)
  - Indirect — Bit 0

    0 = *Offset to receiver* specifies directly the location of the attribute value
    1 = *Offset to receiver* specifies the location of a space pointer which in turn specifies the location of the attribute value

  - Return length — Bit 1

    0 = A length field is not present with the attribute
    1 = A length field precedes the attribute

  - Return status — Bit 2

    0 = A status field is not present with the attribute
    1 = A status field precedes the attribute

  - Pad — Bit 3

    0 = No pad field is assumed to precede the attribute
    1 = A pad field of zero, eight, or twelve bytes is assumed to precede the attribute

  - Reserved (binary 0) — Bits 4-7
- Reserved (binary 0) — Char(3)
- Offset to receiver — Bin(4)
- Length of receiver — Bin(4)

Basic structure: The "attribute selection template" allows the user of **MATINVAT** considerable flexibility in deciding what invocation attributes are to be materialized and where their materializations are to be returned. This flexibility is achieved by having the "attribute selection template" consist of a header, followed by a series of entries, each of which identifies an attribute to be materialized, the location where it is to be materialized, and the amount of space reserved for its materialization.

The template header specifies the number of attribute entries present in the template, and it also allows the specification of an optional *attribute index* field. The *attribute index* field, if present, identifies the first *attribute selection entry* to be processed (causing entries prior to that one to be skipped). In addition, if the *attribute index* field is present, it is updated upon the normal or abnormal completion of the instruction to contain either zero (if completion is normal) or the number of the entry being processed (if the instruction ends with an exception).

Each *attribute selection entry* identifies the attribute to be materialized and the area where the materialization is to be returned. The attribute may be returned directly into the area addressed by the *operand 1* space pointer, or it may be returned into an area addressed by a space pointer which is, in turn, contained in the area addressed by the *operand 1* space pointer. These two cases are distinguished by the *indirect* bit.

In addition, each *attribute selection entry* contains:

- An offset value which is the offset relative to the *operand 1* space pointer where either the attribute's materialization area or the pointer to the attributes' materialization area is contained.
- A length value identifying the maximum number of bytes of data to be materialized for the attribute.
- A flag indicating whether the length of the attribute is to be materialized.
- A flag indicating whether the status of the attribute is to be materialized.
- A flag indicating whether a pad field precedes the attribute (or its pointer, if *indirect* is specified). If present, the length of this "pad" field is automatically adjusted so that the combined length of the length, status, and pad fields is either zero or 16, maintaining the relative quadword alignment of the modification value if the length and/or status fields are present.

Note that, for the sake of regularity, the fields of the *attribute selection template* header are arranged in the same general fashion as those in the *attribute selection entries*.

Field descriptions:

Number of attributes
Specifies how many 16-byte *attribute selection entries* follow.

Attribute index indirect
If *attribute index indirect* is binary zero, then *offset to attribute index* specifies the location where the attribute index is stored as as offset from the location addressed by the *operand 1* space pointer. If *attribute index indirect* is binary one, then the location identified by *attribute index offset* must be quadword aligned and must contain a space pointer. This space pointer in turn addresses the location where the attribute index value is stored.

Offset to attribute index
Specifies the offset to the attribute index or the offset to a pointer to the attribute index, depending on the value of *attribute index indirect*.

Length of attribute index
Specifies the length of the area where the attribute index value is stored. This field must have a value of either zero or four.

If this field has a value of zero, then the first attribute entry to be processed is the first attribute entry in the template, and no feedback is given as to which attribute entry was being processed at the time of an exception. *Attribute index indirect* and *attribute index offset* are ignored.

If this field has a value of four, then the value of the attribute index, treated as a signed bin(4) value, must be greater than or equal to one and less than or equal to *number of attributes*. In this case the attribute index identifies the attribute entry to be processed first (with the first entry in the template having an index of one), and, in the event of an exception, the attribute index value is modified by this instruction such that it identifies the attribute entry being processed at the time of the exception. If the instruction completes without an exception, then the attribute index value is set to zero.

Attribute ID

Specifies the attribute to be materialized. Values that may be specified are:

1 Invocation pointer to specified invocation. (16 bytes, quadword aligned.)

2 Automatic storage pointer. Space pointer to the automatic storage for this invocation. If no automatic storage exists for this invocation, then a null pointer is returned. (16 bytes, quadword aligned, access rights required.)

3 Static storage pointer. Space pointer to the static storage for this invocation (OPM invocations only). If no static storage exists for this invocation, or if the invocation is an NPM invocation, then a null pointer is returned. (16 bytes, quadword aligned, access rights required.)

Note: For NPM invocations there is no single "distinguished" static storage area, but instead there may be multiple static storage areas. The list of static storage areas corresponding to the invocation's activation can be obtained by using the **Materialize Activation Attributes** instruction.

4 Parameter list pointer. Space pointer to the parameter list passed to this invocation (NPM procedure invocations only). If the procedure for this invocation does not expect a parameter list, or if this is an NPM entry routine or an OPM program, then a null pointer is returned. (16 bytes, quadword aligned, access rights required.)

6 Program pointer. System pointer to the program for this invocation. If the program no longer exists then a null pointer is returned. (16 bytes, quadword aligned, access rights required.)

7 Space pointer to *module* associated space. For NPM procedures, this space pointer addresses the secondary associated space in the NPM program that was propagated from the primary associated space of the NPM module. For OPM programs, this space pointer addresses the program's primary associated space. If the appropriate associated space does not exist in the program or if the program no longer exists, then a null pointer is returned. For both NPM and OPM invocations the requesting invocation must have space authority to the program. (16 bytes, quadword aligned, access rights required.)

8 Pointer to containing scope. If the specified invocation is in a nested scope, then this is an invocation pointer to the invocation of the containing scope. Otherwise a null pointer is returned. (16 bytes, quadword aligned.)

9 Relative invocation offset to containing scope. If the specified invocation is in a nested scope, then this is the relative invocation offset to the invocation of the containing scope. Otherwise, a value of zero is returned. Note that the relative invocation offset will be a negative number and is relative to the specified invocation. (4 bytes.)

10 Lexical level number. Outer procedures have a lexical level number of 1. (4 bytes.)

11 Invocation number. (2 bytes.)

12 Invocation mark. (4 bytes.)

13 Activation mark. If no activation exists for this invocation, then a zero value is returned. (4 bytes.)

14 Activation group mark. (4 bytes.)

15 Invocation type. The possible values for invocation type are:

Hex 01 Call external

Hex 02 Transfer control

Hex 03 Event handler

Hex 04 External exception handler (for OPM program)

Hex 05 Initial program in process problem state

Hex 06 Initial program in process initiation state

Hex 07 Initial program in process termination state

Hex 08 Invocation exit (for OPM program)

Hex 09 Return or return/XCTL trap handler

Hex 0A Call program

Hex 0B Cancel handler (NPM only)

Hex 0C Exception handler (NPM only)

Hex 0D Call bound procedure/call with procedure pointer

Hex 0E Process Default Exception Handler

(1 byte.)

16 Routine type. The possible values for routine type are:

Hex 01 OPM Program

Hex 02 NPM Program Entry Procedure (PEP)

Hex 03 NPM Procedure

(1 byte.)

17 State invocation was invoked with. (2 bytes.)

18 State for invocation. (2 bytes.)

19 Invocation status of the specified invocation (including invocation flags).

Bit 0 Cancelled

Bit 1 Ending -- a return operation has been initiated from within the invocation or the actual termination of a cancelled invocation has begun.

Bit 2 Invocation interrupted by exception

Bit 3 Invocation interrupted by event (reserved)

Bit 4 Invocation is an OPM CALLX exception handler

Bit 5 Invocation contains an OPM CALLI exception handler

Bit 6 Invocation contains a signalled OPM branchpoint handler

Bit 7 Retry not allowed

Bit 8 Resume not allowed

Bit 9 Resume point has been modified

Bit 10 Invocation is a program entry procedure and is marked as the oldest in the activation group

Bits 11-15 Reserved

Bits 16-31 Invocation flags

(4 bytes.)

Performance consideration: When the only invocation status information required is the invocation flags, there may be a significant performance advantage if the following attribute is materialized instead of this one.

20 Invocation flags of the specified invocation. This attribute has the same format as the *invocation status* attribute, except that the first two bytes are returned as zero. (4 bytes.)

23 Cancel reason of the specified invocation. (4 bytes.)

24 Suspend point. Suspend pointer identifying the location within the invocation's routine where execution was suspended due to a call, interrupt, or machine operation. If the program no longer exists then a null pointer is returned. (16 bytes, quadword aligned, access rights required.)

25 Resume point.

A suspend pointer identifying the location within the invocation's routine where execution will resume if execution is allowed to resume in the invocation. If the invocation is suspended for some cause that permits resumption, then this is initially set to the location that logically follows the suspend point. If the invocation is suspended for some cause that does not permit resumption, then this is initially set to be a null pointer. If the resume point is modified via **Modify Invocation Attributes** then a suspend pointer (or null pointer) corresponding to the modified resume point is returned. If the program no longer exists or if the invocation is cancelled or ending, then a null pointer is returned. (16 bytes, access rights required, quadword aligned.)

26 Interrupt message invocation. If the invocation is interrupted due to an exception interrupt, and the message causing the interrupt has not been removed or modified to a non-interrupt state, then this is an invocation pointer which addresses the invocation to which the interrupt message is enqueued. If no interrupt cause currently exists, then a null pointer is returned. (16 bytes, quadword aligned.)

27 Interrupt message reference key. If the invocation is interrupted due to an exception interrupt, and the message causing the interrupt has not been removed or modified to a non-interrupt state, then this is the message reference key of the interrupt cause message. If no interrupt cause currently exists, then a value of zero is returned. (4 bytes.)

28 External exception handler's monitoring invocation. If the specified invocation is an external exception handler for an OPM program, then this is an invocation pointer identifying the invocation which enabled the handler (also the invocation where the exception message is currently enqueued). Otherwise, a null pointer is returned. (16 bytes, quadword aligned.)

29 External exception handler's message reference key. If the specified invocation is an external exception handler for an OPM program, then this is the message reference key of the corresponding exception message. Otherwise, a zero value is returned. (4 bytes.)

30 OPM internal exception handler's message reference key. If the specified invocation is an OPM invocation with an internal exception handler active, then this is the message reference key of the exception message corresponding to the currently active internal exception handler. Otherwise, a zero value is returned. (4 bytes.)

31 OPM branchpoint exception handler's message reference key. If the specified invocation is an OPM invocation with a branchpoint exception handler in a signalled state, then this is the message reference key of the exception message corresponding to the most recently signalled branchpoint exception exception handler. Otherwise, a zero value is returned. (4 bytes.)

32 Trap handler's message reference key. If the specified invocation was invoked as a trap handler, then this is the message reference key of the corresponding trap message. (Note that the trapped invocation is, by definition, the immediately preceding invocation.) Otherwise, a zero value is returned. (4 bytes.)

Where "access rights required" is specified above, the activation group of the invocation identified as the originating invocation must have activation group access rights to the activation group of the source invocation or else an *activation group access violation* (hex 2C12) exception is signaled.

The invocation with an invocation number of 1 is always the first invocation in the stack.

Indirect
If *indirect* is binary zero, then *offset to receiver* specifies the location where the selected attribute value is to be materialized as as offset from the location addressed by operand 1. If *indirect* is binary one, then the location identified by *offset to receiver*, after accounting for any length, status, or pad fields specified, must be quadword aligned and must contain a space pointer. This space pointer in turn addresses the location where the selected attribute value is to be materialized.

Return length
If *return length* and *return status* are both binary zero, then only the attribute itself is materialized. If *return length* is binary one, then the attribute (or attribute pointer, if *indirect* is true) is preceded by a four-byte value which specifies the length of the attribute (exclusive of the length value itself, and the status and pad fields, if present).

Return status
If *return status* is binary one, then the attribute (or attribute pointer, if *indirect* is true) is preceded by a four-byte value which contains the status of the attribute.

If the status value is returned, it has the following format:

Bits 0-2 Reserved (binary 0)

Bit 3 Attribute unavailable at this time. (Eg, asking for the system pointer to a destroyed OPM program.) The result returned is zeros for the minimum length defined.

Bit 4 Attribute not defined in this context. (Eg, asking for lexical level number from OPM invocation.) The result returned is zeros for the minimum length defined.

Bit 5 Attribute not defined at this time. (Eg, asking for interrupt message invocation when the invocation is not interrupted.) The result returned is zeros for the minimum length defined.

Bit 6 Attribute defined but null. (Eg, when asking for the resume point for an invocation for which resume is not currently allowed.) The result returned is zeros for the minimum length defined.

Bit 7 Attribute truncated. Indicates that the specified *length of receiver* was too small to allow the entire attribute to be returned. The truncated result is returned, as described earlier.

Bits 8-31 Reserved (binary 0)

If *return length* and *return status* are both binary one then the length field comes first, followed immediately by the status field.

Pad
If either *return length* or *return status* is binary one, and *pad* is also binary one, then twelve bytes of pad are assumed between the length or status value and the attribute (or attribute pointer, if *indirect* is true). If both *return length* and *return status* are binary one, and *pad* is also binary one, then eight bytes of pad are assumed between the status value and the attribute (or attribute pointer). If *return length* and *return status* are both binary zero, then no padding occurs, regardless of the value of *pad*. The area occupied by the pad is not modified by this instruction.

Note: *Pad* makes it easier to quadword align the area to receive the materialized attribute (if *indirect* is false) or the area containing the attribute pointer (if *indirect* is true) when *return status* and/or *return length* are also specified.

Offset to receiver
Specifies the offset to the location where the selected attribute value is to be materialized, or the offset to a pointer to the location, depending on the value of *indirect*.

Length of receiver
Specifies the length of the area where the attribute value is to be materialized.

This length indicates the length of the actual area available for materializing the attribute, and *does not* include the length of any length, status, or pad field. If the number of bytes of attribute data available to be materialized (exclusive of the status, length, and pad fields, if any) exceeds *length of receiver*, then only *length of receiver* bytes of data are returned. No exception is signalled in this case.

If *indirect* is a binary zero, then *length of receiver* indicates the length of the area located by *offset to receiver*. If *indirect* is a binary one, then *length of receiver* indicates the length of the area located by the indirect space pointer identified by *offset to receiver*.

In the case that *length of receiver* is sufficient to receive only part of a field in an attribute structure, then the partial field may or may not be materialized.

Individual attribute entries are processed in order, with the attributes specified by each entry being materialized before processing of the next entry begins. If an exception occurs while processing an attribute entry, then the attributes materialized due to the preceding attribute entries will still be present in their specified result locations.

For attributes which include pointers, the specified direct or indirect value location, after accounting for any length, status, or pad fields, must be quadword aligned or a *boundary alignment* (hex 0602) exception may occur. (The exception is not guaranteed to occur, eg, in the case where *length of receiver* is insufficient to include the materialized pointer, or when a null pointer is returned.)

If the value locations of individual attribute entries overlap, then the values will be overlaid in the sequence implied by the attribute entry order. If the value location of a non-indirect result overlays the location of the space pointer for an indirect result, then the validity of the space pointer will depend on the order of the associated entries.

Authorization Required

- Activation group access
  - From the activation group of the invocation issuing the instruction to the activation group of the originating invocation identified by operand 2
  - When an attribute annotated with "access rights required" is specified: From the activation group of the originating invocation identified by operand 2 to the activation group of the source invocation identified by operand 2
- Space authority
  - For the module associated space option, the requesting invocation must have space authority to the program executing in the source invocation identified by operand 2

Lock Enforcement: None

Exceptions

Exception		Operands 1	2	3	Other
06	Addressing				
	01 space addressing violation	X	X	X	
	02 boundary alignment	X	X	X	
	03 range	X	X	X	
	06 optimized addressability invalid	X	X	X	
08	Argument/parameter				
	01 parameter reference violation	X	X	X	
0A	Authorization				
	01 Unauthorized for operation				X
10	Damage encountered				
	04 system object damage state				X
	44 partial system object damage				X
1C	Machine-dependent exception				
	03 machine storage limit exceeded				X
20	Machine support				
	02 machine check				X
	03 function check				X
22	Object access				
	02 object destroyed	X	X	X	
	03 object suspended	X	X	X	
	08 object compressed				X
24	Pointer specification				
	01 pointer does not exist	X	X	X	
	02 pointer type invalid	X	X	X	
2C	Program execution				
	11 Process object access invalid		X		
	12 Activation group access violation		X		
	19 Invalid origin invocation		X		
	1A Invocation offset outside range of current stack		X		
2E	Resource control limit				
	01 user profile storage limit exceeded				X
32	Scalar specification				
	03 scalar value invalid		X		
36	Space management				
	01 space extension/truncation				X
38	Template specification				
	01 template value invalid			X	

Materialize Invocation Entry (MATINVE)

Op Code (Hex)	Operand 1	Operand 2	Operand 3
0547	Receiver	Selection information	Materialization options

Operand 1: Character variable scalar (fixed length)

Operand 2: Character(8) scalar (fixed length) or null.

Operand 3: Character(1) scalar (fixed length) or null.

Description: This instruction materializes the attributes of the specified invocation entry within the process issuing the instruction. The attributes specified by operand 3 of the invocation selected through operand 2 are materialized into the receiver designated by operand 1.

Operand 2 is an 8-byte template or a null operand. If operand 2 is null, it indicates that the attributes of the current invocation are to be materialized. If operand 2 is not null, it must be an 8-byte template which specifies the invocation to be materialized. Only the first 8 bytes are used. Any excess bytes are ignored. It has the following format:

• Selection information	Char(8)
— Relative invocation number	Char(2)
— Reserved	Char(6)

If operand 2 is not null, it is restricted to a constant with the **relative invocation number** field specifying a value of zero, which indicates that the attributes of the current invocation are to be materialized.

Operand 3 is a 1-byte value or a null operand. If operand 3 is null, it indicates that the attributes for a materialization option value of hex 00 are to be materialized. If operand 3 is not null, it must be a 1-byte value which specifies the type of materialization to be performed. Option values that are not defined below are reserved values and may not be specified. Only the first byte is used. Any excess bytes are ignored. It has the following format:

• Materialization options	Char(1)

Hex 00 = Long materialization
Hex 01 = Short materialization type 1
Hex 02 = Short materialization type 2
Hex 03 = Short materialization type 3
Hex 04 = Short materialization type 4
Hex 05 = Short materialization type 5

If operand 3 is not null, it is restricted to a constant character scalar or an immediate value.

Operand 1 specifies a receiver into which the materialized data is placed. It must specify a character scalar with a minimum length which is dependent upon the materialization option specified for operand 3. If the length specified for operand 1 is less than the required minimum, an exception is signaled. Only the bytes up to the required minimum length are used. Any excess bytes are ignored. For the materialization options which produce pointers in the materialized data, 16-byte space alignment is required for the receiver. The data placed into the receiver differs depending upon the materialization option specified. The following descriptions detail the formats of the optional materializations.

Long Materialization: For a materialization option value of hex 00. the minimum length for the receiver is 144 bytes. It has the following format:

- Hex 00 = Long materialization — Char(144)
  - Reserved — Char(12)
  - Mark counter — Bin(4)
  - Reserved — Char(32)
  - Associated program pointer (zero for data base select/omit program) — System pointer
  - Invocation number — Bin(2)
  - Invocation type — Char(1)

    Hex 00 = Data base select/omit program
    Hex 01 = Call external
    Hex 02 = Transfer control
    Hex 03 = Event handler
    Hex 04 = External exception handler
    Hex 05 = Initial program in process problem state
    Hex 06 = Initial program in process initiation state
    Hex 07 = Initial program in process termination state
    Hex 08 = Invocation exit
    Hex 09 = Return trap handler or return/XCTL trap handler
    Hex 0A = Call program
    Hex 0B = Reserved
    Hex 0C = Reserved
    Hex 0D = Reserved
    Hex 0E = Process Default Exception Handler
  - Reserved (binary 0) — Char(1)
  - Invocation mark — Bin(4)
  - State invocation was invoked with — Char(2)
  - State for invocation — Char(2)
  - Reserved — Char(4)
  - Automatic storage frame (ASF) pointer — Space pointer
  - Static storage frame (SSF) pointer — Space pointer
  - Reserved — Char(32)

Short Materialization Type 1: For a materialization option value of hex 01, the minimum length for the receiver is 16 bytes. It has the following format:

Hex 01 = Short materialization type 1 — Char(16)

- Associated program pointer (null for data base select/omit program) — System pointer

Short Materialization Type 2: For a materialization option value of hex 02, the minimum length for the receiver is 4 bytes. It has the following format:

Hex 02 = Short materialization type 2 — Char(4)

- Invocation mark — Bin(4)

Short Materialization Type 3: For a materialization option value of hex 03, the minimum length for the receiver is 16 bytes. It has the following format:

Hex 03 = Short materialization type 3	Char(16)
• ASF pointer	Space pointer

Short Materialization Type 4: For a materialization option value of hex 04, the minimum length for the receiver is 16 bytes. It has the following format:

Hex 04 = Short materialization type 4	Char(16)
• SSF pointer	Space pointer

Short Materialization Type 5: For a materialization option value of hex 05, the minimum length for the receiver is 4 bytes. It has the following format:

Hex 05 = Short materialization type 5	Char(4)
• State invocation was invoked with	Char(2)
• State for invocation	Char(2)

The **mark counter** value represents the current value of a counter used by the machine to mark all activations and a subset of the invocations created during the execution of a process with a unique value. This mark indicates the point at which the specific entry was allocated relative to the sequence of all activations and invocations that have been created over time within the process. That is, older activations will be identified by a *mark value* of lesser value, and newer activations and invocations will be identified by a *mark value* of higher value. The *mark counter* is just the highest number assigned so far as a unique id. It does not act as a counter of how many activations and invocations have been created in the process. This is because invocations are assigned a *mark value* only as needed by the machine to ensure the visible function of the mark count. That is, any invocation *mark value* which is supplied to the user of the machine does indeed uniquely identify the invocation and indicate its creation sequence relative to the activations and invocations currently existing in the process, but it is not an absolute counter.

The **associated program pointer** is a system pointer that locates the program associated with the invocation entry.

The **invocation number** is the stack depth of the invocation within the invocation stack. The *invocation number* of a new invocation entry is one more than that in the calling invocation. The first invocation in the current process has an invocation number of one.

The **invocation type** indicates how the associated program was invoked.

The **invocation mark** captures the current mark counter value to uniquely identify the invocation within the process. An activation implicitly created on behalf of the invocation will be identified by a mark value of equal value.

The **state invocation was invoked with** value represents the state in which the machine was running when the program was called or transfered to.

State for invocation value represents the state in which the machine is running the program.

The **ASF pointer** is a space pointer that is set to address the start of the ASF. associated with the invocation. The associated program's automatic data starts 64 bytes after the area addressed by this pointer.

The **SSF pointer** is a space pointer that is set to address the start of the SSF (static storage frame) associated with the invocation. The associated program's static data starts 64 bytes after the area

addressed by this pointer. This pointer will be set to a value of all zeros if the invoked program does not have static data.

The fields labeled reserved in the descriptions of the optional materializations are currently reserved for future use. These fields may be altered by this instruction depending upon the particular implementation of the machine. Any values set into these fields are meaningless.

Exceptions

Exception		Operands 1	2	3	Other
06	Addressing				
	01 space addressing violation	X	X	X	
	02 boundary alignment	X	X	X	
	03 range	X	X	X	
	06 optimized addressability invalid	X	X	X	
08	Argument/parameter				
	01 parameter reference violation	X	X	X	
10	Damage encountered				
	04 system object damage state				X
	44 partial system object damage				X
1C	Machine-dependent exception				
	03 machine storage limit exceeded				X
20	Machine support				
	02 machine check				X
	03 function check				X
22	Object access				
	01 object not found	X	X	X	
	02 object destroyed	X	X	X	
	03 object suspended	X	X	X	
	08 object compressed				X
24	Pointer specification				
	01 pointer does not exist	X	X	X	
	02 pointer type invalid	X	X	X	
2E	Resource control limit				
	01 user profile storage limit exceeded				X
32	Scalar specification				
	01 scalar type invalid	X	X	X	
	02 scalar attributes invalid	X	X	X	
	03 scalar value invalid	X	X	X	
36	Space management				
	01 space extension/truncation				X

Materialize Invocation Stack (MATINVS)

Op Code (Hex)	Operand 1	Operand 2
0546	Receiver	Process

Operand 1: Space pointer.

Operand 2: System pointer or null.

ILE access

```
MATINVS (
      receiver : space pointer;
  var process  : system pointer OR
                 null operand
)
```

Description: This instruction materializes the current invocation stack within the specified process.

The attributes of the invocation entries currently on the invocation stack of the process specified by operand 2 are materialized into the template specified by operand 1.

Operand 2 is a system pointer or a null operand. If operand 2 is null, it indicates that the invocation stack of the current process is to be materialized. If operand 2 is not null, it is a system pointer identifying the process control space associated with the process for which the invocation stack is to be materialized. If the subject process, identified by operand 2, is different from the process executing this instruction, the executing process must be the original initiator of the subject process or must have process control special authorization to the process control space associated with the subject process.

Operand 1 is a space pointer that addresses a 16-byte aligned template into which is placed the materialized data. The format of the data is:

- Materialization size specification — Char(8)
  - Number of bytes provided by the user — Bin(4)
  - Number of bytes available for materialization — Bin(4)
- Number of invocation entries — Bin(4)
- Mark counter — Bin(4)*
- Invocation entries — Char(*)
  (An invocation entry is materialized for each of the invocations currently on the invocation stack of the specified process.)

The invocation entries materialized are each 128 bytes long and have the following format:

- Reserved — Char(32)
- Associated program pointer — System pointer
  (null for data base select/omit program or a destroyed program)
- Invocation number — Bin(2)
- Invocation mechanism — Char(1)

  Hex 01 Call external

  Hex 02 Transfer control

Hex 03 Event handler

Hex 04 External exception handler (for OPM program)

Hex 05 Initial program in process problem state

Hex 06 Initial program in process initiation state

Hex 07 Initial program in process termination state

Hex 08 Invocation exit (for OPM program)

Hex 09 Return or return/XCTL trap handler

Hex 0A Call program

Hex 0B Cancel handler (NPM only)

Hex 0C Exception handler (NPM only)

Hex 0D Call bound procedure/call with procedure pointer

Hex 0E Process Default Exception Handler

- Invocation type — Char(1)

  Hex 01 OPM Program

  Hex 02 NPM Program Entry Procedure (PEP)

  Hex 03 NPM Procedure

- Invocation mark — Bin(4)*
- Instruction identifier — Bin(4)
  (zero for data base select/omit program, destroyed, damaged, or suspended program)
- Activation group mark — Bin(4)
- Suspend point — Suspend pointer
  (null for data base select/omit program or destroyed program)
- Reserved — Char(48)

Note: Values annotated with an asterisk (*) may not be materialized if operand 2 identifies a process other than the one executing this instruction. Unmaterialized fields are set to binary zeros.

Values annotated with a double asterisk (**) are materialized only if operand 2 is null or identifies the process executing this instruction, and the program executing this instruction is in system state. Unmaterialized fields are set to binary zeros.

The **number of invocations** value specifies the number of invocation entries provided in the materialization.

The **mark counter** value represents the current value of a counter used by the machine to mark all activations and invocations created during the execution of a process with a unique value. This mark indicates the point at which the specific entry was allocated relative to the sequence of all activations and invocations that have been created over time within the process.

The **associated program pointer** is a system pointer that locates the program associated with the invocation entry.

The **invocation number** is a number that uniquely identifies each invocation in the invocation stack. When an invocation is allocated, the invocation number of the new invocation entry is one more than that in the calling invocation. The first invocation in the current process state has an invocation number of one.

The **invocation type** indicates how the associated program was invoked.

The **invocation mark** indicates the point at which this invocation entry was allocated relative to the sequence of all activations and invocations that have been created over time within the process. This is set from the incremented mark counter value for each new invocation added to the invocation stack.

If the invocation type is a non-bound program the instruction id field will contain the instruction number which specifies the number of the instruction last being executed when the invocation passed control to the next invocation on the stack. If the invocation type is a bound program entry or a procedure, the instruction id field will contain the statement identifier, which is a compiler supplied number which allows the compiler to to identify the source statement associated with a particular sequence of instructions.

Note: If the program is damaged or destroyed or if a statement identifier was not supplied by the compiler, a value of 0 is set.

The **suspend point** is a suspend pointer which identifies the instruction last being executed when the invocation passed control to the next invocation on the stack.

The fields labeled reserved are currently reserved for future use. These fields may be altered by this instruction depending upon the particular implementation of the machine. Any values set into these fields are meaningless.

The first 4 bytes of the materialization identifies the total **number of bytes provided** for use by the instruction. This value is supplied as input to the instruction and is not modified by the instruction. A value of less than 8 causes the *materialization length* (hex 3803) exception to be signaled.

The second 4 bytes of the materialization identifies the total **number of** bytes available to be materialized. The instruction materializes as many bytes as can be contained in the area specified as the receiver. If the byte area identified by the receiver is greater than that required to contain the information requested, the excess bytes are unchanged.

No exceptions are signaled in the event that the receiver contains insufficient area for the materialization, other than the materialization length exception described previously.

When the materialization is performed for a process other than the one executing this instruction, the instruction attempts to interrogate, snapshot, the invocation stack of the other process concurrently with the ongoing execution of that process. In this case, the interrogating process and subject process may have interleaving usage of the processor resource. Due to this, the accuracy and integrity of the materialization is relative to the state, static or dynamic, of the invocation stack in the subject process over the time of the interrogation. If the invocation stack in the subject process is in a very static state, not changing over the period of interrogation, the materialization may represent a good approximation of a snapshot of its invocation stack. To the contrary, if the invocation stack in the subject process is in a very dynamic state, radically changing over the period of interrogation, the materialization is potentially totally inaccurate and may describe a sequence of invocations that was never an actual sequence that occurred within the process. In addition to the above exposures to inaccuracy in attempting to take the snapshot, the ongoing status of the invocation stack of the subject process may substantially differ from that reflected in the materialization, due to its continuing execution after completion of this instruction.

When the materialization is performed for the process executing this instruction, it does provide an accurate reflection of the status of the process' invocation stack. In this case, concurrent execution of this instruction with execution of other instructions in the process is precluded.

Authorization Required

- Process control special authorization
  - For materializing a different process than the one executing this instruction
- Retrieve
  - Contexts referenced for address resolution

Lock Enforcement

- Materialization
  - Contexts referenced for address resolution

Exceptions

Exception		Operands 1	2	Other
06	Addressing			
	01 space addressing violation	X	X	
	02 boundary alignment	X	X	
	03 range	X	X	
	06 optimized addressability invalid	X	X	
08	Argument/parameter			
	01 parameter reference violation	X	X	
0A	Authorization			
	01 unauthorized for operation	X	X	
10	Damage encountered			
	04 system object damage state			X
	44 partial system object damage			X
1C	Machine-dependent exception			
	03 machine storage limit exceeded		X	
20	Machine support			
	02 machine check			X
	03 function check			X
22	Object access			
	01 object not found	X	X	
	02 object destroyed	X	X	
	03 object suspended	X	X	
24	Pointer specification			
	01 pointer does not exist	X	X	
	02 pointer type invalid	X	X	
28	Process state			
	02 process control space not associated with a process		X	
2E	Resource control limit			

Exception		Operands 1	2	Other
	01 user profile storage limit exceeded			X
32	Scalar specification			
	01 scalar type invalid	X	X	
	02 scalar attributes invalid	X	X	
36	Space management			
	01 space extension/truncation			X
38	Template specification			
	03 materialization length	X		

Materialize Pointer (MATPTR)

Op Code (Hex)	Operand 1	Operand 2
0512	Receiver	Pointer

Operand 1: Space pointer.

Operand 2: System pointer, space pointer data object, data pointer, instruction pointer, invocation pointer, procedure pointer, label pointer, or suspend pointer.

ILE access

```
MATPTR (
     receiver : space pointer;
 var pointer  : pointer
)
```

Description: The materialized form of the pointer object referenced by operand 2 is placed in operand 1.

The format of the materialization is:

• Materialization size specification	Char(8)
— Number of bytes provided for materialization	Bin(4)
— Number of bytes available for materialization	Bin(4)
• Pointer type	Char(1)

Hex 01 = System pointer
Hex 02 = Space pointer
Hex 03 = Data pointer
Hex 04 = Instruction pointer
Hex 05 = Invocation pointer
Hex 06 = Procedure pointer
Hex 07 = Label pointer
Hex 08 = Suspend pointer

• Pointer description	Char(*)

Pointer description depends on the pointer type. One of the following pointer type formats is used.

• System pointer description	Char(68)

The system pointer description identifies the object addressed by the pointer and the context which the object specifies as its addressing context.

— Context identification	Char(32)
- Context type	Char(1)
- Context subtype	Char(1)
- Context name	Char(30)
— Object identification	Char(32)
- Object type	Char(1)
- Object subtype	Char(1)
- Object name	Char(30)
— Pointer authorization	Char(2)

- Object control	Bit 0
- Object management	Bit 1
- Authorization pointer	Bit 2
- Space authority	Bit 3
- Retrieve	Bit 4
- Insert	Bit 5
- Delete	Bit 6
- Update	Bit 7
- Ownership	Bit 8
- Reserved (binary 0)	Bits 9-15
— Pointer target information	Char(2)
- Pointer target accessible from user state	Bit 0
- Reserved (binary 0)	Bits 1-15

The first 4 bytes of the materialization identify the total **number of bytes provided** for use by the instruction. This value is supplied as input to the instruction and is not modified by the instruction. A value of less than 8 causes the *materialization length* (hex 3803) exception to be signaled.

The second 4 bytes of the materialization identify the total **number of bytes available** to be materialized. The instruction materializes as many bytes as can be contained in the area specified as the receiver. If the byte area identified by the receiver is greater than that required to contain the information requested, then the excess bytes are unchanged. No exceptions (other than the materialization length exception) are signaled in the event that the receiver contains insufficient area for the materialization.

Note: If the object addressed by the system pointer specifies that it is not addressed by a context or if the context is destroyed, the **context identification** field is hex 00. If the object is addressed by the machine context, a *context type* field of hex 81 is returned. No verification is made that the specified context actually addresses the object.

The following lists the *object type* codes for system object references:

Value (Hex)	Object Type
01	Access group
02	Program
03	Module
04	Context
06	Byte string space
07	Journal space
08	User profile
09	Journal port
0A	Queue
0B	Data space
0C	Data space index
0D	Cursor
0E	Index
0F	Commit block
10	Logical unit description
11	Network description
12	Controller description
13	Dump space
14	Class of service description
15	Mode Description
16	Network interface description
17	Connection list

18	Queue space
19	Space
1A	Process control space
1B	Authorization list
1C	Dictionary

Note: Only the authority currently stored in the system pointer is materialized.

If the **pointer target accessible from user state** field has a value of 1, then the system pointer addresses an object that is in user domain. If the *pointer target accessible from user state* field has a value of 0, then the system pointer addresses an object that is not in user domain.

- Data pointer description — Char(75)

  The data pointer description describes the current scalar and array attributes and identifies the space addressability contained in the data pointer.

  – Scalar and array attributes — Char(7)

    - Scalar type — Char(1)

      Hex 00 = Signed binary
      Hex 01 = Floating-point
      Hex 02 = Zoned decimal
      Hex 03 = Packed decimal
      Hex 04 = Character
      Hex 06 = Onlyns
      Hex 07 = Onlys
      Hex 08 = Either
      Hex 09 = Open
      Hex 0A = Unsigned binary

    - Scalar length — Char(2)

      If binary, character, floating-point, Onlyns, Onlys, Either, or Open:

      - Length — Bits 0-15

      If zoned decimal or packed decimal:

      - Fractional digits — Bits 0-7
      - Total digits — Bits 8-15

    - Reserved (binary 0) — Bin(4)

  – Data pointer space addressability — Char(68)

    - Context identification — Char(32)
      - Context type — Char(1)
      - Context subtype — Char(1)
      - Context name — Char(30)

    - Object identification — Char(32)
      - Object type — Char(1)
      - Object subtype — Char(1)
      - Object name — Char(30)

    - Offset into space — Bin(4)

  Note: If the object containing the space addressed by the data pointer is not addressed by a context, the **context** field is hex 00. If the object is addressed by the machine context, a *context type* field of hex 81 is returned.

Support for usage of a Data Pointer describing an Onlyns, Onlys, Either, or Open scalar value is limited. For more information, refer to the Copy Extended Characters Left Adjusted With Pad, Set Data Pointer Attributes, and Create Cursor instructions.

- Space pointer description — Char(70)

  The space pointer description describes space addressability contained in the space pointer.

  - Context identification — Char(32)
    - Context type — Char(1)
    - Context subtype — Char(1)
    - Context name — Char(30)
  - Object identification — Char(32)
    - Object type — Char(1)
    - Object subtype — Char(1)
    - Object name — Char(30)
  - Offset into space — Bin(4)
  - Pointer target information — Char(2)
    - Pointer target accessible from user state — Bit 0
    - Reserved (binary 0) — Bits 1-15

  If the **pointer target accessible from user state** field has a value of 1, then the space pointer addresses a space that is in user domain and is writeable when the process is running in user state. If the *pointer target accessible from user state* field has a value of 0, then the space referenced by the pointer is either not in user domain or it is not writeable when the process is running in user state.

  Note: If the object containing the space addressed by the space pointer is not addressed by a context, the **context** field is hex 00. If the object is addressed by the machine context, a *context type* field of hex 81 is returned.

- Instruction pointer description — Char(68)

  The instruction pointer description describes instruction addressability contained in the instruction pointer.

  - Context identification — Char(32)
    - Context type — Char(1)
    - Context subtype — Char(1)
    - Context name — Char(30)
  - Program identification — Char(32)
    - Program type — Char(1)
    - Program subtype — Char(1)
    - Program name — Char(30)
  - Instruction number — Bin(4)

  If the program containing the instruction currently being addressed by the instruction pointer is not addressed by a context, the *context* field is hex 00.

- Invocation pointer description — Char(23)

  The invocation pointer description describes invocation addressability contained in the invocation pointer.

  - Pointer status — Char(1)
    - Process object no longer exists — Bit 0
    - Pointer is from another process — Bit 1
    - Reserved (binary 0) — Bits 2-7
  - Reserved (binary 0) — Char(6)
  - Containing process — System pointer

  Process object no longer exists
  If this bit has a value of 1, then the process object (invocation) referenced by the pointer no longer exists.

  Pointer is from another process
  If this bit has a value of 1, then the process object (invocation) referenced by the pointer exists but belongs to a process other than the current one.

  Containing process
  A system pointer to the process control space object to which the process object belongs. A null pointer is returned if the process object (invocation) no longer exists.

- Procedure pointer description — Char(55)

  The procedure pointer description describes the activation and procedure addressability contained in the procedure pointer.

  - Pointer status — Char(1)
    - Process object no longer exists — Bit 0
    - Pointer is from another process — Bit 1
    - Referenced program cannot be accessed — Bit 2
    - Reserved (binary 0) — Bits 3-7
  - Reserved (binary 0) — Char(6)
  - Module number — Ubin(4)
  - Procedure number — Ubin(4)
  - Activation mark — Ubin(4)
  - Activation group mark — Ubin(4)
  - Containing program — System pointer
  - Containing process — System pointer

  Process object no longer exists
  If this bit has a value of 1, then the process object referenced by the pointer (the activation) no longer exists. All of the remaining information is returned as binary zeros.

  Pointer is from another process
  If this bit has a value of 1, then the process object referenced by the pointer belongs to a process other than the current one. With the exception of the containing process pointer, all of the remaining information is returned as binary zeros.

  Referenced program cannot be accessed
  If this bit has a value of 1, then the program referenced by the pointer could not be accessed to extract the program-related information. The may be because the program is damaged, sus-

pended, compressed, or destroyed. The program pointer. module number. and procedure number are returned as binary zeros.

Module number
Index in the module list of the bound program for the module whose activation the pointer addresses.

Procedure number
Index in the procedure list of the module for the procedure addressed by the pointer.

Activation mark
The activation mark of the activation that contains the activated procedure. Zero if the program activation no longer exists.

Activation group mark
An activation group mark of the activation group that contains the activated procedure. Zero if the program activation no longer exists.

Containing program
A system pointer to the program object that contains the procedure. Null if the program activation no longer exists.

Containing process
A system pointer to the process control space object which contains the procedure's activation group. A null pointer is returned if the process control space object no longer exists, of if it is no longer possible to determine the containing process for a destroyed activation group.

- Label pointer description — Char(*)

  The label pointer description describes instruction addressability contained in the label pointer.

  - Pointer status — Char(1)
    - Reserved (binary 0) — Bit 0
    - Reserved (binary 0) — Bit 1
    - Referenced program is damaged, suspended, compressed or destroyed — Bit 2
    - Reserved (binary 0) — Bits 3-7
  - Reserved (binary 0) — Char(6)
  - Module number — Ubin(4)
  - Procedure number — Ubin(4)
  - Number of statement IDs — Ubin(4)
  - Internal identifier — Char(4)
  - Containing program — System pointer
  - Statement ID (repeated) — Ubin(4)

Referenced program is damaged, suspended, compressed, or destroyed
If this bit has a value of 1, then the program referenced by the pointer could not be accessed to extract the remaining information. The remainder of the template is binary zeros with the exception of the program pointer, which will be binary zeros if the program has been destroyed or so seriously damaged that its identity cannot be determined.

Module number
Index in the module list of the bound program for the module containing the label.

Procedure number
Index in the procedure list of the module for the procedure containing the label.

Number of statement IDs
Number of entries in the statement ID list. (Multiple statement IDs may be associated with a single location in the created program due to optimizations that combine similar code sequences.)

Internal identifier
A machine-dependent value which identifies the label relative to the the internal structure of the program. For use by service personnel.

Containing program
A system pointer to the program object that contains the label.

Statement ID
Each statement ID is a compiler-supplied unsigned Bin(4) number which allows the compiler to identify the source statement associated with a particular sequence of instructions.

- Suspend pointer description — Char(*)

The suspend pointer description describes instruction addressability contained in the suspend pointer.

Field	Type
— Pointer status	Char(1)
- Reserved (binary 0)	Bit 0
- Reserved (binary 0)	Bit 1
- Referenced program is damaged, suspended, compressed or destroyed	Bit 2
- Reserved (binary 0)	Bits 3-7
— Reserved (binary 0)	Char(6)
— Module number	Ubin(4)
— Procedure number	Ubin(4)
— Number of statement IDs	Ubin(4)
— Internal identifier	Char(4)
— Containing program	System pointer
— Statement ID (repeated)	Ubin(4)

Referenced program is damaged, suspended, compressed, or destroyed
If this bit has a value of 1, then the program referenced by the pointer could not be accessed to extract the remaining information. The remainder of the template is binary zeros with the exception of the program pointer, which will be binary zeros if the program has been destroyed or so seriously damaged that its identity cannot be determined.

Module number
Index in the module list of the bound program for the module containing the suspend point.

Procedure number
Index in the procedure list of the module for the procedure containing the suspend point.

Number of statement IDs
Number of entries in the statement ID list. (Multiple statement IDs may be associated with a single location in the created program due to optimizations that combine similar code sequences.)

Internal identifier
A machine-dependent value which locates the suspend point relative to the the internal structure of the program. For use by service personnel.

Containing program
A system pointer to the program object that contains the suspend point.

Statement ID
Each statement ID is a compiler-supplied unsigned Bin(4) number which allows the compiler to identify the source statement associated with a particular sequence of MI instructions.

Note: For suspend pointers which address non-bound programs, module number and procedure number are returned as binary zeros, and the statement ID list is returned with one value which is the MI instruction number of the suspend point.

If the pointer is a system pointer or a data pointer and is initialized but unresolved, the pointer is resolved before the materialization occurs.

This instruction will tolerate a damaged object referenced by operand 2 when operand 2 is a resolved pointer. The instruction will not tolerate a damaged context(s) or damaged programs when resolving pointers. Also, as a result of damage or abnormal machine termination, this instruction can indicate that an object is addressed by a context, when in fact the context will not show this as an addressed object.

A space pointer machine object cannot be specified for operand 2.

Exceptions

Exception		Operands 1	2	Other
06	Addressing			
	01 space addressing violation	X	X	
	02 boundary alignment	X	X	
	03 range	X	X	
	04 external data object not found		X	
	06 optimized addressability invalid	X	X	
08	Argument/parameter			
	01 parameter reference violation	X	X	
10	Damage encountered			
	04 system object damage state	X	X	X
	05 authority verification terminated due to damaged object			X
	44 partial system object damage	X	X	X
1C	Machine-dependent exception			
	03 machine storage limit exceeded			X
20	Machine support			
	02 machine check			X
	03 function check			X
22	Object access			
	01 object not found	X	X	
	02 object destroyed	X	X	
	03 object suspended	X	X	
	07 authority verification terminated due to destroyed object			X

Exception		Operands 1	2	Other
	08 object compressed			X
24	Pointer specification			
	01 pointer does not exist	X	X	
	02 pointer type invalid	X	X	
2E	Resource control limit			
	01 user profile storage limit exceeded			X
32	Scalar specification			
	01 scalar type invalid	X		
36	Space management			
	01 space extension/truncation			X
38	Template specification			
	03 materialization length exception	X		

Materialize Pointer Locations (MATPTRL)

Op Code (Hex)	Operand 1	Operand 2	Operand 3
0513	Receiver	Source	Length

Operand 1: Space pointer.

Operand 2: Space pointer.

Operand 3: Binary scalar.

ILE access

```
MATPTRL (
     receiver : space pointer;
     source   : space pointer;
 var length   : signed binary
)
```

Description: This instruction finds the pointers in a subset of a space and produces a bit mapping of their relative locations.

The area addressed by the operand 2 space pointer is scanned for a length equal to that specified in operand 3. A bit in operand 1 is set for each 16 bytes of operand 2. The bit is set to binary 1 if a pointer exists in the operand 2 space, or the bit is set to binary 0 if no pointer exists in the operand 2 space.

Operand 1 is a space pointer addressing the receiver area. One bit of the receiver is used for each 16 bytes specified by operand 3. If operand 3 is not a 16-byte multiple, then the bit position in operand 1 that corresponds to the last (odd) bytes of operand 2 is set to 0. Bits are set from left to right (bit 0, bit 1,...) in operand 1 as 16-byte areas are interrogated from left to right in operand 2. The number of bits set in the receiver is always a multiple of 8. Those rightmost bits positions that do not have a corresponding area in operand 2 are set to 0.

The format of the operand 1 receiver is:

• Template size specification	Char(8)
— Number of bytes provided by the user	Bin(4)
— Number of bytes available for materialization	Bin(4)
• Pointer locations	Char(*)

Operand 2 must address a 16-byte aligned area; otherwise, a *boundary alignment* (hex 0602) exception is signaled. If the value specified by operand 3 is not positive, the *scalar value invalid* (hex 3203) exception is signaled.

The first 4 bytes of the materialization identify the total **number of bytes provided** for use by the instruction. This value is supplied as input to the instruction and is not modified by the instruction. A value of less than 8 causes the *materialization length* (hex 3803) exception to be signaled.

The second 4 bytes of the materialization identify the total **number of bytes available** to be materialized. The instruction materializes as many bytes as can be contained in the area specified as the receiver. If the byte area identified by the receiver is greater than that required to contain the information requested, then the excess bytes are unchanged. No exceptions (other than the *materialization*

length (hex 3803) exception) are signaled in the event that the receiver contains insufficient area for materialization.

Exceptions

Exception		Operands 1	2	3	Other
06	Addressing				
	01 space addressing violation	X	X	X	
	02 boundary alignment	X	X	X	
	03 range	X	X	X	
	06 optimized addressability invalid	X	X	X	
08	Argument/parameter				
	01 parameter reference violation	X	X	X	
10	Damage encountered				
	04 system object damage state	X	X	X	X
	44 partial system object damage	X	X	X	X
1C	Machine-dependent exception				
	03 machine storage limit exceeded				X
20	Machine support				
	02 machine check				X
	03 function check				X
22	Object access				
	01 object not found			X	
	02 object destroyed	X	X	X	
	03 object suspended	X	X	X	
	08 object compressed				X
24	Pointer specification				
	01 pointer does not exist	X	X	X	
	02 pointer type invalid	X	X	X	
2E	Resource control limit				
	01 user profile storage limit exceeded				X
32	Scalar specification				
	03 scalar value invalid			X	
36	Space management				
	01 space extension/truncation				X
38	Template specification				
	03 materialization length exception			X	

Materialize System Object (MATSOBJ)

Op Code (Hex)	Operand 1	Operand 2
053E	Receiver	Object

Operand 1: Space pointer.

Operand 2: System pointer.

ILE access

```
MATSOBJ (
     receiver : space pointer;
 var object   : system pointer
)
```

Description: This instruction materializes the identity and size of a system object addressed by the system pointer identified by operand 2. It can be used whenever addressability to a system object is contained in a system pointer.

The format of the materialization is:

- Materialization size specification — Char(8)
  - Number of bytes provided by the user — Bin(4)
  - Number of bytes available for materialization — Bin(4)
- Object state attributes — Char(2)
  - Suspended state — Bit 0

    0 = Not suspended
    1 = Suspended
  - Damage state — Bit 1

    0 = Not damaged
    1 = Damaged
  - Partial damage state — Bit 2

    0 = No partial damage
    1 = Partial damage
  - Existence of addressing context — Bit 3

    0 = Not addressed by a temporary context
    1 = Addressed by a temporary context
  - Dump for previous release permitted — Bit 4

    0 = Dump for previous release not permitted.
    1 = Dump for previous release permitted.
  - Object compressed — Bit 5

    0 = Object not compressed
    1 = Object compressed (partially or completely)
  - ASP overflow — Bit 6

    0 = No part of the object has overflowed its ASP

1 = Some part of the object has overflowed its ASP

– Reserved (binary 0)	Bits 7-15
• Context identification	Char(32)
– Context type	Char(1)
– Control subtype	Char(1)
– Context name	Char(30)
• Object identification	Char(32)
– Object type	Char(1)
– Object subtype	Char(1)
– Object name	Char(30)
• Timestamp of creation	Char(8)
• Size of associated space	Bin(4)
• Object size	Bin(4)
• Owning user profile identification	Char(32)
– User profile type	Char(1)
– User profile subtype	Char(1)
– User profile name	Char(30)
• Timestamp of last modification	Char(8)
• Recovery options	Char(4)
• Performance class	Char(4)
• Initial value of space	Char(1)
• Object audit attribute	Char(1)

Hex 00 = No audit for this object
Hex 02 = Audit change for this object
Hex 03 = Audit read and change for this object
Hex 04 = Audit read and change for this object if the user profile is being audited

• Reserved	Char(2)
• Object authorization list (AL) status	Bin(2)

0 = Object not in an AL
1 = Object in AL

• Authorization list identification	Char(48)
– Authorization list (AL) status	Bin(2)

0 = Valid AL
1 = Damaged AL
2 = Destroyed AL (no name below)

– Reserved	Char(14)
– Authorization list type	Char(1)
– Authorization list subtype	Char(1)
– Authorization list name	Char(30)
• Dump for previous release reason code	Bit(64)

• Maximum possible associated space size	Bin(4)
• Timestamp of last use of object	Char(8)
• Count of number of days object was used	Ubin(2)
• Program attributes	Char(2)
– Program state provided	Bit 0
0 = No program state value 1 = Program state value present	
– Reserved (binary 0)	Bits 1-7
– Type of program	Char(1)
Hex 00 = Original program Hex 01 = Bound program Hex 02 = Service program	
• Domain of object	Char(2)
• Program state for program or module	Char(2)
• MI-supplied information	Char(8)
• Earliest compatible release	Char(2)
– Reserved	Bits 0-3
– Version	Bits 4-7
– Release	Bits 8-11
– Modification level	Bits 12-15
• Object size in pages	Ubin(4)
• Reserved	Char(110)

Additional Description:: This instruction will tolerate a damaged object referenced by operand 2 when operand 2 is a resolved pointer. The instruction will not tolerate a damaged context(s) or damaged programs when resolving pointers. Also, as a result of damage or abnormal machine termination, this instruction can indicate that an object is addressed by a context, when in fact the context will not show this as an addressed object. The Modify Addressability instruction can be used to correct this problem.

The **existence of addressing context** field indicates whether the previously (or currently) addressing context was (is) temporary. This field is 0 if the object was (is) not addressed by a temporary context.

The **dump for previous release permitted** field will indicate if the object is eligible for a Request I/O instruction in which a dump for previous is requested[1] . When this field indicates that the object is not eligible, the **dump for previous release reason code** can be used to determine why the object is not eligible.

The **object compressed** field indicates whether the encapsulated part of the object is either partially or completely compressed. The encapsulated part(s) of some object types can be compressed by object-specific create or modify instructions. For example, the executable and observation parts of a program object can be compressed and decompressed by the Modify Program instruction, and can also be decompressed implicitly by machine operation (see the Modify Program instruction for details). Use the object-specific materialization instruction for this type of object (for example, the Materialize

[1] 'Previous release' refers to the previous mandatory release. This is release N-1, mod level zero when release N is the current release. (For version 2, release 1.1, the previous mandatory release is version 1, release 3.0.)

Program instruction for program objects) to determine exactly which part(s) of the object are compressed.

The **ASP overflow** field indicates whether any part of the object is stored in an ASP other than the ASP specified at the time the object was created. If any object created in one ASP has parts that are in a different ASP (due to lack of sufficient available storage in the original ASP), then none of the objects in the first ASP are protected in the event of a failure of any other ASP in the system. By deleting objects that have overflowed, however, it may be possible to eliminate the ASP overflow condition and restore the protection that ASPs provide. Use the object-specific materialization instruction for this type of object to determine what ASP was specified at the time the object was created.

If the object addressed by the system pointer specifies that it is not addressed by a context or if the context is destroyed, the *context type* field is hex 00. If the object is addressed by the machine context, a *context type* field of hex 81 is returned. No verification is made that the specified context actually addresses the object.

Valid **object type** fields and their meanings are:

Value (Hex)	Object Type
01	Access group
02	Program
03	Module
04	Context
06	Byte string space
07	Journal space
08	User profile
09	Journal port
0A	Queue
0B	Data space
0C	Data space index
0D	Cursor
0E	Index
0F	Commit block
10	Logical unit description
11	Network description
12	Controller description
13	Dump space
14	Class of Service Description
15	Mode Description
16	Network interface description
17	Connection list
18	Queue space
19	Space
1A	Process control space
1B	Authorization List
1C	Dictionary

The **timestamp** field is materialized as an 8-byte unsigned binary number in which bit 41 is equal to 1024 microseconds. The **timestamp of creation** field is implicitly set when an object is created.

If the object has an associated space, the **maximum possible associated space size** field will be returned with a value which represents the maximum size to which the associated space can be extended. This value depends on the internal packaging of the object and its associated space as well

as (possibly) the maximum space size field as optionally specified during the create of the object (or on the Create Duplicate Object instruction, if that is how the object was created).

The **object size** field will contain the size of the object in bytes up to a value of 2G-1 (2147483647). If the object's size is greater than this, a value of zero will be returned in the *object size* field. In this case, the **object size in pages** field should be used to get the object's actual size. This field will always contain the object's true size in number of pages.

If the object is a temporary object and is, therefore, owned by no user profile, the *user profile type* field is assigned a value of hex 00.

The **timestamp of last modification** field is explicitly set by the Modify System Object instruction. It is implicitly set, except for the objects restricted below, by any instruction or IMPL function that modifies or attempts to modify an object attribute value or an object state. The timestamp of last modification field is only ensured as part of the normal ensuring of objects.

Implicit setting of the timestamp of last modification field is restricted for the following objects and will only occur for generic, nonobject specific, operations on them such as Rename Object for example.

- Logical unit description
- Controller description
- Network description
- Access group
- Queue

No modification time stamp will be provided for the following objects and a value of zero will be returned in the materialization template for the *modification time stamp*.

- Process control space

The **object authorization list status** field indicates whether or not the object is contained in an authorization list. If it is, the **authorization list identification information** provides the name of the authorization list, except when the authorization list is indicated as destroyed, in which case, the name information is meaningless.

The **dump for previous release reason code** can be used to determine why the object is not eligible according to the *dump for previous release permitted* field. Currently reason codes are only architected for programs. The reason code structure for programs is mapped as follows. Note that more than one reason may be returned.

- Program dump for previous release reason code — Bit(64)
  - Language version and release reason — Bit 0

    0 = Language version and release is not a reason
    1 = Language version and release is one reason
  - Level of machine instructions used reason — Bit 1

    0 = The level of machine instructions used in the program is not a reason
    1 = Machine instructions not available in the previous release are used
  - Program observability reason — Bit 2

    0 = Lack of program observability is not a reason
    1 = Program is not observable and must be to be moved to previous release
  - Reserved — Bits 3-63

The **timestamp of last use of object** field and the **count of number of days object was used** field are set by the Modify System Object instruction or by the Call External or Transfer Control instructions on the objects first use on that day. The timestamp value is only good for the date. The time value obtained from this timestamp is not accurate.

The type of program field indicates the Program Model of a program object, which is determined by how the program was created. It is only present when operand 2 points to a program object. This field is necessary since the object type and object subtype do not provide enough information to identify the Program Model of a program object. Knowing the program type is useful in selecting appropriate program specific instructions.

The **domain of object** field contains the value of the state under which a program or procedure must be running to access this object.

The **program state for program or module** field contains the state under which the program runs. It is only present when the **program state provided** flag is on.

The **MI-supplied information** is simply an 8 byte character field which can be set into an object with the Modify System Object (MODSOBJ) instruction and materialized with the Materialize System Object (MATSOBJ) instruction. The machine has no knowledge or dependencies on the content of this field.

The **earliest compatible release** field contains the earliest release that in which the object can be used in.

Authorization Required

- Retrieve
  - Contexts referenced for address resolution

Lock Enforcement

- Materialize
  - Operand 2
  - Contexts referenced for address resolution

Exceptions

Exception		Operands 1	2	Other
06	Addressing			
	01 space addressing violation	X	X	
	02 boundary alignment	X	X	
	03 range	X	X	
	06 optimized addressability invalid	X	X	
08	Argument/parameter			
	01 parameter reference violation	X	X	
0A	Authorization			
	01 unauthorized for operation		X	
10	Damage encountered			
	04 system object damage state	X	X	X

Exception		Operands 1	2	Other
	05 authority verification terminated due to damaged object			X
	44 partial system object damage	X	X	X
1A	Lock state			
	01 invalid lock state		X	
1C	Machine-dependent exception			
	03 machine storage limit exceeded			X
20	Machine support			
	02 machine check			X
	03 function check			X
22	Object access			
	01 object not found	X	X	
	02 object destroyed	X	X	
	03 object suspended	X	X	
	07 authority verification terminated due to destroyed object			X
	08 object compressed			X
24	Pointer specification			
	01 pointer does not exist	X	X	
	02 pointer type invalid	X	X	
2E	Resource control limit			
	01 user profile storage limit exceeded			X
32	Scalar specification			
	01 scalar type invalid	X	X	
36	Space management			
	01 space extension/truncation			X
38	Template specification			
	03 materialization length exception	X		

Chapter 22. Machine Interface Support Functions Instructions

This chapter describes all instructions used for machine interface support functions. These instructions are arranged in alphabetic order. For an alphabetic summary of all the instructions, see Appendix A,

Materialize Machine Attributes (MATMATR) 22-4
Materialize Machine Data (MATMDATA) 22-30

© Copyright IBM Corp. 1991, 1993

"Instruction Summary."

Materialize Machine Attributes (MATMATR)

Op Code (Hex)	Operand 1	Operand 2
0636	Materialization	Machine attributes

Operand 1: Space pointer.

Operand 2: Character(2) scalar or Space pointer.

ILE access

```
MATMATR1 (
     materialization      : space pointer;
 var machine_attributes : aggregate
)

OR

MATMATR2 (
     materialization      : space pointer;
     machine_attributes : space pointer
)
```

Warning: The following information is subject to change from release to release. Use it with caution and be prepared to adjust for changes with each new release.

Description: The instruction makes available the unique values of machine attributes. The values of various machine attributes are placed in the receiver.

Operand 2 specifies options for the type of information to be materialized. Operand 2 is specified as an attribute selection value (Character(2) scalar).

The machine attributes are divided into nine groups. Byte 0 of the attribute selection operand specifies from which group the machine attributes are to be materialized. Byte 1 of the options operand selects a specific subset of that group of machine attributes.

Operand 1 specifies a space pointer to the area where the materialization is to be placed. The format of the materialization is as follows:

• Materialization size specification	Char(8)
— Number of bytes provided by the user	Bin(4)
— Number of bytes available for materialization	Bin(4)
• Attribute specification (as defined by the attribute selection)	Char(*)

The first 4 bytes of the materialization (operand 1) identify the total **number of bytes provided** for use use by the instruction. This value is supplied as input to the instruction and is not modified by the instruction. A value of less than 8 causes the *materialization length* (hex 3803) exception to be signaled.

The second 4 bytes of the materialization identify the total **number of bytes available** to be materialized. The instruction materializes as many bytes as can be contained in the area specified as the receiver. If the byte are identified by the receiver is greater than that required to contain the information requested for materialization, then the excess bytes are unchanged. No exceptions (other than the

materialization length (hex 3803) exception) are signaled in the event that the receiver contains insufficient area for the materialization.

The machine attributes selected by operand 2 are materialized according to the following selection values:

Selection Value **Attribute Description**

Hex 0004 Machine serial identification

The machine serial identification that is materialized is an 8-byte character field that contains the unique machine identifier.

Hex 0100 Time-of-day clock

The time-of-day clock provides a consistent measure of elapsed time. The maximum elapsed time the clock can indicate is approximately 143 years.

The time-of-day clock is a 64-bit unsigned binary counter with the following format:

0................41 42 reserved 63

The bit positions of the clock are numbered from 0 to 63.

The clock is incremented by adding a 1 in bit position 41 every 1024 microseconds. Bit positions 42 through 63 are used by the machine and have no special meaning to the user. Note that these bits (42-63) may contain either binary 1's or binary 0's.

The maximum unsigned binary value that the time of day clock can be modified to contain is hex DFFFFFFFFFFFFFFF.

Hex 0104 Primary Initial process definition template

The primary initial process definition template is used by the machine to perform an initial process load.

No check is made and no exception is signaled if the values in the template are invalid; however, the next initial process load will not be successful.

Hex 0108 Machine initialization status record

The MISR (machine initialization status record) is used to report the status of the machine. The status is initially collected at IPL and then updated as system status changes.

The materialize format of the MISR is as follows:

- MISR status — Char(8)
  - Restart IMPL — Bit 0

    0 = IMPL was not initiated by the Terminate instruction
    1 = IMPL was initiated by the Terminate instruction
  - Manual power on — Bit 1

    0 = Power on not due to Manual power on
    1 = Manual power on occurred
  - Timed power on — Bit 2

    0 = Power on not due to Timed power on
    1 = Timed power on occurred
  - Remote power on — Bit 3

    0 = Power on not due to Remote power on
    1 = Remote power on occurred
  - Auto-power restart power on — Bit 4

0 = Power on not due to Auto-power restart power on
1 = Auto-power restart power on occurred

–	Uninterrupted power supply (UPS) battery low 0 = UPS battery not low 1 = UPS battery low	Bit 5
–	Uninterrupted power supply (UPS) bypass active 0 = UPS bypass not active 1 = UPS bypass active	Bit 6
–	Utility power failed, running on UPS 0 = Running on utility power 1 = Running on UPS	Bit 7
–	Uninterrupted power supply installed 0 = UPS not installed 1 = UPS installed, ready for use	Bit 8
–	Operation Panel battery failed 0 = Operation Panel battery good 1 = Operation Panel battery failed	Bit 9
–	Operation Panel self test failed 0 = Operation Panel self test successful 1 = Operation Panel self test failed	Bit 10
–	Console status 0 = Console is operative 1 = Console is inoperative	Bit 11
–	Console state 0 = Console is not ready 1 = Console is ready	Bit 12
–	OS Level mismatch 0 = Machine and OS version levels match. 1 = Machine and OS version levels mismatch	Bit 13
–	Reserved	Bit 14
–	Primary console status 0 = Not using Primary console 1 = Using Primary console	Bit 15
–	Reserved	Bit 16
–	ASCII console status 0 = Not using ASCII console 1 = Using ASCII console	Bit 17
–	Termination status 0 = Normal (TERMMPR) 1 = Abnormal	Bit 18
–	Duplicate user profile	Bit 19

(AIPL only)

0 = Not duplicate, new user profile created
1 = Duplicate found and used by AIPL

–	Damaged user profile (AIPL only) 0 = Not damaged, user profile used 1 = Damaged user profile, profile deleted and recreated	Bit 20
–	Damaged machine context 0 = Not damaged 1 = Machine context damaged	Bit 21
–	Object recovery list status 0 = Complete 1 = Incomplete	Bit 22
–	Recovery phase completion 0 = Complete 1 = Incomplete	Bit 23
–	Most recent machine termination 0 = Objects ensured 1 = Object(s) not ensured at most recent machine termination	Bit 24
–	Last MISR reset 0 = Object(s) ensured on every machine termination 1 = Object(s) not ensured on every machine termination since last MISR reset	Bit 25
–	Reserved	Bit 26-27
–	IPL Mode (can be materialized and modified) 00 = DST and BOSS in unattended mode 10 = DST and BOSS is attended mode	Bit 28-29
–	Service Processor power on 0 = Not first service processor power on 1 = First service processor power on	Bit 30
–	MISR damage 0 = MISR not damaged 1 = MISR damaged, information reset to default values	Bit 31
–	Auto keylock position 0 = Keylock not in auto position 1 = Keylock in auto position	Bit 32
–	Normal keylock position 0 = Keylock not in normal position 1 = Keylock in normal position	Bit 33
–	Manual keylock position 0 = Keylock not in manual position 1 = Keylock in manual position	Bit 34
–	Secure keylock position	Bit 35

0 = Keylock not in secure position
1 = Keylock in secure position

- Tower two presence on 9404 system unit — Bit 36

  0 = Tower two not present
  1 = Tower two present

- Battery status for tower one on 9404 system unit — Bit 37

  0 = Battery good for tower one
  1 = Battery low for tower one

- Battery status for tower two on 9404 — Bit 38

  0 = Battery good for tower two
  1 = Battery low for tower two

- Termination due to utility power failure and user specified delay time exceeded — Bit 39

  0 = Delay time not exceeded
  1 = Utility failure and delay time exceeded

- Termination due to utility power failure and battery low — Bit 40

  0 = Battery not low
  1 = Utility failure and battery low

- Termination due to forced microcode completion — Bit 41

  0 = Not forced microcode completion
  1 = Termination due to forced microcode completion

- Auto power restart disabled due to utility failure — Bit 42

  0 = Auto power restart not disabled
  1 = Auto power restart disabled

- Reserved — Bit 43

- Spread the Operating System — Bit 44

  0 = Do not spread the Operating System
  1 = Spread the Operating System

- Install from Disk/Tape — Bit 45

  0 = Install from tape
  1 = Install from disk

- Use Primary/Alternate PDT — Bit 46

  0 = Use Primary Process Definition Template
  1 = Use Alternate Process Definition Template

- Time/Date source — Bit 47

  0 = Time/Date is accurate
  1 = Time/Date default value used

• Install Type — Bin(2)

0 = Normal IPL
1 = Manual Install
2 = Automated Install

• Number of damaged main storage units	Bin(2)
• National language index (Can be materialized and modified)	Bin(2)
• Number of entries in object recovery list	Bin(4)
• Tape sequence number for an AIPL	Bin(4)
• Tape volume number for an AIPL	Bin(4)
• Address of object recovery list	Space pointer
• Process control space created as the result of IPL or AIPL	System pointer
• Queue Space object created as the result of an IPL or AIPL	System pointer
• Reserved	Char(16)
• Console Information list (Array of five entries each 80 bytes in size) (1st=Primary, 2-5=Reserved)	Char(400)
— Console entry	Char(80)
- Display LUD	System pointer
- Display CD	System pointer
- Controller model	Char(4)
- Controller type	Char(4)
- Controller serial number	Char(4)
- Controller object data	Char(12)
• Direct select address	Char(2)
— IOP bus number	Char(1)
— IOP card, board structure	Char(1)
- IOP card number	Bits 0-3
- IOP board number	Bits 4-7
• Logical bus address	Char(1)
• IOP unit address	Char(2)
• Resource Identifier	Char(4)
• Reserved	Char(3)
- Work station object data	Char(12)
• Direct select address	Char(2)
— IOP bus number	Char(1)
— IOP card board structure	Char(1)
- IOP card number	Bits 0-3
- IOP board number	Bits 4-7
• Logical bus address	Char(1)
• Device unit address	Char(2)

— Port	Char(1)
— Switch Setting	Char(1)
• Reserved	Char(7)
- Device type	Char(4)
- Device model	Char(4)
- Flags	Char(1)
• Information valid in entry	Bit 0
• Reserved	Bit 1-7
- Console keyboard type	Char(1)
- Console extended keyboard type	Char(1)
- Reserved	Char(1)
• Load/Dump Tape device information list (Array of two entries each 48 bytes in size) (1st=LUD information, 2nd=CD information)	Char(96)
— Load/Dump tape device entry	Char(48)
- Reserved	Char(16)
- LUD/CD information	Char(12)
• Direct select address	Char(2)
— IOP bus number	Char(1)
— IOP card board structure	Char(1)
- IOP card number	Bits 0-3
- IOP board number	Bits 4-7
• Logical bus address	Char(1)
• Device unit address	Char(2)
• Flags	Char(1)
— Information valid in entry	Bit 0
— Reserved	Bit 1-7
• Reserved	Char(6)
- Device type	Char(4)
- Device model	Char(4)
- Reserved	Char(12)
• Recovery object list (located by recovery object list pointer)	Char(*)
— Recovery entry (repeated for number of entries)	Char(32)
- Object pointer	System pointer
- Object type	Char(1)
- Object status	Char(15)

Restart IMPL indicates that a Terminate Machine Processing instruction was issued with the restart option set to yes. The machine performed an IMPL without powering down the machine.

Manual power on indicates the power switch on the operation panel was pressed to power the system on.

Timed power on indicates the system was powered on using the system value specified by the customer. This option will only be honored when the Timed power on function is enabled.

Remote power on indicates the system was power on by a phone call placed by the customer. This option will only be honored when the Remote power on function is enabled.

Auto-power restart indicates the system was automatically powered on after a utility failure occurred and power was restored. This option will only be honored when the Auto-power restart function is enabled.

UPS battery low indicates that a UPS battery is installed on the system and the battery is low.

UPS bypass active indicates that the UPS has been bypassed. If a utility power failure occurs, the UPS will not supply power.

UPS power failed indicates that a utility failure has occurred and the system is currently running on battery power.

UPS installed indicates that a Uninterrupted Power Supply is installed on the system and is available for use should the power fail.

Operation Panel battery failure indicates the battery in the operation panel has failed and the system will not be able to determine the correct time and date upon the next IMPL. An approximate time and date will be given to the customer for verification.

Operation Panel self test failed indicates the Operation Panel is possibly bad and some function concerning the operation panel may not work correctly.

Console status indicates whether the selected console is functioning normally or is inoperative.

Console state indicates whether the selected console is ready to be used.

Primary console status is set when the customer selected primary console is being used as the system console.

ASCII console status is set when a ASCII console is being used as the system console.

Termination status indicates how the previous IMPL was terminated. If *normal*, the Terminate Machine Processing instruction successfully terminated the previous IMPL. If *abnormal*, the Terminate Machine Processing instruction did not successfully terminate the previous IMPL. This also implies that some cleanup of permanent objects may be required by the user.

The duplicate user profile is valid only for AIPL and indicates if a user profile that is the same as the AIPL user profile to be created already exists in the machine context. The machine in this instance does not create the user profile for AIPL but rather uses the one located with the same name.

Damaged AIPL user profile indicates if the currently existing user profile was detected as damaged and a new user profile was created as specified in the AIPL user profile creation template.

Damaged machine context indicates if damage was detected in the machine context when an attempt was made to locate the duplicate user profile or to insert addressability to a newly created user profile. In either case, all current addressability is removed from the machine context, the new AIPL user profile is created, its addressability is inserted into the machine context, and the AIPL continues. Objects whose addressability was removed may have it reinserted using the Reclaim instruction for all objects or the Modify Addressability instruction for a specific object.

The object recovery list status field indicates that the status is complete unless one of the following conditions is true:

- The recovery list was lost.
- More objects were to be placed in the list but there was insufficient space.

The recovery phase completion field indicates that the status is complete unless one of the following conditions occurs:

- An object to be recovered and/or inserted into the object recovery list no longer exists.
- The objects to be recovered could not be determined due to loss of internal machine indicators that specified which objects were in use at machine termination.

The **most recent machine termination** field is set to 0 unless all objects were not ensured at the most recent machine termination.

The **last MISR reset** field is set to 0 if all objects were ensured at every machine termination since the MISR was last reset (to 0) using the Modify Machine Attributes instruction.

IPL mode indicates which mode DST and the OS will be IPLed. Either both will be attended or both will be unattended.

Service Processor power on indicates if this is the first time the service processor card has been powered on.

MISR Damage indicates if the machine detected that the MISR was damaged and it's contents has to be reset to the default system values.

Auto keylock position indicates if the keylock was is the auto position on the operation panel on the most recent IMPL.

Normal keylock position indicates if the keylock was is the normal position on the operation panel on the most recent IMPL.

Manual keylock position indicates if the keylock was is the manual position on the operation panel on the most recent IMPL.

Secure keylock position indicates if the keylock was is the secure position on the operation panel on the most recent IMPL.

Tower two present on 9404 system unit indicates if the system has second tower when the system is a 9404 system unit.

Battery status for tower one on 9404 system unit indicates if a UPS battery is installed on the first tower of a 9404 system unit, the battery is low.

Battery status for tower two on 9404 system unit indicates if a UPS battery is installed on the second tower of a 9404 system unit, the battery is low.

Termination due to utility power failure and user specified delay time exceeded indicates the last termination of the system was due to a utility power failure and the system value specified by the delay time had elapsed so the system was terminated.

Termination due to utility power failure and battery low indicates the last termination of the system was due to a utility power failure and while running on battery power the voltage dropped below a level to continue to power the system so the system was terminated.

Termination due to forced microcode completion indicates that the system when down by the user selecting power down from DST or the delayed power off switch was pressed on the operation panel.

Auto power restart disabled due to a utility failure indicates the microcode disabled the auto power restart option when a condition was detected that would prevent the auto power restart to function properly.

Reset utility power bits indicates that the power bits should be reset. This bit is only looked at when modifying the MISR.

Spread the operating system. indicates to spread the operating system on the next install instead of overlaying the existing objects. This bit is set to spread after a new dasd has been added.

Install from Disk/Tape indicates when performing an install to use the initial OS/400 install program off of disk or to load the initial OS/400 install program off of tape.

Primary/Alternate Process Definition Template indicates on IPL to initiate the initial OS/400 process using the Primary or the Alternate Process Definition Template.

Time/Date source informs OS/400 if the machine was able to determine the correct time/date or if it was forced to use the default time/date.

Install type is set indicate whether an IPL or install was performed and if an install was performed, what type of install occurred.

The **number of damaged main storage transfer blocks** field indicates the number of main storage transfer blocks that were detected as damaged by the machine during IMPL.

National language index is the value used to index to the the National language array kept by the system.

The **number of entries in the object recovery list** field indicates how many objects are listed in the space located by the *address of object recovery list* field.

The **tape sequence number** is set by the machine to allow the OS to perform their install.

The **tape volume number** is set by the machine to allow the OS to perform their install.

The **address of object recovery list** field contains a space pointer to the list of the potentially damaged objects that were identified during machine initialization. The number of such objects is indicated by the *number of entries in the object recovery list* field.

The **process control space created as a result of IPL or AIPL** is identified by a system pointer returned in this field.

The **Queue Space object pointer** addresses the queue space object that was implicitly created by the machine for use by the inital process. This is created for both an IPL or an AIPL. During IPL or AIPL processing, before the queue space is created by the machine, the machine will attempt to destroy the queue space addressed by the MISR (this will be the queue space used on the previous IPL or AIPL). If the destroy fails (the MI user may have destroyed it sometime in the previous IPL), no error is reported and the IPL or AIPL processing continues with the creation of the new queue space. The queue space is a permanent object, owned by the user profile used to initiate the initial process and is not addressed by a context.

The **console information list** contains information for each console device as obtained from the Resource Configuration Record or set by the customer.

The **Load/Dump tape device information** is information needed to build a Logical Unit Description and Controller Description object for the device used to install the operating system.

The **recovery object list** identifies objects that required some activity performed on the object(s) during IPL. The list is located by the *recovery object list pointer*.

Each entry in the list has the following general format:

• Object	System pointer
• Object type	Char(1)
• Object status	Char(15)
— General status	Char(2)
- Damaged	Bit 0

0 = Object not damaged
1 = Object damaged

- Reserved — Bit 1
- Suspended — Bit 2

  0 = Object not suspended
  1 = Object suspended

  Partially damaged — Bit 3

  0 = Object not partially damaged
  1 = Object partially damaged

- Journal synchronization — Bit 4

  0 = Synchronization complete or not necessary
  1 = Synchronization failure

- Reserved — Bits 5-6
- IPL detected damage — Bit 7

  0 = Any indicated damage was not detected by directory recovery
  1 = Indicated damage was detected by directory recovery

- Reserved — Bits 8-15

— Object specific status — Char(13)

(The format for the IPL recovery status for this portion of the object recovery list entries is different for each object type. A description of each follows by object type.)

- Commit block status — Char(2)

  — Decommit — Bit 0

  0 = The journal has successfully been read backwards until either a start commit or a decommit entry was found. An attempt has been made to decommit all the data base changes but the attempt may not have been successful if the data space is damaged or if the function check flag is on.
  1 = The journal has not successfully been read backwards to a start commit or decommit entry and all the changes have not been decommitted.

  — Journal read errors — Bit 1

  0 = No journal read errors
  1 = Journal read errors occurred during decommit

  — Journal write errors — Bit 2

  0 = No journal write errors
  1 = Journal write errors occurred during decommit

  — Partial damage to data space

  0 = No partial damage encountered
  1 = Partial damage encountered on 1 or more data spaces

  — Damage to data space — Bit 4

  0 = No damage encountered

  1 = Damage encountered on 1 or more data spaces

  — Function check — Bit 5

  0= No function check encountered

1= Function check encountered

- Reserved — Bit 6
- Data space during IMPL — Bit 7

  0= Data space is synchronized with the journal
  1= Data space is not synchronized with the journal. All changes may not be decommitted.

- Decommit reason code — Bits 8-10

  000 = Decommit not performed
  001 = Decommit at IPL
  010 = Process termination
  100 = Decommit instruction (all other values reserved)

- System initiated MI DECOMMIT — Bit 11

  0= Operating system did not initiate the decommit
  1= Operating system did initiate the decommit

- Reserved — Bits 12-15
- Reserved (binary 0) — Char(7)
- Start commit journal — Bin(4)
- Sequence number

- Data space
  - Status — Char(13)
    - Indexes detached from data space — Bit 0

      0 = Indexes remain attached
      1 = All indexes detached from this data space

      Reserved (binary 0) — Bits 1-15
  - Reserved (binary 0) — Char(7)
  - Ordinal entry number of last entry — Bin(4)
- Data space index
  - Status — Char(13)
    - Invalidated — Bit 0

      0 = Not invalidated
      1 = Invalidated

    - Recovered by journal — Bit 1

      0 = Not recovered
      1 = Recovered

    - Reserved (binary 0) — Bits 2-15

  .Reserved (binary 0) — Char(11)
- Journal port
  - Status — Char(13)
    - Synchronization status — Bit 0

      0 = All objects synchronized

1 = Not all objects synchronized

- Reserved	Bits 1-7
— Reserved	Char(10)
— Number of journal spaces attached	Bin(2)
• Journal space	
— Status	Char(13)
- Journal space usable	Bit 0
0 = Journal space is usable 1 = Journal space is not usable	
-	
- Threshold reached	Bit 1
0= Threshold has not been reached 1= Threshold has been reached	
Reserved	Bits 2-7
— Reserved	Char(4)
— First journal sequence number	Bin(4)
— Last journal sequence number	Bin(4)

All objects-Any damage detected during IPL is reported in the **general status** information. If this damage is detected as a result of special processing performed during directory rebuild, it is indicated in the **IPL detected damage** field. A **journal synchronization** failure indicates the designated object was not made current with respect to the journal. Subsequent attempts to apply journal changes from the journal to this object will not be allowed.

Commit block-All commit blocks that were attached to an active process during the previous IPL are interrogated at the following IPL. The system attempts to decommit any uncommitted changes referenced through these commit blocks. The results of this attempted decommit is reported in the status field. The system also returns the *journal entry sequence number* of the start commit journal entry (hex 0500) last created for this commit block if there were any uncommitted changes. If the number is not returned, a value of binary zero is returned.

Data space-If object damage was detected during IPL, the object is marked as damaged, damage is indicated in the object status field, and an event is signaled. In this case, the highest ordinal entry number is 0. In certain situations, the data space indexes over the data space become detached and therefore must be recreated. If the object is not damaged, the data space is usable and the highest ordinal entry number is set. The ordinal entry number of last entry indicates the last entry in the data space. Updates are not guaranteed. Updates may be out of sequence or partially applied and must be verified by the user for correctness.

Data space index-If object damage was detected during IPL, the object is marked as damaged, damage is indicated in the object status field, and an event is signaled. If the object was invalidated because changes were made in a data space addressed by the data space index, the data space index is included in the list and marked as invalidated. The *ASP number* of the data space index is indicated in the list. The associated data space is also included elsewhere in the object recovery list. Only damaged or invalidated data space indexes are included in the list.

Journal port-Each journal port in the system is interrogated at IPL. The status field contains the result of this checking and also the result of the attempt to synchronize the objects (if

necessary) being journaled through the indicated journal port. A default journal port is specified when created. and indicates the port is to be used by the machine in implicitly journaling objects. The system also returns the number of journal spaces attached to the journal port after IPL is complete.

Journal space-Each journal space that was attached to a journal port or used by the system to synchronize an object which was being journaled at the time of the previous machine termination is interrogated during IPL. The status field reports the results of this interrogation and synchronization use. Journal spaces are referenced by the object recovery list if this IPL was preceded by an abnormal failure, some unexpected condition was discovered during the IPL, or the journal space is a default journal space. The first journal sequence number on the journal space is returned. The last usable entry on the journal space is also identified. If the journal space is damaged, these fields will contain zeroes.

Hex 0118 Uninterruptible power supply delay time and calculated delay time. Note: The UPS delay time is meaningful only if a UPS is installed.

The format of the template for the uninterruptible power supply delay time (including the 8-byte prefix) is as follows:

Field	Type
• Number of bytes available	Bin(4)
• Number of bytes provided	Bin(4)
• UPS Delay time	Bin(4)
• Calculated UPS Delay time	Bin(4)

The delay time interval is the amount of time the system waits for the return of utility power. If a utility power failure occurs, the system will continue operating on the UPS supplied power. If utility power does not return within the user specified delay time, the system will perform a quick power down. The delay time interval is set by the customer. The calculated delay time is determined by the amount of main storage and DASD that exists on the system. Both values are in seconds.

Hex 012C Vital Product Data

The VPD (vital product data) is a template that contains information for memory card VPD, processor VPD, columbia/Colomis VPD, central electronic complex (CEC) VPD and the panel VPD.

The materialize format of the VPD (Including the 8-byte prefix) is as follows:

Field	Type
• Number of bytes available	Bin(4)
• Number of bytes provided	Bin(4)
• Reserved	Char(8)
• System VPD location	Char(48)
– Offset to memory VPD	Bin(4)
– Offset to Processor VPD	Bin(4)
– Offset to Columbia/Colomis	Bin(4)
– Offset to CEC VPD	Bin(4)
– Offset to Panel VPD	Bin(4)
– Reserved	Char(28)
• Main store memory VPD	Char(1040)
– Usable memory installed (In megabytes)	Bin(2)

— Minimum memory required (In megabytes)	Bin(2)
— Reserved	Char(12)
— Memory array (An array of 16, 64-byte entries)	Char(1024)
- Memory status	Char(2)
• Memory card status	Bit(2)
• Invalid memory card	Bit 0
• Unrecognized card	Bit 1
• Memory not supported	Bit 2
• Card/model mismatch	Bit 3
• Interface error	Bit 4
• Refresh error	Bit 5
• Addressing error	Bit 6
• 1st Nonterminating FRU	Bit 7
• 2nd Nonterminating FRU	Bit 8
• 3rd Nonterminating FRU	Bit 9
• Reserved	Bits 10-15
- Memory card size (In megabytes)	Bin(2)
- Number of CCINS for this card	Char(1)
- Array of CCINS for this card (Array of 8, 4-byte entries)	Char(32)
- Physical slot # of this card	Char(1)
- Reserved	Char(26)
• Processor VPD (An array of 16, 80-byte entries)	Char(1280)
— Processor status	Char(4)
- Processor card status	Bits 0-1
- Reserved	Bits 2-31
— Processor CCIN number	Char(4)
— Processor model number	Char(4)
— Processor part number	Char(12)
— Processor serial number	Char(4)
— Processor manufacturing ID	Char(4)
— Processor load identifier	Char(4)
— Reserved	CHAR(44)
• Columbia/Colomis VPD	Char(32)

(An array of 2, 64-byte entries)

(1st entry is Columbia, 2nd is Colomis)

— Card status	Char(4)
- Card usability status	Bits 0-1
- Reserved	Bits 2-31
— Card CCIN number	Char(4)
— Card model number	Char(4)
— Card part number	Char(12)
— Card serial number	Char(4)
— Card manufacturing ID	Char(4)
— Reserved	Char(32)
• CEC VPD	Char(32)
— Status of last CEC read	Char(4)
— System manufacturing ID	Char(4)
— System serial number	Char(4)
— System type	Char(4)
— System model number	Char(4)
— Pseudo model number	Char(4)
— System Password	Char(8)
• Panel VPD	Char(64)
— Panel VPD entry status	Char(2)
- Panel VPD is usable	Bit 0
- Reserved	Bits 1-15
— Panel type	Char(4)
— Panel model number	Char(3)
— Panel part number	Char(12)
— Panel serial number	Char(4)
— Panel manufacturing ID	Char(4)
— Alterable ROS part number	Char(12)
— Alterable ROS card number	Char(10)
— Alterable ROS ID	Char(1)
— Alterable ROS flag	Char(1)
— Alterable ROS fix ID	Char(1)
— Reserved	Char(10)

The **system VPD location information** is used to determine where each of the separate VPD sections start relative to the beginning of the VPD materialization template.

The **usable memory installed** field contains the amount of memory (in megabytes) which the system recognizes as being valid.

The **minimum memory required** field contains the amount of memory (in megabytes) which is required for the system to run at optimum performance.

The remainder of the memory VPD is an array of 16, 64 byte entries which contains specific information about each memory card installed on the system. The **memory card status** field should be interpreted in the following way:

- 00 = Memory card usable, no failures
- 01 = Memory card usable, but has failures
- 10 = Memory card is not installed
- 11 = Memory card is not usable due to critical failure

The remaining status bits will have a value of 1 if the condition is true or a value of 0 if the condition is not true.

The **memory card size** is a two byte field which will contain the number of megabytes of main store this card represents.

The **number of CCINs** field for this card contains a count of the number of CCINs which were found for this card. This number should be used to determine how many CCIN entries follow in the array of CCINs for this card field. This is a 32 byte field which is divided into 8, 4-byte entries. Each entry contains a CCIN number for this memory card.

The **Processor VPD** is an array of 16 entries, each 80 bytes in length. Each entry corresponds to a processor card. To determine the status of a processor card, the **processor card status** field should be interpreted in the following way:

- 00 = Processor usable, no failures
- 01 = Processor usable, but has failures
- 10 = Processor is not installed
- 11 = Processor is not usable due to critical failure

The **Columbia/Colomis VPD** is an array of 2 entries, each 64 bytes in length. The first entry contains information concerning the Columbia card, the second entry contains information concerning the Colomis card. To determine the status of either card, the status field should be interpreted in the following way:

- 00 = Card usable, no failures
- 01 = Card usable, but has failures
- 10 = Card is not installed
- 11 = Card is not usable due to critical failure

The **Panel VPD** may or may not be filled in on a particular system depending upon what type of panel is installed, to determine if the panel VPD information is valid, the **panel VPD usable bit** must be 1. If this field is 0, the panel VPD information is not valid.

Hex 0130 Network Attributes
(can be materialized and modified)
(only allowed in attribute selection value)

The Network Attributes is a template that contains information concerning APPN network attributes.

The materialize format of the Network attributes is as follows:

- Network Data — Char(190)
  - System name — Char(8)
  - System name length — Bin(2)

— New System name	Char(8)
— New System name length	Bin(2)
— Local system network identification	Char(8)
— Local system network identification length	Bin(2)
— End node data compression	Bin(4)
— Intermediate node data compression	Bin(4)
— Reserved	Char(2)
— Local system control point name	Char(8)
— Local system control point name length	Bin(2)
— Reserved	Char(10)
— Default local location name	Char(8)
— Default local location name length	Bin(2)
— Default mode name	Char(8)
— Default mode name length	Bin(2)
— Maximum number of intermediate sessions	Bin(2)
— Maximum number of conversations per APPN LUD	Bin(2)
— Local system node type	Char(1)
— Reserved	Char(1)
— Route addition resistance	Bin(2)
— List of network server network ID's (An array of five entries each 8 bytes in size)	Char(40)
— List of network server network ID lengths (An array of five entries each 2 bytes in size)	Char(10)
— List of network server control point names (An array of five entries each 8 bytes in size)	Char(40)
— List of network server control point name lengths (An array of five entries each 2 bytes in size)	Char(10)
— Alert flags	Char(1)
- Alert priority focal point	Bit 0
- Alert default focal point	Bits 1-7
- Reserved	Bit(6)
— Network Attribute Flags (Materializable only)	Char(1)
- Network attributes	Bit 0 initialized
- Pending system name system name	Bits 1 made current
- Reserved	Bit(2-7)

The **machine system name** is defaulted to the system serial number with a 'S' in the first position. Thereafter, it may be modified to any value of 1 through 8 characters with the first character alphabetic.

The machine system name length is kept to determine how long the system name is. The default value for the length is eight.

The **new system name** is a tentative new value chosen for the machine system name. This value will become the machine system name at the next IPL. The initial value is null and the syntax rules are the same as those for the machine system name.

The **new system name length** is kept to determine how long the new system name is. The default value for the length is zero.

The **local system network identification** default is 'APPN' and the default **local system network identification length** is four.

The **end node data compression** field controls whether the machine will allow data compression when it's an end node. This value is used when the mode description is equal to *NETATR. If one of the values listed in the table below is not specifed, then the value specified is equal to the maximum line speed that data should be compressed. Any configuration with a line speed slower than what is specified here will cause the data to be compressed. Valid values range from 1 bits-per-second through 2147483647.

0 = *NONE (default)
: No data compression will be done.

-1 = *REQUEST
: Data compression is requested on the session.

-2 = *ALLOW
: Data compression is allowed, but not requested for this session.

-3 = *REQUIRE
: Data compression is required on this session.

The **intermediate node data compression** field controls whether data compression will be requested by the machine when it's an intermediate node. This value is used when the mode description is equal to *NETATR. If one of the values listed in the table below is not specifed, then the value specified is equal to the maximum line speed that data should be compressed. Any configuration with a line speed slower than what is specified here will cause the data to be compressed. Valid values range from 1 bits-per-second through 2147483647.

0 = *NONE (default)
: No data compression will be done.

-1 = *REQUEST
: Data compression is requested on the session.

The **local system control point name** default is the system serial number with a character 'S' in the first position and the default **control point name length** is eight.

The **local location name** default is the system serial number with a character 'S' in the first position and the default **local location name length** is eight.

The **mode name** default is all blanks and the default **mode length** is eight.

The **maximum number of intermediate sessions** default is 200.

The **maximum number of conversations per APPN LUD** is 64.

The **local system node type** default is hex 01.

The **route addition resistance** default is 128.

All entries of the **network server network IDs** are defaulted to blanks with all entries of the **network server network IDs lengths** defaulting to zero.

All entries of the **network server control point names** are defaulted to blanks with all entries of the **network server control point name lengths** defaulting to zero.

Hex 0134 Date Format

The date format is the format is which the date will be presented to the customer. The possible values are YMD. MDY, DMY. JUL where Y = Year, M = Month, D = Day and JUL = Julian.

The format of the template for date format is as follows:

- Number of bytes available — Bin(4)
- Number of bytes provided — Bin(4)
- Date Format — Char(3)

Hex 0138 Leap Year Adjustment

The leap year adjustment is added to the leap year calculations to determine the year in which the leap should occur. The valid values are 0, 1, 2, 3.

The format of the template for leap year adjustment is as follows:

- Number of bytes available — Bin(4)
- Number of bytes provided — Bin(4)
- Leap year adjustment — Bin(2)

Hex 013C Timed Power On

The timed power on is the time and date at which the system should automatically power on if it is not already powered on.

The format of the template for timed power on is as follows:

- Number of bytes available — Bin(4)
- Number of bytes provided — Bin(4)
- Minute — Bin(2)
- Hour — Bin(2)
- Day — Bin(2)
- Month — Bin(2)
- Year — Bin(2)

Hex 0140 Timed Power On Enable/Disable

The timed power on enable/disable allows the timed power on function to be queried to determined if the function is enabled or disabled.

The format of the template for timed power on enable/disable is as follows:

- Number of bytes available — Bin(4)
- Number of bytes provided — Bin(4)
- Enable/Disable — Bin(2)

  Hex 8000-indicates timed power on is enabled

  Hex 0000-indicates timed power on is disabled

Hex 0144 Remote Power On Enable/Disable

The remote power on enable/disable allows the remote power on function to be queried to determined if the function is enabled or disabled.

The format of the template for remote power on enable/disable is as follows:

- Number of bytes available — Bin(4)

- Number of bytes provided — Bin(4)
- Enable/Disable — Bin(2)

  Hex 8000-indicates remote power on is enabled

  Hex 0000-indicates remote power on is disabled

Hex 0148 Auto power restart Enable/Disable

The auto power restart enable/disable allows the auto power restart function to be queried to determined if the function is enabled or disabled.

The format of the template for auto power restart enable/disable is as follows:

- Number of bytes available — Bin(4)
- Number of bytes provided — Bin(4)
- Enable/Disable — Bin(2)

  Hex 8000-indicates auto power restart is enabled

  Hex 0000-indicates auto power restart is disabled

Hex 014C Date separator

The date separator is used when the date is presented to the customer. The valid values are a slash(/), dash(-), period(.), comma(,) and a blank( ).

The format of the template for the date separator is as follows:

- Number of bytes available — Bin(4)
- Number of bytes provided — Bin(4)
- Date Separator — Char(1)

Hex 0164 Uninterruptible power supply type

Note: The *UPS type* is meaningful only if a UPS is installed.

The uninterruptible power supply type option allows the MI user to tell the machine how much of the system is powered by a UPS (ie, what type of UPS is installed). A *full UPS* will power all racks in the system. A *mini UPS* will power the racks containing the CEC and the load source.

The format of the template for UPS Type is as follows:

- Number of bytes available — Bin(4)
- Number of bytes provided — Bin(4)
- UPS Type — Bin(2)

  Hex 0000-indicates a full UPS is installed (all racks have a UPS installed)

  Hex 8000-indicates a mini UPS is installed (only the minimum number of racks are powered)

Hex 0168 Panel Status Request

The Panel Status Request option allows BOSS to determine what current status of the operations panel.

The format of the template for Panel Status Request is as follows (including the usual 8-byte prefix):

- Number of bytes available — Bin(4)
- Number of bytes provided — Bin(4)
- Current IPL type — Char(1)

• Panel status	Char(2)
— Uninterrupted power supply installed 0 = UPS not installed 1 = UPS installed, ready for use	Bit 0
— Utility power failed, running on UPS 0 = Running on utility power 1 = Running on UPS	Bit 1
— Uninterrupted power supply (UPS) bypass active 0 = UPS bypass not active 1 = UPS bypass active	Bit 2
— Uninterrupted power supply (UPS) battery low 0 = UPS battery not low 1 = UPS battery low	Bit 3
— Auto keylock position 0 = Keylock not in auto position 1 = Keylock in auto position	Bit 4
— Normal keylock position 0 = Keylock not in normal position 1 = Keylock in normal position	Bit 5
— Manual keylock position 0 = Keylock not in manual position 1 = Keylock in manual position	Bit 6
— Secure keylock position 0 = Keylock not in secure position 1 = Keylock in secure position	Bit 7
— Reserved	Bits 8-15
• Reserved	Char(5)
• Most recent IPL type	Char(1)

The **current IPL type** is the state of the IPL type at the operations panel. Possible values are A, B, C, D.

UPS installed indicates that a Uninterrupted Power Supply is installed on the system and is available for use should the power fail.

UPS power failed indicates that a utility failure has occurred and the system is currently running on battery power.

UPS bypass active indicates that the UPS has been bypassed. If a utility power failure occurs, the UPS will not supply power.

UPS battery low indicates that a UPS battery is installed on the system and the battery is low.

Auto keylock position indicates that the keylock is currently in the auto position on the operation panel.

Normal keylock position indicates that the keylock is currently in the normal position on the operation panel.

Manual keylock position indicates that the keylock is currently in the manual position on the operation panel.

Secure keylock position indicates that the keylock is currently in the secure position on the operation panel.

The **most recent IPL type** is the type of IPL that was performed on the most recent IPL. Possible values are A, B, C, D.

Hex 016C Extended machine initialization status record

The XMISR (extended machine initialization status record) is used to report the status of the machine.

The materialize format of the XMISR is as follows:

- Number of bytes available — Bin(4)
- Number of bytes provided — Bin(4)
- Save storage status — Char(4)
  - Checksumming status — Bit 0

    0 = Checksumming was not stopped
    1 = Checksumming was stopped
  - Completion status — Bit 1

    0 = Save storage did not complete
    1 = Save storage completed
  - System restored status — Bit 2

    0 = Save storage did not restore the system
    1 = Save storage restored the system
  - Save storage attempted — Bit 3

    0 = Save storage not attempted
    1 = Save storage was attempted
  - Unreadable sectors — Bit 4

    0 = Unreadable sectors were not found
    1 = Unreadable sectors were found during save operation
  - Check for active files on save storage media — Bit 5

    0 = Do not check for active files on save storage media
    1 = Check for active files on save storage media
  - Reserved — Bits 6-31
- Save storage information — Char(118)
  - Tape device information — Char(18)
    - Number of tape device entries — UBin(2)
      - Tape device address — Char(16)
      - (Array of four entries, each 4 bytes in size)
        - Tape device IOP address — Char(2)

— Tape device device address	Char(2)
- Tape volume names structure	Char(62)
• Number of tape volume entries	UBin(2)
• Tape volume names	Char(60)
• (Array of ten entries, each 6 bytes in size)	
- Tape expiration date	Char(6)
- Bad sector count	Char(4)
- Date from save tape	Char(6)
- Time last successful save started	Char(8)
- Reserved	Char(14)
• Install tape Volume ID	Char(6)
• IPL sequence number ID	Bin(4)
• Physical address of tape device used for last D-type IPL	Char(4)

Hex 0170 Alternate initial process definition template

The alternate initial process definition template is used by the machine when performing an automated install.

No check is made and no exception is signaled if the values in the template are invalid; however, the next automated install will not be successful.

Hex 0178 Hardware storage protection state

Note: Hardware storage protection is meaningful only on version 2 hardware or later.

The hardware storage protection state indicates if the machine will honor the protection state of objects on the system. When it is active, references by user state programs to protected objects will result in *object access denied* (hex 4401) exceptions. When hardware storage protection is inactive, or when running on version 1 hardware, accesses of protected objects by user state programs will go undetected.

The format of the template for hardware storage protection state option is as follows (including the usual 8-byte prefix):

• Number of bytes available	Bin(4)
• Number of bytes provided	Bin(4)
• Hardware storage protection state	Bin(2)

Hex 0000-indicates hardware storage protection is inactive

Hex 8000-indicates hardware storage protection is active

Hex 0180 Time separator

The time separator is used when the time is presented to the customer. The valid values are a colon(:), period(.), comma(,) and a blank( ).

The format of the template for the time separator is as follows:

• Number of bytes available	Bin(4)
• Number of bytes provided	Bin(4)
• Time separator	Char(1)

Hex 0184 Software Error Logging

The software error logging machine attribute is used to allow the MI user to determine whether or not software error logging is active for the machine

The format of the template for software error logging is as follows:

- Number of bytes available — Bin(4)
- Number of bytes provided — Bin(4)
- Software error logging — Bin(2)

  Hex 8000-indicates software error logging is active

  Hex 0000-indicates software error logging is not active

Hex 0188 Machine task termination event control option (can be materialized and modified).

The machine task termination event option controls if the machine will signal a machine wide event when machine tasks terminate. The default, which is established every IPL, is to not signal machine task termination events. The machine task termination event id is hex 0016,05,01.

The format of the template for the machine task termination event option is as follows:

- Number of bytes available — Bin(4)
- Number of bytes provided — Bin(4)
- VLIC task termination event option — Bin(2)

  Hex 8000-indicates machine task termination events will be signaled.

  Hex 0000-indicates machine task terminations events will not be signaled.

Limitations: Data-pointer-defined scalars are not allowed as a primary operand for this instruction. An invalid operand type exception is signaled if this occurs.

Exceptions

Exception		Operands 1	2	Other
06	Addressing			
	01 space addressing violation	X	X	
	02 boundary alignment	X	X	
	03 range	X	X	
	06 optimized addressability invalid	X	X	
10	Damage encountered			
	04 system object damage state	X	X	X
	44 partial system object damage	X	X	X
1C	Machine-dependent exception			
	03 machine storage limit exceeded			X
	0A service processor unable to process request			X
20	Machine support			
	02 machine check			X
	03 function check			X

Exception		Operands 1	Operands 2	Other
22	Object access			
	01 object not found	X	X	
	02 object destroyed	X	X	
	03 object suspended	X	X	
	08 object compressed			X
24	Pointer specification			
	01 pointer does not exist	X	X	
	02 pointer type invalid	X	X	
2E	Resource control limit			
	01 user profile storage limit exceeded			X
32	Scalar specification			
	01 scalar type invalid	X	X	
	02 scalar attributes invalid		X	
	03 scalar value invalid		X	
36	Space management			
	01 space extension/truncation			X
38	Template specification			
	03 materialization length exception	X		

Materialize Machine Data (MATMDATA)

Op Code (Hex)	Operand 1	Operand 2
0522	Receiver	Materialization options

Operand 1: Character variable scalar (fixed length)

Operand 2: Character(2) or unsigned binary(2) scalar or immediate (fixed length)

ILE access

```
MATMDATA (
      receiver                : space pointer;
      materialization_options : aggregate
)

OR

MATTOD (
 var time_of_day : aggregate
)
```

Description: This instruction makes available the unique values of machine data. The values of various machine data are placed in the receiver.

Operand 2 is a 2-byte value. The value of operand 2 determines which machine data are materialized. Operand 2 is restricted to a constant character or unsigned binary scalar or an immediate value.

- Materialization options — Char(2)

  Hex 0000 = Materialize time of day clock
  Hex 0001 = Materialize system parameter integrity validation flag
  Hex 0002 through FFFF Reserved

Operand 1 specifies a receiver into which the materialized data is placed. It must specify a character scalar with a minimum length which is dependent upon the materialization option specified for operand 2. The reciever may be substringed. The start position of the substring may be a variable. However, the length of the substring must be an immediate or constant. If the length specified for operand 1 is less than the required minimum, an exception is signaled. Only the bytes up to the required minimum length are used. Any excess bytes are ignored.

The data placed into the receiver differs depending upon the materialization option specified. The following descriptions detail the formats of the optional materializations.

- Hex 0000 = Materialize time of day clock
  minimum receiver length is 8

  Time-of-day clock — Char(8)

  The time-of-day clock provides a consistent measure of elapsed time. The maximum elapsed time the clock can indicate is approximately 143 years.

  The time-of-day clock is a 64-bit unsigned binary counter with the following format:

  0................41 42 reserved 63

  The bit positions of the clock are numbered from 0 to 63.

The clock is incremented by adding a 1 in bit position 41 every 1024 microseconds. Bit positions 42 through 63 are used by the machine and have no special meaning to the user. Note that these bits (42-63) may contain either binary 1's or binary 0's.

Unpredictable results occur if the time of day is materialized before it is set.

The maximum unsigned binary value that the time of day clock can be modified to contain is hex DFFFFFFFFFFFFFFF.

- Hex 0001 = Materialize system parameter integrity validation flag
  minimum receiver length is 1

System parameter integrity validation flag	Char(1)

  This option returns the value of the machine attribute which specifies whether additional validation of parameters passed to programs which run when the process is in system state is to be performed, such as for U. S. government's Department of Defense security ratings.

  A value of hex 01 indicates this additional checking is being performed. A value of hex 00 is returned otherwise.

Performance note: The time of day clock and the system parameter integrity validation flag may be materialized both with this instruction and also with the Materialize Machine Attributes instruction. The performance of this instruction is considerably better.

Exceptions

Exception		Operands 1	2	Other
06	Addressing			
	01 space addressing violation	X		
	02 boundary alignment	X		
	03 range	X		
	06 optimized addressability invalid	X		
08	Argument/parameter			
	01 parameter reference violation	X		
10	Damage encountered			
	04 system object damage state	X		
	44 partial system object damage	X		
1C	Machine-dependent exception			
	03 machine storage limit exceeded	X		
20	Machine support			
	02 machine check			X
	03 function check			X
22	Object access			
	01 object not found	X		
	02 object destroyed	X		
	03 object suspended	X		
24	Pointer specification			
	01 pointer does not exist	X		

Exception		Operands 1	2	Other
	02 pointer type invalid	X		
2E	Resource control limit			
	01 user profile storage limit exceeded			X
32	Scalar specification			
	01 scalar type invalid	X	X	
	02 scalar attributes invalid	X	X	
	03 scalar value invalid		X	
36	Space management			
	01 space extension/truncation			X

Instruction support interfaces

© Copyright IBM Corp. 1991, 1993

Chapter 23. Exception Specifications

This chapter describes the exceptions which can be signaled by the machine. Exception generation is the only facility for synchronously communicating error conditions that are a direct result of AS/400 instruction processing. Machine exceptions identify error conditions that require processing before the next sequential AS/400 instruction is executed. Instructions that cause a particular exception may not function identically before execution is stopped; however, each instruction produces consistent results. These results ensure machine integrity and reliability. The results are inherent in a particular exception definition or in the detailed instruction definition.

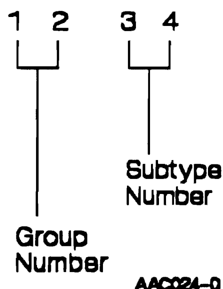
The user can monitor any number of exceptions. There are three basic techniques for the user to handle an exception. One technique is to provide detailed handling specified by a program defined exception description object. The second technique is to provide a default exception handler for the process. This exception handler is invoked whenever an invocation fails to handle an exception. The third technique is to accept the machine default of process termination by not providing an appropriate exception handling mechanism.

© Copyright IBM Corp. 1991, 1993

Machine Interface Exception Data

Exception data is communicated across the machine interface through a Retrieve Exception Data instruction. Certain information is available for all exceptions when an appropriate exception description has been defined by the user. That information includes the following:

- Exception identification-This is a 2-byte hexadecimal field formed by concatenating to the high-order 1-byte exception group number a low-order 1-byte exception subtype number. The format of the exception identification is as follows:



- Compare value length
- Compare value (machine signaled have a compare value of hex 00000000 with a length of 4)
- Exception-specific data
- Signaling program invocation address
- Signaled program invocation address
- Signaling program instruction address
- Signaled program instruction address
- Machine-dependent data identifying the component that generated the exception

The exception-specific data provides additional pointers and data that may be required for an individual exception.

Exception List

The following is a list of all exceptions in alphabetic and numeric order by group. The subtypes within each group are in numeric order.

02 Access Group

- 01 Object ineligible for access group

04 Access State

- 01 Access state specification invalid

06 Addressing

- 01 Space addressing violation
- 02 Boundary alignment
- 03 Range
- 04 External data object not found
- 05 Invalid space reference
- 06 Optimized addressability invalid

08 Argument/Parameter

- 01 Parameter reference violation
- 02 Argument list length violation
- 03 Argument list length modification violation

0A Authorization

- 01 Unauthorized for operation
- 02 Privileged instruction
- 03 Attempt to grant/retract authority state to an object that is not authorized
- 04 Special authorization required
- 05 Create/modify user profile beyond level of authorization
- 06 Grant/retract authority invalid.

0C Computation

- 01 Conversion
- 02 Decimal data
- 03 Decimal point alignment
- 04 Edit digit count
- 05 Edit mask syntax
- 06 Floating-point overflow
- 07 Floating-point underflow
- 08 Length conformance
- 09 Floating-point invalid operand
- 0A Size
- 0B Zero divide
- 0C Invalid floating-point conversion

- 0D Floating-point inexact result
- 0E Floating-point zero divide
- 0F Master key not defined
- 10 Weak key not valid
- 11 Key parity invalid
- 12 Invalid extended character data
- 13 Invalid extended character operation
- 14 Invalid compressed data
- 15 Date boundary overflow
- 16 Date format error
- 17 Date value error
- 18 Date boundary underflow
- 19 Space pointer operands do not point to the same space object

0E Context Operation

- 01 Duplicate object identification
- 02 Object ineligible for context

10 Damage Encountered

- 02 Machine context damage state
- 04 System object damage state
- 05 Authority verification terminated due to damaged object
- 44 Partial system object damage state

12 Data Base Management

- 01 Conversion mapping error
- 02 Key mapping error
- 03 Cursor not set
- 04 Data space entry limit exceeded
- 05 Data space entry already locked
- 06 Data space entry not found
- 07 Data space index invalid
- 08 Incomplete key description
- 09 Duplicate key value in existing data space entry
- 0A End of path
- 0B Duplicate key value detected while building unique data space index
- 0D No entries locked
- 0F Duplicate key value in uncommitted data space entry
- 13 Invalid mapping template
- 14 Invalid selection template
- 15 Data space not addressed by index

- 16 Data space not addressed by cursor
- 17 Key changed since set cursor
- 18 Invalid key value modification
- 19 Invalid rule option
- 1A Data space entry size exceeded
- 1B Logical space entry size limit exceeded
- 1C Key size limit exceeded
- 1D Logical key size limit exceeded
- 21 Unable to maintain a unique key data space index
- 25 Invalid data base operation
- 26 Data space index with invalid floating-point field build termination
- 27 Data space index key with invalid floating-point field
- 30 Specified data space entry rejected
- 32 Join value changed
- 33 Data space index with non-user exit selection routine build termination
- 34 Non-user exit selection routine failure
- 36 No mapping code specified
- 37 Operation not valid with join cursor
- 38 Derived field operation error
- 39 Derived field operation error during build index
- 40 Invalid entry definition table
- 41 ISV parameter value in runtime data pointer array not correct
- 42 Unique fanout join failed
- 43 DDAT had an error
- 62 Parent index cannot be used to create new index

16 Exception Management

- 01 Exception description status invalid
- 02 Exception state of process invalid
- 03 Invalid invocation address
- 04 Resume/retry invalid
- 05 No inquiry message found for reply message

18 Independent Index

- 01 Duplicate key argument in index

1A Lock State

- 01 Invalid lock state
- 02 Lock request not grantable
- 03 Invalid unlock request
- 04 Invalid object lock transfer request

05 Invalid space location unlock

1C Machine-Dependent Exception

01 Machine-dependent request invalid

02 Program limitation exceeded

03 Machine storage limit exceeded

04 Object storage limit exceeded

06 Lock limit exceeded

07 Modify main storage pool controls invalid

08 Requested function not valid

09 Auxiliary storage pool number invalid

0A Service Processor unable to process request

1E Machine Observation

01 Program not observable

02 Invocation not found

03 Machine storage limit exceeded

04 DBGINT error

05 DBGINT error

20 Machine Support

01 Diagnose

02 Machine check

03 Function check

22 Object Access

01 Object not found

02 Object destroyed

03 Object suspended

04 Object not eligible for operation

05 Object not available to process

06 Object not eligible for destruction

07 Authority verification terminated due to destroyed object

08 Object Compressed

0A Program not eligible for operation

24 Pointer Specification

01 Pointer does not exist

02 Pointer type invalid

03 Pointer addressing invalid object

04 Pointer not resolved

26 Process Management

02 Queue full

28 Process State

- 01 Process ineligible for operation
- 02 Process control space not associated with a process
- 0A Process attribute modification invalid

2A Program Creation

- 01 Program header invalid
- 02 ODT syntax error
- 03 ODT relational error
- 04 Operation code invalid
- 05 Invalid op code extender field
- 06 Invalid operand type
- 07 Invalid operand attribute
- 08 Invalid operand value range
- 09 Invalid branch target operand
- 0A Invalid operand length
- 0B Invalid number of operands
- 0C Invalid operand ODT reference
- 0D Reserved bits are not zero
- 10 Automatic storage for procedure exceeds maximum
- 11 Machine automatic storage exceeds maximum
- 12 Data type or length of initial value not valid
- 14 Static data initialized to address of automatic data
- 15 Initial value for static data not valid
- 16 Number of procedures exceeds maximum allowed
- 17 Type table entry not valid
- 18 Alias table entry not valid
- 19 Size of constants exceeds maximum
- 1A Procedure size exceeds maximum
- 1B Instruction stream not valid
- 1C Size of literals exceeds maximum
- 1D Dictionary entry not valid
- 1E Level of machine interface not supported on target release
- 1F Size of dictionary exceeds maximum
- 20 Internal machine operation not valid
- 21 Size of internal binding table exceeds maximum
- 5E An error was detected in a static storage definition or initalization
- 5F Overlapping initializations not valid
- 60 Dictionary ID is not valid

61	Binding specification value not valid
62	Copyright component value not valid
63	Module limitation exceeded
A0	Attempt to delete part that may not be deleted
B0	Object list referential extension not valid
B1	Symbol resolution list referential extension not valid
B2	Service program export list referential extension not valid
B3	Secondary associated spaces list referential extesion not valid
B4	Program limitation exceeded
C0	Attempt to delete part that may not be deleted

2C Program Execution

01	Return instruction invalid
02	Return point invalid
03	<D>
04	Branch target invalid
05	Activation in use by invocation
06	Instruction cancellation
07	Instruction termination
08	Branch target defined by label pointer not valid
10	Process object destroyed
11	Process object access invalid
12	Activation group access violation
13	Activation group not found
14	Activation group in use
15	Invalid operation for program
16	Program activation not found
17	Default activation group not destroyed
18	Invalid source invocation
19	Invalid origin invocation
1A	Invocation offset outside range of current stack
1B	Invocation not eligible for operation
1C	Instruction not valid for invocation type
1D	automatic storage overflow
1E	activation access violation
1F	program signature violation
20	static storage overflow
21	program import invalid
22	data reference invalid

| 23 imported object invalid
| 24 activation group export conflict
| 25 import not found

2E Resource Control Limit
- 01 User profile storage limit exceeded
- 02 Security audit journal full or other failure

30 Journal
- 01 Apply journal changes failure
- 02 Entry not journaled
- 03 Maximum objects through a journal port limit exceeded
- 04 Invalid journal space
- 05 Maximum journal spaces attached
- 06 Journal space not at a recoverable boundary
- 07 Journal ID not unique
- 08 Object already being journaled
- 09 Transaction limit list exceeded
- 0A Data space index currently journaled
- 0B Data space index currently in force mode
- 0C Underlying data space not journaled to same journal

32 Scalar Specification
- 01 Scalar type invalid
- 02 Scalar attributes invalid
- 03 Scalar value invalid

34 Source/Sink Management
- 01 Source/sink configuration invalid
- 02 Source/sink physical address invalid
- 03 Source/sink object state invalid
- 04 Source/sink resource not available

36 Space Management
- 01 Space extension/truncation
- 02 Invalid space modification

38 Template Specification
- 01 Template value invalid
- 02 Template size invalid
- 03 Materialization length exception

3A Wait Time-Out
- 01 Dequeue
- 02 Lock

03 Wait on event

04 Space location lock wait

3C Service

01 Invalid service session state

02 Unable to start service session

3E Commitment Control

01 Invalid commit block status change

03 Commit block is attached to process

04 Commit block controls uncommitted changes

06 Commitment control resource limit exceeded

08 Object under commitment control being incorrectly journaled

10 Operation not valid under commitment control

11 Process has attached commit block

12 Objects under commitment control

13 Commit block not journaled

14 Errors during decommit

15 Object ineligible for commitment control

16 Object ineligible for removal from commitment control

40 Dump Space Management

01 Dump data size limit exceeded„

02 Invalid dump data insertion

03 Invalid dump space modification

04 Invalid dump data retrieval

44 Domain Violation

01 Object Domain error

45 Heap Space

01 Heap identifier invalid for the current activacation group

02 The requested heap space operation is invalid

03 The heap space has reached it's maximum allowable size

04 The requested heap space allocation or reallocation size is invalid

05 Heap space destroyed

06 Invalid heap space condition detected

07 The supplied mark identifier is invalid

46 Queue Space

01 Queue Space not associated with the process

02 Cannot modify queue space

03 Invalid message reference key

04 Queue Space not eligible for destruction

02 Access Group

0201 Object Ineligible for Access Group

An attempt was made to insert an object into an access group. The operation could not be performed for one of the following reasons:

- The object is temporary, or the object is permanent and the access group is temporary.
- The object is restricted by the machine from membership in an access group.

Information Passed:

• Access group	System pointer
• Object to be inserted (binary 0 for objects not yet created)	System pointer

Instructions Causing Exception:

- Any create instruction that specifies an access group in the create template
- Signal Exception

04 Access State

0401 Access State Specification Invalid

An access state not supported by the machine was specified for an object.

Information Passed

- The invalid access state Char(1)

Instructions Causing Exception:

- Set Access State
- Signal Exception

06 Addressing

0601 Space Addressing Violation

An attempt has been made to operate outside the current extent of a space.

Information Passed:

Note: IPS stands for "implicit process space".

- System object affiliated with the space — System pointer
- Space offset reference attempted — Bin(4)
  This value may be zero when not available.
- Space class — Char(1)

  Hex 00 = Primary associated space (includes space objects)
  Hex 01 = Secondary associated space
  Hex 02 = IPS used for automatic storage
  Hex 03 = IPS used for static storage
  Hex 04 = IPS used for heap storage
- Reserved (binary 0) — Char(1)
- Secondary associated space number — UBin(2)
  This value will be zero when not available, such as when the space is not a secondary associated space.
- Activation group mark — UBin(4)
  This value will be zero when not available, such as when the space is not an implicit process space.
- Heap space identifier — UBin(4)
  This value will be zero when not available, such as when the space is not an implicit process space used by a heap space.
- Pointer to start of IPS used for allocation — Space pointer
  This value will be zero when not available, such as when the space is not an implicit process space.

Different information is supplied for different space classes. Fields which are not used for a given space class will contain zero values. Here is a summary of the information returned (i.e. the field values set) for each space class.

hex 00 (primary associated space, including space objects)
- system pointer to the object
- offset
- space class

hex 01 (secondary associated space)
- system pointer to the object
- offset
- space class
- secondary space number

hex 02 (auto)
- system pointer to the Process Control Space
- offset
- space class
- activation group mark
- space pointer to the start of the IPS from which the allocation was taken

hex 03 (static)
- system pointer to the Process Control Space
- offset

- space class
- activation group mark
- space pointer to the start of the IPS from which the allocation was taken

hex 04 (heap)

- system pointer to the Process Control Space
- offset
- space class
- activation group mark
- heap space identifier
- space pointer to the start of the IPS from which the allocation was taken.
  - Note: If the heap space was created to force a new IPS on each allocation this field will contain a space pointer to the start of the allocation.

Instructions Causing Exception:

- Any instruction using a pointer or scalar as an operand
- Any instruction using a scalar as an index, a length suboperand, or a space pointer as a base suboperand
- Signal Exception

0602 Boundary Alignment

A program object has been referenced, and it does not have the proper alignment relative to the beginning of a space. Pointers must always be 16-byte aligned. Program objects that are not pointers must have at least the alignment specified by the ODT entry.

Information Passed:

- Addressability to pointer or template — Space pointer

Instructions Causing Exception:

- Any instruction having a pointer operand or a template operand that requires a specific boundary alignment

0603 Range

A subscript value in a compound operand array reference is outside the range defined for the array. A subscript value of less than 1 or greater than the number of elements defined by the array causes this exception.

A reference to a string has a position and/or length that exceeds the bounds of the string. A compound operand that defines a character string that does not completely fall within the bounds of the base character string was referenced. A substring with position (P) e1 and length (L) e1 does not meet the following constraint (n is the length of the base string):

$P + L - 1 \leq n$

Instructions Causing Exception:

- All instructions that use scalar or pointer operands
- Signal Exception

0604 External Data Object Not Found

An unsuccessful attempt was made to resolve a data pointer. The external data object specified by the initial value of the data pointer was not found in the process activations. If a program name was specified in the symbolic address, then only that program's activation is considered for resolution. If no program was specified, then all of the programs with activations in the process are considered for data pointer resolution.

Information Passed:

- External data object name Char(32)

Instructions Causing Exception:

- Any instruction that references an external data object through a data pointer.
- Any instruction where a data pointer is used as the scalar value for an index of a length suboperand. This includes scalar and pointer operands that may be subscripted.
- Signal Exception
- Compare Pointer Addressability
- Compare Pointer for Space Addressability
- Convert Character to Numeric
- Convert External Form to Numeric
- Convert Numeric to Character
- Copy Bytes Left Adjusted
- Copy Bytes Left Adjusted With Pad
- Copy Bytes Right Adjusted
- Copy Bytes Right Adjusted With Pad
- Copy Numeric Value
- Edit
- Materialize Pointer
- Resolve Data Pointer
- Set Data Pointer Addressability
- Set Data Pointer Attributes
- Set Space Pointer From Pointer
- Set System Pointer From Pointer

0605 *Invalid Space Reference*

An attempt was made to address a space contained in an object that has no space.

Instruction Causing Exception:

- Set Space Pointer from Pointer

0606 *Optimized Addressability Invalid*

An instruction attempted to use the internally optimized value of a space pointer that was invalid due to the failure of a prior instruction in trying to access the pointer's value.

The machine may optimize the retrieval of a pointer's value by using the value retrieved on one instruction for use by multiple instructions that have need to reference the pointer's value. This avoids the overhead of continually retrieving the pointer's value from storage for every instruction that would have need to use it. If, in attempting to retrieve the pointer's value, an exception is detected, the machine marks the internally optimized value as invalid. This is done to provide for detecting the invalid addressability upon subsequent execution of instructions that depend on the internally optimized value. These instructions have no provision for retrieving the pointer's value from storage. These instructions will not redetect the original exception, but instead detect the optimized addressability invalid exception for this condition. This condition can occur when an exception is detected during an attempt to retrieve a pointer's value and the exception is ignored which causes execution of the program to continue without successfully retrieving the pointer's value.

This exception may not be detected on certain cases of internal machine optimizations that may be performed on references to space pointer machine objects. A reference to the space data addressed by the pointer is necessary to ensure consistent detection of this exception. Although the exception may not be detected for initial operations, it will be detected on any subsequent operation that references the space data addressed by the space pointer machine objects.

The optimization of retrieving a pointer's value can be prevented by specifying the abnormal attribute for the pointer.

This exception may not be detected on the operations listed below under certain cases of internal machine optimizations which may be performed on references to space pointer machine objects. The operations listed below refer to the value of a space pointer machine object, but do not have need to reference the space data the pointer addresses. A reference to the space data addressed by the pointer is necessary to insure consistent detection of this exception. Although the exception may not be detected for these operations, it will be detected upon any subsequent operation which references the space data addressed by the space pointer machine object.

The following instructions may not detect this exception upon references to a space pointer machine object.

- Add Space Pointer
- Compare Pointer for Space Addressability
- Compare Pointers for Equality
- Compare Space Addressability
- Set Space Pointer
- Set Space Pointer with Displacement
- Set Space Pointer from Pointer
- Subtract Space Pointer Offset
- Subtract Space Pointers For Offset

See the particular instruction description for more detail.

Instructions Causing Exception:

- Any instruction using a pointer or scalar as an operand
  - Signal Exception

08 Argument/Parameter

0801 Parameter Reference Violation

An attempt was made to reference an internal or an external parameter for which no corresponding argument was passed.

This exception may not be signaled for operations which refer to the value of a space pointer machine object, but which do not attempt to reference the space data the pointer addresses. The following instructions may not signal this exception upon references to a space pointer machine object.

- Add Space Pointer
- Compare Pointers for Equality
- Compare Space Addressability
- Set Space Pointer
- Set Space Pointer with Displacement
- Set Space Pointer from Pointer
- Subtract Space Pointer Offset
- Subtract Space Pointers For Offset

See the particular instruction description for more detail. *Instructions Causing Exception:*

- Any instruction that references a parameter operand
- Signal Exception

0802 Argument List Length Violation

An argument list does not properly correspond to the length required by the parameter list.

Information Passed:

- Minimum number of arguments allowed Char(2)
- Maximum number of arguments allowed Char(2)
- Actual number of arguments allowed Char(2)

Instructions Causing Exception:

- Call External
- Transfer Control
- Signal Exception

0803 Argument List Length Modification Violation

An attempt was made to change the length of a variable-length argument list to a value less than 0 or greater than the maximum size of the argument list.

Instructions Causing Exception:

- Set Argument List Length
- Signal Exception

0C Computation

0C01 Conversion

A scalar value cannot be converted to the necessary type in this instruction.

Instructions Causing Exception:

- Convert Character to Hex
- Convert External Form to Numeric
- Convert SNA to Character
- Convert MRJE to Character
- Convert BSC to Character
- Signal Exception

0C02 Decimal Data

The sign or digit codes of a decimal operand, either packed or zoned, contain an invalid value. For packed and zoned format, either the sign is outside the valid range of A through F or a digit field is outside the range 0 through 9.

Instructions Causing Exception:

- Add Numeric
- Compare Numeric Value
- Convert Character to Numeric
- Convert Decimal Form to Floating-Point
- Convert Numeric to Character
- Copy Numeric Value
- Divide
- Divide With Remainder
- Edit
- Extract Magnitude
- Multiply
- Negate
- Remainder
- Scale
- Search
- Subtract Numeric
- Sum
- Signal Exception

0C03 Decimal Point Alignment

The value of a numeric operand cannot be aligned in a 31-digit decimal field. Decimal point alignment was attempted by padding with 0's on the right. Nonzero digits would have to be truncated on the left to fit the aligned value into a 31-digit decimal field.

Instructions Causing Exception:

- Add Numeric

- Compare Numeric Value
- Subtract Numeric
- Signal Exception

0C04 Edit Digit Count

The number of digit position characters in the mask operand of an Edit instruction is not equal to the number of digits in the source operand value.

Instructions Causing Exception:

- Edit
- Signal Exception

0C05 Edit Mask Syntax

The characters of the mask operand do not follow the valid syntax rules for an Edit instruction.

Instructions Causing Exception:

- Edit
- Signal Exception

0C06 Floating-Point Overflow

The result of a floating-point operation is finite and not an invalid value, but its exponent is too large for the target floating-point format. The signed exponent has exceeded 127 in short format or 1023 in long format.

Information Passed:

• Floating-point value attributes	Char(1)
— Normal bias	Bit 0
— Modified bias	Bit 1
— Rounded to short floating-point precision	Bit 2
— NaN	Bit 3
— Reserved (binary 0)	Bits 4-7
• Reserved (binary 0)	Char(7)
• Overflowed floating-point value	Floating-(8) point
• Reserved (binary 0)	Char(16)

Instructions Causing Exception:

- Add Numeric
- Compute Math Function Using One Input Value
- Compute Math Function Using Two Input Values
- Convert Character to Numeric
- Convert Numeric to Character
- Convert Decimal Form to Floating-Point
- Copy Numeric Value
- Divide
- Extract Magnitude
- Multiply

- Negate
- Scale
- Subtract Numeric
- Signal Exception

0C07 Floating-Point Underflow

The result of a floating-point operation is not zero but has too small an exponent for the destination's format without being denormalized. The signed exponent is less than -126 in short format or less than -1022 in long format.

Information Passed:

• Floating-point value attributes	Char(1)
— Normal bias	Bit 0
— Modified bias	Bit 1
— Rounded to short floating-point precision	Bit 2
— NaN	Bit 3
— Reserved (binary 0)	Bits 4-7
• Reserved (binary 0)	Char(7)
• Underflowed floating-point value	Floating-(8) point
• Reserved (binary 0)	Char(16)

Instructions Causing Exception:

- Add Numeric
- Compute Math Function Using One Input Value
- Compute Math Function Using Two Input Values
- Convert Character to Numeric
- Convert Numeric to Character
- Convert Decimal Form to Floating-Point
- Copy Numeric Value
- Divide
- Extract Magnitude
- Multiply
- Negate
- Scale
- Subtract Numeric
- Signal Exception

0C08 Length Conformance

The operand lengths or resultant value length or both do not conform to the rules of the instruction:

CVTHC   Twice the length of the source operand must be less than or equal to the length of the receiver operand.

CVTCH	The length of the operand must be less than or equal to twice the length of the receiver operand.
CVTMC	The length of a record in the receiver was not enough to contain the converted form of a record from the source.
EDIT	The length of the resultant edited value must be equal to the length of the receiver operand.
SCAN	The length of the compare operand must not be greater than the length of the base string.
SEARCH	The length of the find operand plus the value of the location operand must be less than or equal to the length of an element of the array operand.
XLATE	The source and receiver operands must be the same length.

Instructions Causing Exception:

- Convert Character to Hex
- Convert Hex to Character
- Convert MRJE to Character
- Edit
- Extended Character Scan
- Scan
- Search
- Signal Exception
- Translate

0C09 Floating-Point Invalid Operand

A floating-point invalid operand condition is caused by one of the following conditions:

- A source operand is an unmasked NaN.
- Addition of infinities of different signs and subtraction of infinities of the same sign.
- Multiplication of zero times infinity.
- Division of zero by zero or infinity by infinity.
- Computing the sine, cosine, or tangent function for infinity.
- Computing the arc tangent, exponential, logarithm, square root, or power function for infinity when in projective infinity mode.
- Floating-point values compared unordered and no branch or indicator options are specified for the unordered, negation of unordered, equal, or negation of equal conditions on compare numeric value.
- An unordered resultant condition occurred on a computational instruction because the result was NaN, and branch or indicator conditions are specified but unordered, negation of unordered, zero, or negation of zero conditions are not specified.

Information Passed:

- Exception type — Char(1)

  Hex 00 = Invalid arithmetic operation or operand is unmasked NaN.
  Hex 01 = Invalid branch or indicator conditions.

  Hex 02 through hex FF are reserved.

- Reserved (binary 0) — Char(31)

Instructions Causing Exception:

- Add Numeric
- Compare Numeric Value
- Compute Math Function Using One Input Value
- Compute Math Function Using Two Input Values
- Convert Character to Numeric
- Convert Floating-Point to Decimal Form
- Convert Numeric to Character
- Copy Numeric Value
- Divide
- Extract Magnitude
- Multiply
- Negate
- Scale
- Subtract Numeric
- Signal Exception

0C0A Size

An operand was too small to contain a result. This condition is detected only when a fixed-point result is too large to be assigned to a fixed-point receiver. The receiver operand is set with the result of the operation truncated to the receiver size.

Instructions Causing Exception:

- Add Numeric
- Compute Math Function Using One Input Value
- Compute Math Function Using Two Input Values
- Convert Character to Numeric
- Convert External Form
- Convert Numeric to Character
- Copy Numeric Value
- Divide
- Divide With Remainder
- Extract Magnitude
- Multiply
- Negate
- Remainder
- Scale
- Subtract Numeric
- Sum
- Signal Exception
- Trim Length

0C0B Zero Divide

An attempt was made to divide by 0 on a fixed-point divide operation.

Instructions Causing Exception:

- Divide
- Divide With Remainder
- Remainder
- Signal Exception

0C0C Invalid Floating-Point Conversion

This exception is detected on a conversion from binary floating-point to other than a binary floating-point format because overflow, infinity, or NaN is detected before conversion is complete.

Information Passed:

• Floating-point value attributes	Char(1)
— Normal bias	Bit 0
— Modified bias	Bit 1
— Reserved (binary 0)	Bit 2
— NaN	Bit 3
— Infinity	Bit 4
— Reserved (binary 0)	Bits 5-7
• Reserved (binary 0)	Char(7)
• Invalid floating-point value	Floating-point (8)
• Reserved (binary 0)	Char(16)

Instructions Causing Exception:

- Add Numeric
- Compute Math Function Using One Input Value
- Compute Math Function Using Two Input Values
- Convert Character to Numeric
- Convert Floating-Point to Decimal Form
- Convert Numeric to Character
- Copy Numeric Value
- Divide
- Multiply
- Negate
- Scale
- Subtract Numeric
- Signal Exception

0C0D Floating-Point Inexact Result

This exception is signaled when the rounded result of an operation is not exact because of one of the following conditions:

- The rounded result of an operation is not exact because a value is modified (rounded) to fit in a receiver operand and nonzero fraction digits are lost.
- The occurrence of a floating-point overflow condition when that condition is masked and the result is no longer exact because it is set to infinity.

Information Passed:

- Reserved (binary 0) Char(32)

Instructions Causing Exception:

- Add Numeric
- Compare Numeric Value
- Compute Math Function Using One Input Value
- Compute Math Function Using Two Input Values
- Convert Character to Numeric
- Convert Decimal Form to Floating-Point
- Convert Floating-Point to Decimal Form
- Copy Numeric Value
- Divide
- Extract Magnitude
- Multiply
- Negate
- Scale
- Subtract Numeric
- Signal Exception

0C0E Floating-Point Zero Divide

This exception is signaled for a floating-point division operation if the divisor is zero and the dividend is a finite nonzero number.

Instructions Causing Exception:

- Compute Math Function Using Two Input Values
- Divide
- Signal Exception

0C0F Master Key Not Defined

The cipher operation requested use of the master key but the master key has not been defined by the Modify Machine Attributes instruction or is invalid.

Instructions Causing Exception:

- Cipher
- Cipher Key
- Signal Exception

0C10 Weak Key Not Valid

The key supplied in the template is a weak key and cannot be accepted by the cipher operation.

Instructions Causing Exception:

- Cipher

- Cipher Key
- Modify Machine Attributes
- Signal Exception

0C11 Key Parity Invalid

The key supplied in the template does not have odd parity in each byte and is, therefore, unacceptable for the cipher operation.

Information Passed:

• Offset (byte) to the key field	Bin(2)
• Reserved	Char(6)

Instructions Causing Exception:

- Cipher
- Modify Machine Attributes
- Signal Exception

0C12 Invalid Extended Character Data

The character codes in an extended character data field contain an invalid value.

Instructions Causing Exception:

- Copy Extended Characters Left Adjusted With Pad
- Signal Exception

0C13 Invalid Extended Character Operation

The operand data types specified for an extended character operation are incompatible.

Instructions Causing Exception:

- Copy Extended Characters Left Adjusted With Pad
- Signal Exception

0C14 Invalid Compressed Data

The data to be decompressed contains an invalid *signature*.

Information Passed:

• Actual Result Data Length	Bin(4)

Instructions Causing Exception:

- Decompress Data
- Signal Exception

0C15 Date boundary overflow

The end result of a date calculation or conversion, for a date exceeds the end of the time line of the DDAT

Instructions Causing Exception:

0C15 Data format error

The format of the data does not conform to the format that is described in the DDAT.

Instructions Causing Exception:

0C15 Data value error

The data value of a date, time, or timestamp is not valid with respect to the calendar specified in the DDAT.

Instructions Causing Exception:

0C15 Date boundary underflow

The end result of a date calculation or conversion, the date, precedes the beginning of the time line described in the DDAT.

Instructions Causing Exception:

0C19 Space pointer operands do not point to the same space object

The program that was running attempted to subtract two space pointers from each other, but the pointers did not point to the same space.

Change the program so that the space pointers point to the same space.

Information Passed:

• First space pointer	Space pointer
• Second space pointer	Space pointer

Instructions Causing Exception:

- Signal Exception

0E Context Operation

0E01 Duplicate Object Identification

An attempt was made to place addressability in a context to an object having the same name, type, and subtype as an existing entry in the context.

Information Passed:

- System pointer to the existing object
- Object identification — Char(32)
  - Object type — Char(1)
  - Object subtype — Char(1)
  - Object name — Char(30)

Instructions Causing Exception:

- All create instructions
- Modify Addressability
- Rename Object
- Signal Exception

0E02 Object Ineligible For Context

An attempt was made to delete addressability to an object of a type that may be addressed only by the machine context, or an attempt was made to place addressability to an object in a temporary or permanent context that may be addressed only by the machine context.

Information Passed:

- System pointer to object
- Object identification — Char(32)
  - Object type — Char(1)
  - Object subtype code — Char(1)
  - Object name — Char(30)

Instructions Causing Exception:

- Modify Addressability
- Signal Exception

10 Damage

1002 Machine Context Damage State

The machine context cannot be referenced because it is in the damaged state. The machine context rebuild option of the Reclaim instruction can be used to correct the problem or an IPL can correct the problem.

Information Passed:

- Reserved (binary 0) — Char(16)
- VLOG dump ID — Char(8)
- Error class — Bin(2)
- The error class codes for the type of damage detected are as follows:

  Hex 0000 = Previously marked damaged
  Hex 0001 = Detected abnormal condition
  Hex 0002 = Locally invalid device sector
  Hex 0003 = Device failure

- Auxiliary storage device failure — Bin(2)

  This field is defined for error classes hex 0002 and hex 0003. It is the unit number of the failing device or 0 for a main storage failure.

- Reserved (binary 0) — Char(100)

Instructions Causing Exception:

- Materialize Context
- Resolve System Pointer
- Any instruction that resolves a system object that is located by the machine context
- Signal Exception

1004 System Object Damage State

A system object cannot be accessed because it is in the damaged state.

Information Passed:

- System pointer to the damaged object — System pointer
- VLOG dump ID — Char(8)
- Error class — Bin(2)
- The error class codes for the type of damage detected are as follows:

  Hex 0000 = Previously marked damaged
  Hex 0001 = Detected abnormal condition
  Hex 0002 = Locally invalid device sector
  Hex 0003 = Device failure

- Auxiliary storage device indicator — Bin(2)

  This field is defined for error classes hex 0002 and hex 0003. It is the unit number of the failing device or 0 for a main storage failure.

- Reserved (binary 0) — Char(100)

Instructions Causing Exception:

- Any instruction that references a system object
- Signal Exception

1005 Authority verification terminated due to damaged object

Authority verification processing terminated due to a damaged user profile or authorization list found during the check of authority for a permanent system object.

Information Passed

• System Pointer to the object for which authority was being checked	System pointer
• Reason Code	Bin(2)

0 = Damaged User Profile
1 = Damaged Authorization List

(all other values reserved)

• Reserved	Char(14)
• System Pointer to the damaged User Profile or Authorization List	System pointer

Instructions Causing Exception

- Any instruction with operands or operand lists that refer to a permanent system object
- Signal Exception

1044 Partial System Object Damage

Partial damage to a system object has been detected.

Information Passed:

• System pointer to the damaged object	System pointer
• VLOG dump ID	Char(8)
• Error Class	Bin(2)

- The error class codes for the type of damage detected are as follows:

  Hex 0000 = Previously marked damaged
  Hex 0001 = Detected abnormal condition
  Hex 0002 = Locally invalid device sector
  Hex 0003 = Device failure

• Auxiliary storage device indicator	Bin(2)

This field is defined for error classes hex 0002 and hex 0003. It is the unit number of the failing device or 0 for a main storage failure.

• Reserved (binary 0)	Char(100)

Instructions Causing Exception:

- Any instruction that references a system object
- Signal Exception

Meaning and recovery:

- For a user profile, partial damage occurs when internal processing is unable to complete due to insufficient machine storage. When the system is IPLed, if there is then sufficient storage, the internal processing is completed and the partial damage condition is reset.

12 Data Base Management

1201 Conversion Mapping Error

During conversions of a field from one data representation to another, an error occurred. The specific error is indicated in the error type value, below.

Information Passed:

The following data is provided:

- Cursor — System pointer
- Data space number — Bin(2)
- Ordinal entry number (0 if signaled during an Insert Data Space Entry or an Insert Sequential Data Space Entries instruction) — Bin(4)
- Number of fields in error — Bin(2)
- Field data (repeated for each field that is in error)
  - Field number — Bin(2)
  - Error type — Char(2)
- Tolerate all mapping errors exception data
  - Status bits — Char(2)
    - Bitmap is specified — Bit 0

      0 = Bit map is not returned in the exception data
      1 = Bit map is returned in the exception data
    - Join field incurred an error — Bit 1

      0 = A join field did not incur an error
      1 = A join field did incur an error
    - No records in group — Bit 1

      0 = Group-by processing not in effect or a valid group was returned
      1 = Group-by processing found no records which qualify as part of the defined group (set only when grouping WITHOUT a data space index)
    - Reserved (binary zero) — Bit 3-15
  - Bit map byte length — Bin(2)
  - Bit map identifying the erroneous user buffer fields — Char(*)
  - Number of first exceptions — Bin(2)
  - First exception data (repeated for each offending field) — Char(14)
    - First error cursor data space number — Bin(2)
    - First error ordinal entry number — Bin(4)
    - First error join cursor ordinal positions — Char(128)
    - First error field number — Bin(2)
    - First error error type — Char(2)

- First error field location Char(1)
- Reserved Char(5)

The ordinal entry number will contain a value of binary zero when the exception occurs when inserting new data space entries or performing group-by and join operations.

The field number is the relative location of the field as specified when creating the cursor. A field number of 1 is the first field in the data field location.

The error type values are as follows:

- Hex 0001-Conversion: An invalid character was found by the derived operation, Invalid Character Scan.
- Hex 0002-Decimal Data: (1) Sign encoding is invalid for packed or zoned format, or (2) digit encoding is invalid for packed or zoned format.
- Hex 0006-Floating-Point Overflow: During conversion, a floating-point value exceeded the maximum value that can be represented.
- Hex 0007-Floating-Point Underflow: During conversion, a floating-point value became less than the minimum value that can be represented.
- Hex 0008-Length Conformance: A length specified for a variable length field was outside the limits of that field.
- Hex 0009-Floating-Point Invalid Operand: A floating-point NaN was used as an operand to convert from long floating-point format to short floating-point format or from short floating-point format to long floating-point format.
- Hex 000A-Size: The destination field is too small to hold all significant digits of the source field.
- Hex 000C-Invalid Floating-Point Conversion: An invalid floating-point value was used as an operand to convert floating-point to packed decimal or floating-point to binary.
- Hex 000D-Floating-Point Inexact Result: In a conversion operation, a floating-point value had at least one bit of precision rounded away.
- Hex 0012- Invalid extended character data; The character data in an extended character data field contain an invalid value.
- Hex 0015-Date boundary overflow: The end result of a date calculation or conversion, a date, exceeds the end of the time line described in the DDAT. Reference the Common Template Appendix for information on the DDAT.
- Hex 0016-Data format: The format of the data does not conform to the format that is described in the DDAT. Reference the Common Template Appendix for information on the DDAT.
- Hex 0017-Data value: The data value of a date, time, or timestamp is invalid with respect to the calendar specified in the DDAT. Reference the Common Template Appendix for information on the DDAT.
- Hex 0018-Date boundary underflow: The end result of a date calculation or conversion, a date, precedes the beginning of the time line described in the DDAT. Reference the Common Template Appendix for information on the DDAT.
- Hex 0019-Null status: An invalid situation exists relating to null capable fields. On retrieve related instructions, a null field was encountered and a null map was not provided with the instruction. On insert/update instructions, a null map was provided and indicated a field which is not null capable should be set to null or the null map contains an invalid null or not null value.

These errors are equivalent to the computational exceptions numbered 0C01, 0C02, 0C06, 0C07, 0C08, 0C09, 0C0A, 0C0C, 0C0D, 0C12, 0C14, 0C15, 0C16, and 0C17, and occur for similar reasons.

The bit map is specified bit indicates if the bit map identifying the erroneous user buffer fields is present in the exception data

The join field incurred an error bit indicates if any join fields incurred mapping errors. This bit is always binary zero for non-join cursors.

The bit map identifying the erroneous user buffer fields maps each field of the user buffer to a bit map in a one-to-one correspondence. Field 1 is bit 1, field 2 is bit 2, etc. If the bit corresponding to a user buffer field is 1 then that field is erroneous and has been replaced with the default value (blank for character, zero for numeric). The length of the bit map is given by:

(((# of fields in user buffer) + 7)/8 + 1)&'FFFE'X.

The first exception data area identifies the first field that incurred an error when a cursor created with the tolerate all mapping errors attribute is being used for retrieval operations. The field identified is not necessarily represented in the returned record nor is it necessarily represented in the bitmap of erroneous fields since it may not have been mapped to the user buffer. Instead, this is the very first field which incurred an error in the series of mappings which ultimately produced the record-view in the user's buffer.

The first error cursor data space number will contain binary zero when the error occurs during group-by derived field operations.

The first error ordinal entry number will contain binary zero when the error occurs during group-by derived field operations or while processing default values during a join operations.

The first error join cursor ordinal positions area is meaningful only on operations involving a join cursor and is zero in instances of non-join. It will also be zero if the exception occurs during group-by processing or derived field operations. Each ordinal is 4 bytes in length. All current ordinals associated with a join cursor are returned.

The first error field number is as described for field number above.

The first error error-type is as described for error type above with the addition of the following types:

Hex 000B -An attempt was made to divide by zero in a fixed point divide operation.

Hex 000E -An attempt has been made to divide by zero in a floating point divide operation.

The first error field location identifies the buffer where the first error occurred. The following values can be returned:

00 = Data space entry

01 = Cursor intermediate buffer

02 = Group-by intermediate buffer

Instructions Causing Exception:

- Copy Data Space Entries
- Insert Data Space Entry
- Insert Sequential Data Space Entries
- Retrieve Data Space Entry
- Retrieve Sequential Data Space Entries
- Update Data Space Entry
- Signal Exception

1202 Key Mapping Error

During conversions of a field from one data representation to another, an error occurred. The specific error is indicated in the error type value, below.

Information Passed:

The following data is provided:

Cursor	System pointer
Data space number	Bin(2)
Ordinal Entry Number	Char(4)
Number of fields in error	Bin(2)
Field data (repeated for each field that is in error)	
• Field number	Bin(2)
• Error type	Char(2)
Tolerate all mapping errors exception data	
• Status bits	Char(2)
— Bitmap is specified	Bit 0
0 = Bit map is not returned in the exception data 1 = Bit map is returned in the exception data	
— Reserved (binary zero)	Bit 1
— Reserved (binary zero)	Bit 2-15
• Bit map byte length	Bin(2)
• Bit map identifying the erroneous key buffer fields	Char(*)
• Number of first exceptions	Bin(2)
• First exception data (repeated for each offending field)	Char(144)
— First error cursor data space number	Bin(2)
— First error ordinal entry number	Bin(4)
— Reserved	Char(128)
— First error field number	Bin(2)
— First error error type	Char(2)
— First error field location	Char(1)
— Reserved	Char(5)

The field number is the relative location of the field as specified when creating the cursor. A field number of 1 is the first field in the data field location.

The error type values are as follows:

- Hex 0002-Decimal Data: (1) Sign encoding is invalid for packed or zoned format, or (2) digit encoding is invalid for packed or zoned format.
- Hex 0006-Floating-Point Overflow: During conversion, a floating-point value exceeded the maximum value that can be represented.

- Hex 0007-Floating-Point Underflow: During conversion, a floating-point value became less than the minimum value that can be represented.
- Hex 0008-Length Conformance: A length specified for a variable length field was outside the limits of that field.
- Hex 0009-Floating-Point Invalid Operand: A floating-point NaN was used as an operand to convert from long floating-point format to short floating-point format or from short floating-point format to long floating-point format.
- Hex 000A-Size: The destination field is too small to hold all significant digits of the source field.
- Hex 000C-Invalid Floating-Point Conversion: An invalid floating-point value was used as an operand to convert floating-point to packed decimal or floating-point to binary.
- Hex 000D-Floating-Point Inexact Result: In a conversion operation, a floating-point value had at least one bit of precision rounded away.
- Hex 0014-Date boundary underflow: The end result of a date calculation or conversion, a date, precedes the beginning of the time line described in the DDAT. Reference the Common Template Appendix for information on the DDAT.
- Hex 0015-Date boundary overflow: The end result of a date calculation or conversion, a date, exceeds the end of the time line described in the DDAT. Reference the Common Template Appendix for information on the DDAT.
- Hex 0016-Data format: The format of the data does not conform to the format that is described in the DDAT. Reference the Common Template Appendix for information on the DDAT.
- Hex 0017-Data value: The data value of a date, time, or timestamp is invalid with respect to the calendar specified in the DDAT. Reference the Common Template Appendix for information on the DDAT.
- Hex 0018-Null status: An invalid situation exists relating to null capable fields. On instructions that materialize a key, a null field was encountered and a null map was not provided with the instruction. On instructions with a key as input, a null map was provided and indicated a field which is not null capable should be set to null or the null map contains an invalid null or not null value.
- Hex 00F0-A substring set cursor operation was requested. The byte count specified ended in a key field that cannot be severed. This key field type could be but is not limited to: binary, floating point, date, or timestamp.

These errors are equivalent to the computational exceptions numbered 0C02, 0C06, 0C07, 0C08, 0C09, 0C0A, 0C0C, 0C0D, 0C14, 0C15, 0C16, and 0C17, and occur for similar reasons.

The ordinal entry number field will be zero if the exception occurred on input key mapping. For output key mapping, ordinal entry number will contain the ordinal number of the DSE which caused the exception.

The bit map is specified bit indicates if the bit map identifying the erroneous user buffer fields is present in the exception data

The bit map identifying the erroneous key buffer fields maps each field of the key buffer to a bit map in a one-to-one correspondence. Field 1 is bit 1, field 2 is bit 2, etc. If the bit corresponding to a key buffer field is 0 then that field incurred no errors. If the bit corresponding to a key buffer field is 1 then that field is erroneous and has been replaced with the default value (blank for character, zero for numeric). The length of the bit map is given by:

(((# of fields in user buffer) + 7)/8 + 1)&'FFFE'X.

The first exception data area identifies the first field that incurred an error when a cursor created with the tolerate all mapping errors attribute is being used for retrieval operations. The

field identified is not necessarily represented in the returned key nor is it necessarily represented in the bitmap of erroneous fields since it may not have been mapped to the key buffer. Instead, this is the very first field which incurred an error in the series of mappings which ultimately produced the key-view in the key buffer.

The first error cursor data space number will contain binary zero when the error occurs during group-by derived field operations.

The first error ordinal entry number will contain binary zero when the error occurs during group-by derived field operations.

The first error field number is as described for field number above.

The first error error-type is as described for error type above with the addition of the following types:

000B An attempt was made to divide by zero in a fixed point divide operation.
000E An attempt has been made to divide by zero in a floating point divide operation.

The first error field location identifies the buffer where the first error occurred. The following values can be returned:

00 = Data space entry
03 Intermediate key buffer

Instructions Causing Exception:

- Copy Data Space Entries (mapping from template)
- Materialize Cursor Attributes (mapping key out to buffer)
- Retrieve Sequential Data Space Entries (mapping key to buffer)
- Set Cursor (mapping key in/out)
- Signal Exception

1203 Cursor Not Set

An attempt was made to perform a data base operation using a cursor that is not set to address a data space entry.

Information Passed:

- System pointer to cursor
- Data space number Bin(2)

The data space number will be zero for a non-join cursor.

Instructions Causing Exception:

- Retrieve Data Space Entry
- Set Cursor
- Signal Exception

1204 Data Space Entry Limit Exceeded

The operation caused the user-provided maximum number of entries limitation for the data space to be exceeded.

Information Passed:

- Cursor (binary 0 for instruction not involving a cursor) System pointer
- Data space System pointer

Instructions Causing Exception:

- Copy Data Space Entries
- Data Base Maintenance (insert default entries and insert deleted entries option)
- Insert Data Space Entry
- Insert Sequential Data Space Entries
- Signal Exception

1205 Data Space Entry Already Locked

An attempt has been made to lock a data space entry using the Set Cursor instruction when the data space entry is already locked to a cursor (this cursor or another cursor) or to a commit block that is not attached to this process. A system pointer to the process control space of the process that activated the cursor or attached the commit block that holds the lock is returned.

Information Passed:

• Cursor	System pointer
• Data space	Bin(2)
• Ordinal entry number	Bin(4)
• Return code (bit significant) Hex 00 = Locked to another process Hex 01 = Locked to current process	Char(1)
• Reserved (binary 0)	Char(9)
• Process control space	System pointer

Instructions Causing Exception:

- Set Cursor
- Signal Exception

1206 Data Space Entry Not Found

An attempt has been made to refer to a data space entry that could not be found because the entry has been deleted or its key has been omitted from the data space index.

Information Passed:

• Cursor	System pointer
• Data space number	Bin(2)

Instructions Causing Exception:

- Retrieve Data Space Entry
- Set Cursor
- Signal Exception

1207 Data Space Index Invalid

The index specified for a data base operation is not usable.

Information Passed:

• Cursor (binary 0 for instructions not involving cursor)	System pointer
• Data space index	System pointer

Instructions Causing Exception:

- Activate Cursor

- Copy Data Space Entries
- Retrieve Data Space Entry
- Retrieve Sequential Data Space Entries
- Set Cursor
- Signal Exception

1208 Incomplete Key Description

The cursor cannot be set by key for this data space index because the output mapping template used to create this cursor failed to provide a description of each field that comprises the key, or an alternate entry definition supplied at Create Cursor differs from the entry definition used at Create Data Space Index.

Information Passed:

- Cursor — System pointer
- Data space number of the key format selected — Bin(2)

Instructions Causing Exception:

- Copy Data Space Entries
- Create Cursor
- Set Cursor
- Signal Exception

1209 Duplicate Key Value in Existing Data Space Entry

An attempt has been made to insert or update a data space entry in a data space over which a unique keyed index has been built, and the data space entry has a key value identical to an existing data space entry addressed by the index.

Information Passed:

- Cursor (binary 0 for operations not involving a cursor) — System pointer
- Data space index — System pointer
- The data space number of the entry associated with the key already in the data space index — Bin(2)
- The ordinal number of the entry associated with the key already in the data space index — Bin(4)
- The data space number of the entry that was being added or changed and caused the exception — Bin(2)
- The ordinal number of the entry that was being changed and caused the exception(0 if an insert was being attempted) — Bin(4)

Instructions Causing Exception:

- Apply Journaled Changes
- Copy Data Space Entries
- Insert Data Space Entry
- Insert Sequential Data Space Entries
- Update Data Space Entry
- Signal Exception

120A End of Path

The end of an access path was reached when an attempt was made to position a cursor.

Information Passed:

- Cursor — System pointer

Instructions Causing Exception:

- Retrieve Sequential Data Space Entries
- Set Cursor
- Signal Exception

120B Duplicate Key Value Detected

While creating or rebuilding a data space index with the unique key attribute, entries were found to generate the same key value. The build detected up to a maximum of 20 duplicate key values before terminating.

Information Passed:

- Data space index — System pointer
- Number of duplicates detected — Bin(2)
- (Repeated for each duplicate)
  - Data space number of first entry — Bin(2)
  - Ordinal number of first entry — Bin(4)
  - Data space number of second entry — Bin(2)
  - Ordinal number of second entry — Bin(4)

Instructions Causing Exception:

- Create Data Space Index
- Data Base Maintenance (rebuild option)
- Signal Exception

120D No Entries Locked

No data space entries were locked to this cursor.

Information Passed:

- Cursor — System pointer

Instructions Causing Exception:

- Delete Data Space Entry
- Update Data Space Entry
- Signal Exception

120F Duplicate Key Value in Uncommitted Data Space Entry

An attempt has been made to insert or update a data space entry in a data space over which a unique keyed index has been created, and the data space entry has a key value identical to a data space entry key value that has been deleted or changed under commitment control but is still reserved by the index. The insert or update cannot be done until the uncommitted changes are committed.

Information Passed:

- Cursor (binary 0 for — System pointer

operations not involving a cursor)

- Data space index — System pointer
- The data space number of the entry associated with the key reserved in the data space index — Bin(2)
- The ordinal number of the entry associated with the key reserved in the data space index — Bin(4)
- The data space number of the entry that was being added or changed and caused the exception — Bin(2)
- The ordinal number of the entry that was being changed and caused the exception (zero if an insert was being attempted) — Bin(4)

Instructions Causing Exception:

- Copy Data Space Entries
- Insert Data Space Entry
- Insert Sequential Data Space Entries
- Update Data Space Entry
- Signal Exception

1213 Invalid Mapping Template

An error was detected in a mapping template. The data space number indicates the template in the mapping template list that contains the error. This field will equal 1 for a group-by mapping template, and will reference the appropriate template for the per data space mapping templates for Create Cursor and Create Data Space Index instructions. This field will contain a zero when an intermediate mapping table is missing.

The template field number indicates which field number is in error for the scalar part of the mapping template (number of bytes in mapping template and mapping type). The contents of the reserved field immediately following the mapping type field is ignored (only present when intermediate mapping is specified).

The template field number also indicates the field in error for input and output mapping tables. Field number equal 0 indicates the template field number of bytes in the mapping template, field number equal 1 indicates field designating input mapping type, field number equal 2 indicates field designating output mapping type, field number equal 3 indicates field designating number of fields in the input/output mapping tables, and so forth. Each specification in the input/output tables is considered one field for counting purposes.

When an intermediate mapping table is in error, the template field number will equal 0. The offset to field in error field will designate the offset to the field from the start of the intermediate mapping table. Offset equal 0 designates the field number of intermediate mapping specifications, and so forth.

Possible errors are an invalid value, a value that exceeds allowed range, a length that is invalid for the specified type or a type that is inconsistent with the type specified for the field in the data space, key description, or an intermediate buffer description.

The invalid mapping template exception will not be signaled when a data pointer fails verification. The normal exception associated with verifying data pointers will be signaled instead.

Information Passed:

- Data space number — Bin(2)
- Template field number (valid only for input/output mapping tables and the scalar part of the mapping template; number of bytes in template, input mapping type, output mapping type) — Bin(2)

- Template type — Char(1)

  Hex 00 = Per data space mapping template (input/output mapping table and scalar part of template)
  Hex 01 = Per data space mapping template (output intermediate mapping table)
  Hex 02 = Group-by intermediate mapping table
  Hex 03 = Derived key intermediate mapping table

- Reserved — Char(1)
- Offset to field in error (valid only for intermediate mapping table errors) — Bin(4)
- Error type (valid only for intermediate mapping table errors) — Char(2)
  - GENERAL ERROR CODES

    Hex 0000 = Data Pointer error
    Hex 0001 = Reserved field invalid
    Hex 0300 = Missing Intermediate mapping table
    Hex 1000 = Operation field error
    Hex 1012 = Specified Operation invalid for Variable length fields
    Hex 1014 = Invalid Null Operation modifier

  - OPERAND 1 ERROR CODES

    Hex 8010 = Operand 1 Location invalid
    Hex 8020 = Operand 1 Length invalid
    Hex 8040 = Operand 1 Field type invalid
    Hex 8080 = Operand 1 Field number invalid
    Hex 8008 = Operand 1 Start character invalid
    Hex 8004 = Operand 1 End character invalid
    Hex 8001 = Operand 1 Reserved Area invalid
    Hex 8400 = Operand 1's Array Position of the Data Pointer is invalid
    Hex 8088 = Operand 1's DDAT is invalid
    Hex 8082 = Operand 1's Preferred format or separator is invalid
    Hex 8048 = Operand 1's Start or End character referenced an invalid field type for a Substring operation
    Hex 8044 = Operand 1 is a Split Variable length field that is not compatible with the specified operation

  - OPERAND 2 ERROR CODES

    Hex 4010 = Operand 2 Location invalid
    Hex 4020 = Operand 2 Length invalid
    Hex 4040 = Operand 2 Field type invalid
    Hex 4080 = Operand 2 Field number invalid
    Hex 4008 = Operand 2 Start character invalid
    Hex 4004 = Operand 2 End character invalid
    Hex 4001 = Operand 2 Reserved Area invalid
    Hex 4100 = Operand 2's Array Position of the Translate Table is invalid
    Hex 4400 = Operand 2's Array Position of the Data Pointer is invalid
    Hex 4088 = Operand 2's DDAT is invalid
    Hex 4044 = Operand 2 is a Split Variable length field that is not compatible with the specified operation
    Hex 4022 = Operand 2's Immediate Value or Length is invalid

  - RESULT ERROR CODES

    Hex 2020 = Result Length invalid
    Hex 2040 = Result Type invalid

Hex 2001 = Result Reserved area invalid
Hex 2002 = Result Rounding mode invalid
Hex 2004 = Result Type modifier is invalid
Hex 2088 = Result's DDAT is invalid

- DATA POINTER ERRORS

  Hex 0020 = Data Pointer length is invalid
  Hex 0040 = Data Pointer field type is invalid

- Error Types for Input and Output mapping templates

  Hex 0001 = Field Location is invalid
  Hex 0002 = Field Type is invalid
  Hex 0003 = Field Length is invalid
  Hex 0004 = Reserved area is not zero
  Hex 0005 = DDAT number references an invalid DDAT number
  Hex 0006 = DDAT number references an invalid format code
  Hex 0007 = DDAT number references an invalid preferred format or separator
  Hex 0008 = Type modifier is invalid
  Hex 0009 = Variable Length/Type mismatch
  Hex 000A = DDAT offset mismatch between mapping template DDAT number and field table DDAT offset
  Hex 000B = Mapping Conversion is invalid
  Hex 000C = Mapping a pseudo Date/Time field that is variable length and the fixed allocation length is greater than zero and less than the maximum size of the field.

Instructions Causing Exception:

- Create Cursor
- Signal Exception

1214 Invalid Selection Template

An error was detected in a selection template. The data space number indicates which template in the selection template list contains the error. This field will equal 1 for a group-by selection template, and will reference the appropriate template for the per data space mapping template for Create Cursor and Create Data Space Index.

The offset to field in error indicates which field is in error in the selection template. The offset equal 0 designates the field length of selection template is in error, offset equal 4 designates the field number of selection descriptors is in error, and so forth.

The invalid selection template exception will not be signaled when a data pointer fails verification. The normal exception associated with verifying data pointers will be signaled instead.

Information Passed:

- Data space number (position list — Bin(2)
- Offset to field in error — Bin(4)
- Selection template type — Char(1)

  Hex 00 = Per data space selection
  Hex 01 = Group-by selection

- Reserved — Char(1)
- Selection descriptor errors — Char(2)
  - Descriptor type error — Bit 0
  - Operand/operation error — Bit 1
  - Maximum number specifications — Bit 2

exceeded

– Literal content in error	Bit 3
– Reserved	Bits 4-7
– Operand location invalid	Bit 8
– Field number invalid	Bit 9
– Starting offset invalid	Bit 10
– Ending offset invalid	Bit 11
– Array position of data pointer invalid	Bit 12
– Offset to pattern descriptor invalid	Bit 13
– Number pattern descriptor invalid	Bit 14
– Reserved field invalid	Bit 15
• Pattern descriptor error types	Char(2)
– Descriptor type invalid	Bit 0
– Descriptor field invalid	Bit 1
– Reserved	Bits 2-7
– Field location invalid	Bit 8
– Field number invalid	Bit 9
– Starting offset invalid	Bit 10
– Ending offset invalid	Bit 11
– Array position of data pointer invalid	Bit 12
– Span type invalid	Bit 13
– Span width invalid	Bit 14
– Reserved field invalid	Bit 15

Instructions Causing Exception:

- Create Cursor
- Signal Exception

1215 Data Space Not Addressed by Index

An entry in the data space list does not address the same data space that is addressed by the corresponding entry in the data space list defined for the data space index.

Information Passed:

• Entry in the data space list of the Create Cursor instruction template	Space pointer

Instructions Causing Exception:

- Create Cursor
- Signal Exception

1216 Data Space Not Addressed by Cursor

An entry in the data space list does not address the same data space that is addressed by the corresponding list that is defined for the cursor.

Information Passed:

- Cursor — System pointer
- Entry in the data space list of the Activate Cursor instruction template — Space pointer

Instructions Causing Exception:

- Activate Cursor
- Signal Exception

1217 Key Value Changed Since Set Cursor

The data space index key for the entry currently addressed by the cursor has changed since the cursor was set. The former value of the key was instrumental in finding the entry and is no longer valid; therefore, the entry is no longer the expected entry.

Information Passed:

- Cursor — System pointer
- Data space number — Bin(2)

Instructions Causing Exception:

- Retrieve Data Space Entry
- Set Cursor
- Signal Exception

1218 Invalid Key Value Modification

An attempt to update a data space entry would result in a difference key value when the Update Data Space Entry instruction specified the inhibit key change option. This exception is signaled for rejected key values if the rejected key image would change on the update, regardless of whether the updated key image would be rejected or selected.

Note: Updating a deleted data space entry is not a change in the key image.

Information Passed:

- Cursor — System pointer
- Data space number — Bin(2)
- Ordinal number of entry — Bin(2)

Instructions Causing Exception:

- Update Data Space Entry
- Signal Exception

1219 Invalid Rule Option

The cursor has addressability to a data space index and the current cursor setting allows only rule options of relative or ordinal.

Information Passed:

- Cursor — System pointer

Instructions Causing Exception:

- Retrieve Sequential Data Space Entries
- Set Cursor
- Signal Exception

121A Data Space Entry Size Exceeded

The sum of the field lengths in the entry definition template exceeds 32 766 bytes which is the maximum size allowed for a data space entry.

Instructions Causing Exception:

- Create Data Space
- Signal Exception

121B Logical Data Space Entry Size Limit Exceeded

The user's view of the data space entry (defined by the mapping code) exceeds 32 766 bytes, which is the maximum size allowed.

Information Passed:

- Template number (position list) — Bin(2)
- Template type — Char(1)

  Hex 00 = Input mapping template
  Hex 01 = Output mapping template
  Hex 02 = Intermediate mapping template
  Hex 03 = Group-by output mapping template
  Hex 04 = Group-by intermediate mapping template

Instructions Causing Exception:

- Create Cursor
- Signal Exception

121C Key Size Limit Exceeded

The sum of the key field lengths plus the specified fork characters exceeds 120 bytes, which is the maximum size allowed for a data space index key.

Information Passed:

- Data space number — Bin(2)

Instructions Causing Exception:

- Create Data Space Index
- Signal Exception

121D Logical Key Size Limit Exceeded

The user's view of the data space index key exceeds 32 766 bytes, which is the maximum size allowed.

Information Passed:

- Data space number — Bin(2)

Instructions Causing Exception:

- Create Cursor
- Signal Exception

1221 Unable to Maintain a Unique Key Data Space Index

An attempt has been made to insert or update a data space entry in a data space over which a unique keyed index exists that has been implicitly invalidated.

Information Passed:

- Cursor (binary 0 for operations not involving a cursor) — System pointer

- Data space — System pointer
- Data space index (invalidated) — System pointer

Instructions Causing Exception:

- Apply Journaled Changes
- Copy Data Space Entries
- Insert Data Space Entry
- Insert Sequential Data Space Entries
- Update Data Space Entry
- Signal Exception

1225 Invalid Data Base Operation

A data base operation was attempted through a cursor whose activation options indicated that the operation was not to be allowed.

Information Passed:

- Cursor — System pointer
- Extended activation functions (as defined in the cursor activation template) — Char(2)
- Operation attempted — Char(1)

  Hex 80 = Retrieval of data space entry
  Hex 40 = Update of data space entry
  Hex 20 = Delete of data space entry
  Hex 10 = Insert of data space entry

Instructions Causing Exception:

- Copy Data Space Entries
- Delete Data Space Entry
- Insert Data Space Entry
- Insert Sequential Data Space Entries
- Retrieve Data Space Entry
- Retrieve Sequential Data Space Entries
- Update Data Space Entry
- Signal Exception

1226 Data Space Index with Invalid Floating-Point Field Build Termination

While creating or rebuilding a data space index that contains floating-point keys, an invalid floating-point value was encountered. Up to 20 instances of these types of errors may be found before the instruction is terminated.

Information Passed:

- Data space index (binary 0 is signaled during creation) — System pointer
- Number of errors detected — Bin(2)
- Error description (repeated for each selection routine error)
  - Data space number — Bin(2)
  - Ordinal entry number — Bin(4)

  - Reason code — Char(1)

    Hex 01 = Floating-point NaN detected

  - Reserved (binary 0) — Char(1)

Instructions Causing Exception:

- Create Data Space Index
- Data Base Maintenance
- Signal Exception

1227 Data Space Index Key with Invalid Floating-Point Field

An attempt was made to insert or update a data space entry in a data space under a data space index that contains floating-point key fields, or a key is being used to search a data space index that contains floating-point key fields and a floating-point key field contains an invalid value.

Information Passed:

- Cursor (binary 0 for operations not involving a cursor) — System pointer
- Data space — System pointer
- Data space index — System pointer
- Data space number (in the data space list of the data space index) — Bin(2)
- Ordinal entry number (zero if entry was being inserted) — Bin(4)
- Reason code — Char(1)

  Hex 01 = Floating-point NaN detected
- Reserved — Char(1)
- Cursor data space number — Bin(2)

Instructions Causing Exception:

- Apply Journaled Changes
- Data Base Maintenance
- Copy Data Space Entries
- Insert Data Space Entry
- Insert Sequential Data Space Entries
- Set Cursor
- Update Data Space Entry
- Signal Exception

1230 Specified Data Space Entry Rejected

An attempt has been made to position a cursor to a specific data entry but the retrieval selection criteria has rejected the entry.

Information Passed:

- Cursor — System pointer
- Data space number — Bin(2)
- Ordinal number — Bin(4)

Instructions Causing Exception:

- Retrieve Data Space Entry
- Set Cursor
- Signal Exception

1232 Join Value Changed

A join value in a data space entry field used in the current join position in the cursor has changed since the cursor was positioned. The former value of the field was instrumental in performing the join operation and is no longer valid; therefore, the entry is no longer the expected entry.

Information Passed:

• Cursor	System pointer
• Data space number in cursor associated with changed field	Bin(2)

Instructions Causing Exception:

- Retrieve Data Space Entry
- Set Cursor
- Signal Exception

1233 Data Space Index with Non-User Exit Selection Routine Build Termination

While creating or rebuilding a data space index that contains a non-user exit selection routine, data space entries were encountered which resulted in an error in the selection routine. The build, before terminating, found up to 20 instances of these types of errors. The instruction is terminated.

Information Passed:

• Data space index (binary zeros if signaled during creation)	System pointer
• Number of errors detected	Bin(2)
• Reserved	Char(4)
• Error description (repeated for each selection routine error)	Char(22)
– Data space number	Bin(2)
– Ordinal entry number	Bin(4)
– Reserved (binary 0)	Char(6)
– Error type	Char(2)
– Operand 1 field data	Char(4)
– Field number	Bin(2)
– Field location Hex 00 = Data space entry Hex 01 = Cursor intermediate buffer Hex 03 = Intermediate key buffer Hex 04 = Key field	Char(1)
– Reserve	Char(1)

- Operand 2 field data (same as operand 1 field data) — Char(4)

The field number designates the relative location of the field as specified when creating the cursor. index, or data space. Field number equal 1 is the first field in the field location.

The error type values are as follows:

Hex 0000-A tolerated exception occurred during derived field mapping and was detected during non-user exit selection when the field incurring the error was referenced.

Hex 0002-decimal data: (1) Sign encoded is invalid for packed or zoned format, (2) Digit encoding is invalid for packed or zoned format.

Hex 0009-floating-point invalid operand: A floating-point NaN was used as an operand in a comparison.

Hex 0012-Invalid Extended Character Data; A character code in an extended character data field contains an invalid value.

Instructions Causing Exception:

- Activate Cursor (over delayed maintenance data space index)
- Create Data Space Index
- Data Base Maintenance (rebuild data space index)
- Modify Data Space Index Attributes
- Signal Exception

1234 Non-User Exit Selection Routine Failure

An attempt has been made to insert, retrieve, or update a data space entry in a data space, and an error was encountered in a non-user exit selection routine.

Information Passed:

- Cursor (binary 0 for operations not involving a cursor) — System pointer
- Data space — System pointer
- Data space index (binary 0 if selection error not involving the index) — System pointer
- Join cursor ordinal positions — Char(128)
- Error description — Char(22)
  - Index data space number (binary 0 if selection error not on the index) — Bin(2)
  - Cursor data space number (binary 0 for operations not involving a cursor) — Bin(2)
  - Ordinal entry number (0, if entry was being inserted or group-by selection) — Bin(4)
  - Reserved — Char(4)
  - Error type — Char(2)
  - Operand 1 field data (0 if literal) — Char(4)
    - Field number — Bin(2)
    - Field location — Char(1)
    - Hex 00 = Data space entry

Hex 01 = Cursor intermediate buffer
Hex 02 = Group-by intermediate buffer
Hex 03 = Intermediate key buffer
Hex 04 = Key field

Reserved	Char(1)
— Operand 2 field data (same as operand 1 field data)	Char(4)
— Operand 3 field data (same as operand 1 field data)	Char(4)

The field number designates the relative location of the field as specified when creating the cursor, index, or data space. Field number equal 1 is the first field in the field location. A field number equal 0 designates there is no exception data for this operand.

The error type values are identical with those received for exception hex 1233.

The ordinal entry number will contain a binary zero value if the exception occurs while processing default values during a join operation.

The join cursor ordinal positions area is meaningful only on operations with a join cursor and is zero otherwise. It will be zero for group-by selection. Each ordinal number occupies 4 bytes. All current ordinal numbers associated with a join cursor are returned.

Instructions Causing Exceptions

- Apply Journaled Changes
- Data Base Maintenance (insert default entries)
- Insert Data Space Entry
- Insert Sequential Data Space Entries
- Retrieve Data Space Entry
- Retrieve Sequential Data Space Entry
- Set Cursor
- Update Data Space Entry
- Signal Exception

1236 No Mapping Code Specified

Cursor cannot be used to perform inserts, retrieves, or updates with the specified data space due to no data space entry input mapping code or no output mapping code specified in a Create Cursor instruction.

Information Passed:

• Cursor	System pointer
• Data space number	Bin(2)

Instructions Causing Exception:

- Insert Data Space Entry
- Insert Sequential Data Space Entries
- Retrieve Data Space Entry
- Retrieve Sequential Data Space Entry
- Signal Exception
- Update Data Space Entry

1237 Operation Not Valid with Join Cursor

An attempt has been made to insert, update, or delete a data space entry through a join cursor.

Information Passed:

• Cursor	System pointer

Instructions Causing Exception:

- Copy-Data Space Entries
- Delete Data Space Entry
- Insert Data Space Entry
- Insert Sequential Data Space Entries
- Signal Exception
- Update Data Space Entry

1238 Derived Field Operation Error

During derived field operations, one of a variety of derived field operational errors occurred. The field number designates the relative location of the field as specified when creating the cursor or data space index. Field number equal 1 is the first field in the field location. The field location is described by the field location exception data.

The error type is identical with those for exception number hex 1201, conversion mapping error with the addition of:

- Hex 000B = An attempt has been made to divide by zero on a fixed-point divide operation.
- Hex 000E = An attempt has been made to divide by zero on a floating-point divide operation.

Information Passed:

• Cursor (0 for operations not involving a cursor)	System pointer
• Index (derived key operations)	System pointer
• Join cursor ordinal positions	Char(128)
• Index data space number (binary 0 for operations not involving the index)	Bin(2)
• Cursor data space number (binary 0 for operations not involving the cursor)	Bin(2)
• Ordinal entry number	Bin(4)
• Buffer location type	Char(1)
Hex 01 = Cursor intermediate buffer Hex 02 = Group-by intermediate buffer Hex 03 = Intermediate key buffer	
• Reserved	Char(3)
• Number of offending fields	Bin(2)
• Field data (repeated)	Char(6)
– Field number	Bin(2)
– Error type	Char(2)

- Reserved — Char(2)

The ordinal entry number will contain a binary zero value if the exception occurs during inserts of new entries into a data space or the error occurs while processing default values during a join operation.

The ordinal entry number and data space number will contain a value of binary zero when the error occurs during group-by derived field operations.

Instructions Causing Exception:

- Apply Journal Changes
- Copy Data Space Entries
- Data Base Maintenance
- Insert Data Space Entry
- Insert Sequential Data Space Entries
- Materialize Cursor Attributes
- Modify Data Space Index Attributes
- Retrieve Data Space Entry
- Retrieve Sequential Data Space Entry
- Set Cursor
- Signal Exception
- Update Data Space Entry

1239 Derived Field Operation Error During Build Index

While creating or rebuilding a data space index, data space entries were encountered which resulted in derived field operational errors. The build, before terminating, found up to 20 instances of these types of errors. The instruction is terminated.

Even though multiple errors may have occurred on a data space entry, only the first occurrence per entry is reported.

Information Passed:

• Data space index (binary zeros if during index creation)	System pointer
• Number of error descriptions	Bin(2)
• Reserved	Char(4)
• Error description (repeated)	Char(14)
– Data space number	Bin(2)
– Ordinal entry number	Bin(4)
– Field number	Bin(2)
– Error type	Char(2)
– Field location	Char(1)
Hex 03 = Intermediate key buffer	
Unassigned values reserved	
– Reserved	Char(3)

The exception field definitions are the same as for the hex 1238 exception (derived field operation error).

Instructions Causing Exception:

- Create Data Space Index
- Data Base Maintenance
- Activate Cursor (over delayed maintenance index)
- Insert Sequential Data Space Entries
- Signal Exception

1240 Invalid Entry Definition Table

During validation of an entry definition table supplied at Create Data Space, Create Data Space Index or Create Cursor, an invalid field may be found.

Information Passed

• Error type	Char(2)
— Invalid table offset	Bit 1
— Invalid field type	Bit 2
— Invalid field length	Bit 3
— Invalid field precision	Bit 4
— Invalid entry length	Bit 5
— Invalid default length	Bit 6
— Invalid default offset	Bit 7
— Invalid field allocation	Bit 8
— Invalid null type	Bit 9
— Type/variable length mismatch	Bit 10
— Invalid field modifier	Bit 11
— Reserved area not zero	Bit 12
— Reserved (binary zero)	Bit 13-15
• Data Space Number (0 for Create Data Space instruction)	Bin(2)
• DSE field number (0 if error type is not field related)	Bin(2)

Instructions Causing Exception:

- Create Cursor
- Create Data Space
- Create Data Space Index
- Signal Exception

1241 ISV parameter value in runtime data pointer array not correct.

Information Passed

• Reason code	Bin(2)
• Array element number	Bin(2)
• Create time type	Bin(2)
• Create time length	Char(2)
• Runtime type	Bin(2)
• Runtime length	Char(2)

Instructions Causing Exception:

1242 Non-Unique Fanout on Unique Join

On a Unique fanout join more than one record in the "join to" file that satisfied the relationship was encountered.

Information Passed

- Data Space Number — Bin(4)
- Ordinal # of candidate record (the first key match) — Bin(4)
- Ordinal # of record identified as the duplicate key — Bin(4)

Instructions Causing Exception:

- Set cursor
- Retrieve Sequential

1243 DDAT had an error.

An error was found in a DDAT.

Information Passed

- Error type — Bin(2)

  01 – No valid template size

  02 – No valid number of DDATs

  03 – No valid reserved are

  04 – No valid DDAT length

  05 – No valid format code

  06 – No valid date separator

  07 – No valid time separator

  08 – No valid time zone

  09 – No valid time zone, hour

  10 – No valid time zone, minute

  11 – No valid calendar information

  12 – No valid calendar, month

  13 – No valid calendar, year

  14 – No valid century definition

  15 – No valid century definition current century

  16 – No valid century definition century division

  17 – No valid calendar table offset

  18 – No valid reserved area

  19 – No valid number of era table elements

  20 – No valid era origin date

  21 – No valid era name

  22 – No valid era reserved area

  23 – No valid number of calendar table elements

24 -- No valid calendar effective date

25 -- No valid calendar type

26 -- No valid calendar reserve area

- DDAT number in error — Bin(2)
- Era number — Bin(2)
- Calender number — Bin(2)

Instructions Causing Exception:

1262 Parent index cannot be used to create new index

The parent index referenced in the create index instruction is internally damaged. When the index was converted by the system on a previous release of the operating system, the concatenated key within the index was improperly converted.

Instructions Causing Exception:

16 Exception Management

1601 Exception Description Status Invalid

The tested exception description was not in the deferred state.

Instructions Causing Exception:

- Test Exception
- Signal Exception

1602 Exception State of Process Invalid

An attempt was made to retrieve exception data or resignal an exception when the process is not in an exception handling state; that is, the process is not in an external program, internal entry point, or branch point exception handler. The re-signal option is valid only for an external exception handler.

Instructions Causing Exception:

- Signal Exception
- Retrieve Exception Data

1603 Invalid Invocation Address

The invocation address specified in the space/invocation pointer on the instruction did not represent an existing program invocation.

Information Passed:

- Invocation pointer

Instructions Causing Exception:

- Return From Exception
- Sense Exception Description
- Signal Exception
- Send Process Message
- Cancel Invocations

1604 Retry/resume Invalid

An attempt was made to either retry a failed instruction or resume at an instruction after the instruction that failed in an invocation were retry or resume is not allowed.

Information Passed:

- Invocation pointer

Instruction causing the exception:

- Return From Exception

1605 No inquiry message found for reply message

An attempt was made to send a reply message for an inquiry message that does not exist.

Instruction causing the exception:

- Send Process Message

1A Lock State

1A01 Invalid Lock State

The lock enforcement rule or rules were violated when an attempt was made to access an object.

Information Passed:

- System pointer to the object

Instructions Causing Exception:

- All instructions that enforce the lock rules
- Signal Exception

1A02 Lock Request Not Grantable

The lock request cannot be granted immediately and neither the synchronous nor asynchronous wait option was specified.

Information Passed:

• Pointer to lock request template	Space pointer
• Failing request number (relative entry position)	Bin(2)

Instructions Causing Exception:

- Lock Object
- Signal Exception

1A03 Invalid Unlock Request

An attempt was made to unlock a lock state not held by the current requesting process.

Information Passed:

• Pointer to unlock request template	Space pointer
• Number of requests not unlocked	Bin(2)
• Request number (relative entry position for each lock not unlocked)	Bin(2)

Instructions Causing Exception:

- Unlock Object
- Signal Exception

1A04 Invalid Object Lock Transfer Request

An attempt was made to transfer locks that were not held by the transferring process, or the transfer lock request was not granted because the lock granting rules would have been violated.

Information Passed:

• Pointer to lock transfer request template	Space pointer
• Number of requests not transferred	Bin(2)
• Request number (relative entry position for each lock not transferred)	Bin(2)

Instructions Causing Exception:

- Transfer Object Lock
- Signal Exception

1A05 Invalid Space Location Unlocked

An attempt was made to unlock a space location lock not held by the current requesting process.

Information Passed:

• Space location process attempted to unlock	Space pointer
• Unlock request	Char(1)

Instructions Causing Exception:

- Unlock Space Location
- Signal Exception

1E Machine Observation

1E01 Program Not Observable

The program observation functions were destroyed for the program referenced by the executing instruction.

Information Passed:

- Program — System pointer

Instructions Causing Exception:

- Materialize Instruction Attributes
- Materialize Invocation
- Signal Exception

1E02 Invocation Not Found

An invocation matching the specified criteria was not found.

Instructions Causing Exception:

- Find Relative Invocation
- Signal Exception

1E03 Invalid D-Code Instruction

The Debug Interpreter (DBGINT) MI instruction has detected an invalid field of a D-Code Instruction. The instruction and field are identified in the information passed (described below).

Information Passed:

- Erroneous D-Code instruction — Char(40)

  The erroneous D-Code instruction is formatted as a hexadecimal string. Each hexadecimal digit represents 4 bits. D-Code instructions occupy twenty bytes.

- D-Code instruction index — Bin(4)

  The D-Code instruction index is the index of the erroneous D-Code instruction in the D-Code instruction array. It is the value of the program counter (PC) register when the Debug Interpreter MI instruction issued this exception.

- Erroneous D-Code instruction field — Bin(4)

  The erroneous D-Code instruction field is an integer code indicating which field of the D-Code instruction is in error. The table below associates codes with corresponding fields.

Table 23-1. Erroneous D-Code Instruction Field

Code	D-Code Field	Explanation
0	None	No field is in error
1	D-Code operator	The D-Code operator field is invalid.
2	Type operand	The type operand field is invalid.
3	Label operand	The label operand is invalid.
4	Reference operand	The reference operand is invalid.
5	Constant operand	The constant operand is invalid.
6	Invocation delta operand	The invocation delta operand is invalid.
7	Function operand	The function operand is invalid.
8	Conversion operand	The conversion operand is invalid.

- Erroneous reference operand field BIN(4)

  The erroneous reference operand field is an integer code indicating which field of the reference operand is in error. The table below associates codes with corresponding fields.

Table 23-2. Erroneous Reference Operand Field

Code	Reference Operand Field	Explanation
0	None	No field is in error
1	Type field	The type field is invalid.
2	Identifier field	The identifier field is invalid.
3	Storage class field	The storage class field is invalid.
4	Invocation delta field	The invocation delta field is invalid.

- Erroneous constant operand field BIN(4)

  The erroneous constant operand field is an integer code indicating which field of the constant operand is in error. The table below associates codes with corresponding fields.

Table 23-3. Erroneous Constant Operand Field

Code	Constant Operand Field	Explanation
0	None	No field is in error
1	Type field	The type field is invalid.
2	Cardinal field	The value of the cardinal constant is invalid.
3	Integer field	The value of the integer constant is invalid.
4	Character	The value of the 8-bit character constant is invalid.
5	Real	The value of the 64-bit real constant is invalid.
6	Enumeration field	The value of the enumeration constant is invalid.
7	String field	The value of the string offset is invalid.

1E04 DBGINT error

The Debug Interpreter (DBGINT) MI instruction has detected an error. The instruction and error are identified in the information passed (described below).

Information Passed:

- Erroneous D-Code instruction Char(40)

  The erroneous D-Code instruction is formatted as a hexadecimal string. Each hexadecimal digit represents 4 bits. D-Code instructions occupy twenty bytes.

- D-Code instruction index Bin(4)

  The D-Code instruction index is the index of the erroneous D-Code instruction in the D-Code instruction array. It is the value of the program counter (PC) register when the Debug Interpreter MI instruction issued this exception.

- Error field Bin(4)

  The table below associates codes with corresponding errors.

Table 23-4 (Page 1 of 2). Error Field

Code	Error Field	Explanation
0	None	No field is in error

Table 23-4 (Page 2 of 2). Error Field		
Code	**Error Field**	**Explanation**
1	Mark Pointer	The Mark Pointer is invalid.
2	Storage class	The storage class is invalid.
3	Results space	The results space is not large enough.
4	Stack error	A stack entry is in error.
5	Register	A register in the directory is invalid.
6	Conversion specified	The conversion specified is invalid.

1E05 DBGINT error

An error occurred on the operation while executing the debug interpreter (DBGINT).

Information Passed:

- Operation field Bin(4)

  The table below associates codes with corresponding operations.

Table 23-5. Opcode Field	
Code	**Operation**
0	None
1	Equal
2	Not equal
3	Greater than
4	Greater than or equal to
5	Less than
6	Less than or equal to
19	And
20	Or
21	Exclusive Or
22	Not
23	Add
24	Subtract
25	Negate
26	Multiply
27	Divide
31	Increment
32	Decrement
33	Modulo

- Data Type field Bin(4)

  The table below associates codes with corresponding data types.

Table 23-6 (Page 1 of 2). datatyp Field	
Code	**Data Type**
1	Char_8

Table 23-6 (Page 2 of 2). datatyp Field

Code	Data Type
2	Char_16
3	Bool32
4	Card_16
5	Card_32
6	Int_16
7	Int_32
8	Real_32
9	Real_64
10	SpcPtr
11	FncPtr
12	MchAdr
13	Record
14	Array
15	Enum
16	String

20 Machine Support

2001 Diagnose

An error or discrepancy was found when a Diagnose instruction was processed.

Information Passed:

- Space element to the subelement in the operand 2 object that was being processed
- Data — Bin(4)
  - Subidentifier unique to the requested function — Bin(2)
  - Indicator of the pointer in operand 2 that was being processed — Bin(2)

Instructions Causing Exception:

- Diagnose
- Signal Exception

2002 Machine Check

A machine malfunction affecting system-wide operation has been detected during execution of an instruction in this process.

Information Passed:

- Timestamp that gives the current value of the machine time-of-day clock. — Char(8)
- Error code indicating nature of machine check. (This value is machine-dependent and is only defined in the machine service documentation.) — Char(2)
- Reserved (binary 0) — Char(6)
- VLOG dump ID — Char(8)
- Error class — Bin(2)

  The error class codes for the type of damage detected are as follows:

  Hex 0000 = Unspecified abnormal condition
  Hex 0002 = Logically invalid device sector
  Hex 0003 = Device failure

- Auxiliary storage device indicator — Bin(2)

  This field is defined for error classes hex 0002 and hex 0003. It is the OU number of the failing device or 0 for a main storage failure.

- Reserved (binary 0) — Char(100)

Instructions Causing Exception:

- Any instruction
- Signal Exception

2003 Function Check

The executing instruction has failed unexpectedly during execution within the process.

Information Passed:

- Timestamp giving the current value of the machine time-of-day clock. — Char(8)

- Error code indicating the nature of the function check. (This value is machine-dependent.) Char(2)
- Reserved (binary 0) Char(6)
- VLOG dump ID Char(8)
- Error class Bin(2)

  The error class codes for the type of damage detected are as follows:

  Hex 0000 = Unspecified abnormal condition
  Hex 0002 = Logically invalid device sector
  Hex 0003 = Device failure
- Auxiliary storage device indicator Bin(2)

  This field is defined for error classes hex 0002 and hex 0003. It is the OU number of the failing device or 0 for a main storage failure.
- Reserved (binary 0) Char(100)

Instructions Causing Exception:

- Any instruction
- Signal Exception

22 Object Access

2201 Object Not Found

An attempt to resolve addressability into a system pointer was not successful for one of the following reasons:

- The named object was not located in the context specified in the symbolic address or in any context referenced in the name resolution list.
- An object with a corresponding name was found but the user profile(s) governing execution of the instruction did not have the authority required for resolution.

Information Passed:

• Object identification	Char(32)
– Object type	Char(1)
– Object subtype	Char(1)
– Object name	Char(30)
• Required authorization	Char(2)

Instructions Causing Exception:

- Any instruction that references an object through a system pointer
- Signal Exception

2202 Object Destroyed

An attempt was made to reference an object that no longer exists or part of it no longer exists.

This exception may not be signaled for operations which refer to the value of a space pointer machine object, but which do not attempt to reference the space data the pointer addresses. This can also be signaled when an associated space of the object is missing. The following instructions may not signal this exception upon references to a space pointer machine object.

- Add Space Pointer
- Compare Pointer for Space Addressability
- Compare Pointers for Equality
- Compare Space Addressability
- Set Space Pointer
- Set Space Pointer with Displacement
- Set Space Pointer from Pointer
- Subtract Space Pointer Offset
- Subtract Space Pointers For Offset

See the particular instruction description for more detail.

Instructions Causing Exception:

- Any instruction that references an object through a system pointer, a space pointer, or a data pointer
- Any instruction that references a scalar or a pointer operand when the object and the space containing the scalar or pointer have been destroyed
- Signal Exception

24 Pointer Specification

2401 Pointer Does Not Exist

A pointer reference was made to a storage location in a space that does not contain a pointer data object, or a reference was made to a space pointer machine object that was not set to address a space.

This exception may not be signaled for operations which refer to the value of a space pointer machine object, but which do not attempt to reference the space data the pointer addresses. The following instructions may not signal this exception upon references to a space pointer machine object.

- Add Space Pointer
- Compare Pointer for Space Addressability
- Compare Space Addressability
- Set Space Pointer
- Set Space Pointer With Displacement
- Set Space Pointer from Pointer
- Subtract Space Pointer Offset

See the particular instruction description for more detail.

Instructions Causing Exception:

- Any instruction that has pointer operands
- Any instruction that references a base operand (scalar or pointer) when the base pointer is not a space pointer
- Any instruction that allows a scalar defined by a data pointer to be an operand
- Any instruction that requires a pointer as part of the input template
- Signal Exception

2402 Pointer Type Invalid

An instruction has referenced a pointer object that contains an incorrect pointer type for the operation requested.

Instructions Causing Exception:

- Any instruction that has pointer operands
- Any instruction that contains a base operand (scalar or pointer) when the base pointer is not a space pointer
- Any instruction that allows a scalar defined by a data pointer to be an operand
- Any instruction that requires a pointer as part of the input template
- Signal Exception

2403 Pointer Addressing Invalid Object Type

An instruction has referenced a system pointer that addresses an incorrect type of system object for this operation.

Information Passed:

- The invalid system pointer — System pointer

Instructions Causing Exception:

- Any instruction that references a system pointer, either as an operand or within a template operand, and that requires a specific object type as a part of its operation
- Signal Exception

2404 Pointer Not Resolved

The operation did not find a resolved system pointer. For example, NRL (name resolution list) entries must be resolved system pointers that address contents.

Information Passed:

- The invalid pointer — System pointer

Instructions Causing Exception:

- Resolve System Pointer
- Any instruction that causes a system pointer to be implicitly resolved when the NRL is used in the resolution. All entries in the NRL must be resolved.
- Resolve Data Pointer
- Any instruction that causes a data pointer to be implicitly resolved. All activation entries in the process must contain a resolved pointer to the associated program.
- Signal Exception

26 Process Management

2602 Queue Full

An attempt was made to enqueue a message to a queue that is full and is not extendable.

Information Passed:

- Queue for which the enqueue was attempted — System pointer

Instructions Causing Exception:

- Enqueue
- Request I/O
- Signal Exception

2A Program Creation

2A01 Program Header Invalid

The data in the program header was invalid.

Instructions Causing Exception:

- Signal Exception

2A02 ODT Syntax Error

The syntax (bit setting) of an ODT (object definition table) entry was invalid.

Information Passed:

- ODT entry number Char(2)

Instructions Causing Exception:

- Signal Exception

2A03 ODT Relational Error

An ODT (object definition table) entry reference to another ODT entry was invalid or missing.

Information Passed:

- ODT entry number Char(2)

Instructions Causing Exception:

- Signal Exception

2A04 Operation Code Invalid

One of the following conditions occurred.

- The operation code did not exist.
- The optional form was not allowed.

Information Passed:

- Instruction number of the instruction being analyzed UBin(2)

Instructions Causing Exception:

- Signal Exception

2A05 Invalid Op Code Extender Field

The branch/indicator options were invalid.

Information Passed:

- Instruction number of the instruction being analyzed UBin(2)

Instructions Causing Exception:

- Signal Exception

2A06 Invalid Operand Type

One of the following conditions was detected:

- An operand was not the required type (signed immediate, immediate, constant data object, scalar data object, pointer data object, null, branch point, or instruction definition list).
- An operand was described as an immediate or constant data object. However, the instruction specifies that the operand be modified to something other than an immediate or con-

stant data object, or the instruction does not allow an immediate or constant data object operand.

- The operand type specified is not a valid operand type.
- The type of one operand does not satisfy a required relationship with the type of another operand.

Information Passed:

- Instruction number of the instruction being analyzed — UBin(2)

Instructions Causing Exception:

- Signal Exception

2A07 Invalid Operand Attribute

One of the following conditions was detected:

- An operand did not have the attributes required by the instruction (character, packed decimal, zoned decimal, binary, floating-point, scalar, array, assumed, overlay, restricted, open, based, explicitly based).
- The attributes of one operand did not match the required attributes of another operand.
- At least one operand in the argument list for a Transfer Control instruction was specified as automatic.
- The receiver for an instruction specified with the optional round form has the floating-point attribute.

Information Passed:

- Instruction number of the instruction being analyzed — UBin(2)

Instructions Causing Exception:

- Signal Exception

2A08 Invalid Operand Value Range

One of the following conditions was detected:

- An operand was a constant or immediate data object and was used as an index into an array or indicated a position in a character string, but it was outside the range of the array or character string.
- An operand was a constant or immediate data object and did not conform to the value required by the instruction.
- The operand immediate value is outside of the accepted range. The valid range for an unsigned immediate value is equal to or greater than 0 and less than or equal to 8191. The valid range for a signed immediate value is from negative 4096 through a positive 4095.

Information Passed:

- Instruction number of the instruction being analyzed — UBin(2)

Instructions Causing Exception:

- Signal Exception

2A09 Invalid Branch Target Operand

One of the following conditions was detected:

- An operand was not an instruction pointer, branch point, instruction number, or relative instruction number.
- An operand was an instruction number or relative instruction number but was outside the range of the program.
- A branch target operand identified an instruction that was not indicated as a branch target.

Information Passed:

- Instruction number of the instruction being analyzed — UBin(2)

Instructions Causing Exception:

- Signal Exception

2A0A Invalid Operand Length

One of the following conditions was detected:

- The length attribute of an operand was not greater than or equal to the length required by the instruction.
- The length attribute of an operand was invalid based on its relationship to the length attribute of another operand in the same instruction.
- The length attribute of a decimal operand exceeds 15 digits when specified in conjunction with a floating-point operand.
- A decimal operand has an invalid integer or fractional digit length in relationship to that required by the instruction.

Information Passed:

- Instruction number of the instruction being analyzed — UBin(2)

Instructions Causing Exception:

- Signal Exception

2A0B Invalid Number of Operands

The number of arguments in a Call Internal instruction was not equal to the number of parameters in the called entry point.

Information Passed:

- Instruction number of the instruction being analyzed — UBin(2)

Instructions Causing Exception:

- Signal Exception

2A0C Invalid Operand ODT Reference

The ODT reference was not within the range of the ODV.

Information Passed:

- Instruction number of the instruction being analyzed — UBin(2)

Instructions Causing Exception:

- Signal Exception

2A0D Reserved Bits Are Not Zero

The reserved bits in an opcode or operand are nonzero.

Information Passed:

- Instruction number of the instruction being analyzed — UBin(2)

Instructions Causing Exception:

- Signal Exception

2A10 Automatic storage for procedure exceeds maximum

The object was not created because an internal system limit was reached. Not enough automatic storage was available to allocate a data object within a procedure.

Reduce the number or size of automatic data objects within the procedure.

Information Passed:

Field	Type
• Current automatic storage offset	UBin(4)
• Maximum automatic storage offset	UBin(4)
• Data object size	UBin(4)
• Data object dictionary index	UBin(4)
• Data object dictionary entry information	Char(128)
– Entry length	UBin(2)
– Entry	Char(*)
• Data object dictionary entry offset	UBin(4)
• Procedure dictionary index	UBin(4)
• Procedure dictionary entry information	Char(128)
– Entry length	UBin(2)
– Entry	Char(*)
• Procedure dictionary entry offset	UBin(4)
• Machine log note ID	Char(8)
• reserved	Char(8)

Instructions Causing Exception:

- Signal Exception

2A11 Machine automatic storage exceeds maximum

The object was not created because an internal system limit was reached. Not enough machine automatic storage was available to allocate a data object within a procedure.

Reduce the size and complexity of the procedure or reduce the level of optimization used to create the object.

Information Passed:

Field	Type
• Current machine automatic storage offset	UBin(4)
• Maximum machine automatic storage offset	UBin(4)
• Data object size	UBin(4)
• Data object dictionary index	UBin(4)
• Data object dictionary entry information	Char(128)

– Entry length	UBin(2)
– Entry	Char(*)
• Data object dictionary entry offset	UBin(4)
• Procedure dictionary index	UBin(4)
• Procedure dictionary entry information	Char(128)
– Entry length	UBin(2)
– Entry	Char(*)
• Procedure dictionary entry offset	UBin(4)
• Machine log note ID	Char(8)
• reserved	Char(8)

Instructions Causing Exception:

- Signal Exception

2A12 Data type or length of initial value not valid

The object was not created because an initialization table entry is not valid. An initial value for a data object has either a data type that is not valid or a length that is not valid.

There is a problem in the compiler. Report this problem to the supplier of the compiler.

Information Passed:

• Initial value data type	UBin(4)
• Initial value length	UBin(4)
• Initialization table index	UBin(4)
• Initialization table entry information	Char(128)
– Entry length	UBin(2)
– Entry	Char(*)
• Initialization table entry offset	UBin(4)
• Machine log note ID	Char(8)
• reserved	Char(8)

Instructions Causing Exception:

- Signal Exception

2A14 Static data initialized to address of automatic data

The object was not created because an initialization table entry is not valid. The static data cannot be initialized to the address of automatic data.

There is a problem in the compiler. Report this problem to the supplier of the compiler.

Information Passed:

• Initialization table index	UBin(4)
• Initialization table entry information	Char(128)
– Entry length	UBin(2)
– Entry	Char(*)
• Initialization table entry offset	UBin(4)
• Static data dictionary index	UBin(4)

- Static data dictionary entry information — Char(128)
  - Entry length — UBin(2)
  - Entry — Char(*)
- Static data dictionary entry offset — UBin(4)
- Automatic data dictionary index — UBin(4)
- Automatic data dictionary entry information — Char(128)
  - Entry length — UBin(2)
  - Entry — Char(*)
- Automatic data dictionary entry offset — UBin(4)
- Machine log note ID — Char(8)
- reserved — Char(8)

Instructions Causing Exception:

- Signal Exception

2A15 Initial value for static data not valid

The object was not created because an initialization table entry is not valid. Static data is initialized, but the initial value is not valid.

There is a problem in the compiler. Report this problem to the supplier of the compiler.

Information Passed:

- Reason code — UBin(4)

  1 = Reserved initialization entry not binary zeros.
  2 = Initial length specified when zero is expected.
  3 = Target of initialization is too small.
  4 = Procedure not target of initCodeAddress.
  5 = PEP specified as target of initCodeAddress.
  6 = Offset specified when zero is expected.
  7 = Rep specified when zero is expected.
  8 = Bitfield initialization was not terminated by entry with dictNo=0.
  9 = Bitfield length is not a valid value.
  10 = Length is not large enough to initialize bitfield.
  11 = A dictionary entry (or its owner) did not have _TOBEMAPD = TRUE.
  12 = Attempt made to initialize storage with invalid storage class.
  13 = Invalid dictionary index.
  14 = Invalid target dictionary index.
  15 = Target of initialized space pointer does not specify addr_taken.

- Initialization table index — UBin(4)
- Initialization table entry information — Char(128)
  - Entry length — UBin(2)
  - Entry — Char(*)
- Initialization table entry offset — UBin(4)
- Machine log note ID — Char(8)
- reserved — Char(8)

Instructions Causing Exception:

- Signal Exception

2A16 Number of procedures exceeds maximum allowed

The object was not created because an internal system limit was reached. The number of procedures exceeds the maximum allowed.

Reduce the number of procedures.

Information Passed:

- Maximum procedures — UBin(4)
- Procedure dictionary index — UBin(4)
- Procedure dictionary entry information — Char(128)
  - Entry length — UBin(2)
  - Entry — Char(*)
- Procedure dictionary entry offset — UBin(4)
- Machine log note ID — Char(8)
- reserved — Char(8)

Instructions Causing Exception:

- Signal Exception

2A17 Type table entry not valid

The object was not created because the type table contains an entry that is not valid.

There is a problem in the compiler. Report this problem to the supplier of the compiler.

Information Passed:

- Reason code — UBin(4)

  1 = Length of data object is invalid.
  2 = Type of data object is invalid.
  3 = Bit field displacement is invalid.
  4 = Bit field width is invalid.
  5 = Reserved area is invalid.

- Type table index — UBin(4)
- Type table entry information — Char(128)
  - Entry length — UBin(2)
  - Entry — Char(*)
- Type table entry offset — UBin(4)
- Machine log note ID — Char(8)
- reserved — Char(8)

Instructions Causing Exception:

- Signal Exception

2A18 Alias table entry not valid

The object was not created because the alias table contains an entry that is not valid.

There is a problem in the compiler. Report this problem to the supplier of the compiler.

Information Passed:

- Reason code — UBin(4)

  1 = Record type of alias table entry is invalid.

2 = Reserved field in alias class not binary zeros.
3 = Invalid alias class id.
4 = Invalid alias class size.
5 = Invalid dictionary id.
6 = Dictionary id is not for a data object.
7 = Reserved field in alias union not binary zeros.
8 = Invalid alias union id.
9 = Invalid alias union size.
10 = Invalid class id.

- Alias table entry information — Char(128)
  - Entry length — UBin(2)
  - Entry — Char(*)
- Alias table entry offset — UBin(4)
- Machine log note ID — Char(8)
- reserved — Char(8)

Instructions Causing Exception:

- Signal Exception

2A19 Size of constants exceeds maximum

The object was not created because an internal system limit was reached. Not enough space was available to add a constant within a procedure.

Reduce the number or size of constants within the procedure.

Information Passed:

- Current size of all constants — UBin(4)
- Maximum size of all constants — UBin(4)
- Constant size — UBin(4)
- Constant dictionary index — UBin(4)
- Constant dictionary entry information — Char(128)
  - Entry length — UBin(2)
  - Entry — Char(*)
- Constant dictionary entry offset — UBin(4)
- Procedure dictionary index — UBin(4)
- Procedure dictionary entry information — Char(128)
  - Entry length — UBin(2)
  - Entry — Char(*)
- Procedure dictionary entry offset — UBin(4)
- Machine log note ID — Char(8)
- reserved — Char(8)

Instructions Causing Exception:

- Signal Exception

2A1A Procedure size exceeds maximum

The object was not created because an internal system limit was reached. Not enough space was available for a procedure.

Reduce the size of the procedure.

Information Passed:

• Maximum procedure size	UBin(4)
• Procedure dictionary index	UBin(4)
• Procedure dictionary entry information	Char(128)
– Entry length	UBin(2)
– Entry	Char(*)
• Procedure dictionary entry offset	UBin(4)
• Machine log note ID	Char(8)
• reserved	Char(8)

Instructions Causing Exception:

- Signal Exception

2A1B Instruction stream not valid

The object was not created because the sequence of instruction stream objects is not valid.

There is a problem in the compiler. Report this problem to the supplier of the compiler.

Information Passed:

• Reason code	UBin(4)

1 = Undefined MI operation code.
2 = Invalid type table index.
3 = Invalid dictionary number.
4 = Invalid label identifier.
5 = One or more instructions are out of sequence. Typically signaled when the first instruction in a procedure is not ENT.
6 = The dictionary id on the END instruction does not match the dictionary id on first ENT instruction in the current procedure.
7 = An operand of a an MI instruction has has an invalid value.
8 = The data type of the stack operand(s) is invalid for the operation being attempted.
9 = The length of the stack operand(s) is invalid for the operation being attempted.
10 = An operand of a load instruction has has an invalid data type.
11 = An object which is loaded has scope greater than the current procedure's scope.
12 = The target of a STR operation in the new MI instruction stream is not a variable, or the target is a register variable and either offset or typeNo are nonzero.
13 = The MI value stack is not empty at a label, jump, or other point in the instruction stream at which the MI architecture asserts it must.
14 = The MI value stack has fewer items on it than need to be popped.
15 = The values retrieved from the MI value stack are not compatible.
16 = The data type of the stack operands is invalid for the comparison being attempted, (i.e. UNORD on binary dt).
17 = There is control flow in the NMI program which reaches either a secondary entry point or an END instruction.
18 = No exception handlers are currently enabled when a DHNDLR is encountered.
19 = A DHNDLR instruction specifies an exception handler that was not the last one enabled.
20 = A LAB instruction was encountered where the label qualifier was not one of: FwdOnlyLabel (1) or GeneralLabel (2).
21 = An object in the dictionary with the "is_reg" attribute lacks one or more of the required characteristics for inclusion in the machine storage class. For example, the dictionary object is the target of a LDA operation, or the dictionary object has

its data type or length superseded with a type table entry on a load or store operation.

22 = A·blocked built-in function is called.
23 = An invalid built-in function number in CALLBI instruction.
24 = An invalid alias union id in PALI instruction.
25 = An invalid pali flag in PALI instruction.
26 = A switch-specific operation (i.e. CASE, DEFAULT, BRK, ENDS) was encountered outside of a SWE-ENDS block.
27 = Two CASE ranges were found with one or more elements in common, violating the requirement that all CASE ranges be totally disjoint.
28 = The low selector value for the range of a CASE statement is greater than the high selector value.
29 = A second DEFAULT case was found for a switch statement in the instruction stream.
30 = Built-in OPMPARMADDR or OPMPARMCNT is not invoked from PEP.
31 = A required built-in function argument was not specified.
35 = The number of arguments specified for a built-in function is incorrect.
36 = The null op mask specifies an argument that is not valid.
37 = A switch statement is still active (i.e. the ENDS operation for an open switch has not been encountered) at the END of a procedure.
38 = All three labels on a MiCMPB operation were found to equal zero.
39 = Buit-in function NPMPARMLISTADDR was invoked from PEP.
40 = A label that was the target of a jump (FJP,TJP,CMPB,UJP) was not found in the instruction stream.
41 = A label that is in the dictionary, and that is indicated therein to have had its address taken, was not found in the instruction stream.
42 = A label constant on the stack when IJMP was encountered was now owned by the current procedure being translated.
43 = A data object which has been allocated to a machine storage class is the target of a LDA instruction, has its address referenced from the initialization component, or is used as an exception handler communication area. This situation can occur if the front end did not set the addr_taken flag for the data object (or for the data object's owner if it is the member of a aggregate).

- Current instruction count UBin(4)
- Data offset UBin(4)
- Current instruction offset UBin(4)
- Current instruction opcode UBin(4)
- Current statement number UBin(4)
- Machine log note ID Char(8)
- reserved Char(8)

Instructions Causing Exception:

- Signal Exception

2A1C Size of literals exceeds maximum

The object was not created because an internal system limit was reached. Not enough space was available in the literal pool to add the literal.

Reduce the number or size of literals.

Information Passed:

- Current size of all literals UBin(4)
- Maximum size of all literals UBin(4)

• Literal size	UBin(4)
• Literal information	Char(128)
– Literal length	UBin(2)
– Literal	Char(*)
• Machine log note ID	Char(8)
• reserved	Char(8)

Instructions Causing Exception:

- Signal Exception

2A1D Dictionary entry not valid

The object was not created because the dictionary contains an entry that is not valid.

There is a problem in the compiler. Report this problem to the supplier of the compiler.

Information Passed:

• Reason code	UBin(4)

1 = Undefined data type.
2 = Invalid reserved entry.
3 = Parent of data object is IsReg variable.
4 = Invalid alias union number.
5 = Invalid alignment.
6 = Invalid data type.
7 = Invalid flags.
8 = Invalid flags2.
9 = Invalid handler.
10 = Invalid handler comm area.
11 = Invalid handler level.
12 = Invalid handler type.
13 = Invalid length.
14 = Invalid linkage.
15 = Invalid literal type.
16 = Invalid offset.
17 = Invalid owner.
18 = Invalid procedure type.
19 = Invalid replication.
20 = Invalid reserved1.
21 = Invalid reserved2.
22 = Invalid reserved3.
23 = Invalid return length.
24 = Invalid return type.
25 = Invalid scope.
26 = Invalid storage class.
27 = Invalid extended description.
28 = Invalid control action.

• Dictionary index	UBin(4)
• Dictionary entry information	Char(128)
– Entry length	UBin(2)
– Entry	Char(*)
• Dictionary entry offset	UBin(4)
• Machine log note ID	Char(8)

- reserved Char(8)

Instructions Causing Exception:

- Signal Exception

2A1E Level of machine interface not supported on target release

The object was not created because the software release level of the machine interface that was used is not compatible with the software release level targeted.

There is a problem in the compiler. Report this problem to the supplier of the compiler.

Information Passed:

- Machine interface level used UBin(2)
- Machine interface level targeted UBin(2)
- Machine log note ID Char(8)
- reserved Char(8)

Instructions Causing Exception:

- Signal Exception

2A1F Size of dictionary exceeds maximum

The object was not created because an internal system limit was reached. Not enough space was available in the dictionary.

Reduce the size and complexity of the procedure or reduce the level of optimization used to create the object.

Information Passed:

- Procedure dictionary index UBin(4)
- Procedure dictionary entry information Char(128)
  - Entry length UBin(2)
  - Entry Char(*)
- Procedure dictionary entry offset UBin(4)
- Machine log note ID Char(8)
- reserved Char(8)

Instructions Causing Exception:

- Signal Exception

2A20 Internal machine operation not valid

The object was not created because the internal representation of a machine operation is not valid. The machine interface template probably contains an error.

There is a problem in the compiler. Report this problem to the supplier of the compiler.

Information Passed:

- Name of machine operation Char(15)
- Statement number UBin(4)
- Machine log note ID Char(8)
- reserved Char(8)

Instructions Causing Exception:

- Signal Exception

2A21 Size of internal binding table exceeds maximum

The object was not created because an internal system limit was reached. Not enough space was available in an internal binding table.

Reduce the number of external data objects in the program.

Information Passed:

• Current identifier	UBin(4)
• Maximum identifier	UBin(4)
• Machine log note ID	Char(8)
• reserved	Char(8)

Instructions Causing Exception:

- Signal Exception

2A5E An error was detected in a static storage definition or initialization.

An attempt was made to declare a static variable larger than the maximum size, or initialize a static variable beyond its defined length.

Correct the invalid initialization.

Information Passed:

• Reason code	UBin(2)
• VLIC log note ID	Char(8)

Instructions Causing Exception:

- Signal Exception

2A5F Overlapping initializations not valid.

Two initialization entries within the initialization component are for the same storage location.

Contact your service representative to report the problem.

Instructions Causing Exception:

- Signal Exception

2A60 Dictionary ID is not valid.

A dictionary ID specified on one of the binding messages generated by the optimizing translator was not valid.

Contact your service representative to report the problem.

Information Passed:

• Dictionary ID	UBin(4)
• Reason code	UBin(2)

Instructions Causing Exception:

- Signal Exception

2A61 Binding specification value not valid.

A value within the binding specifications component was in error.

Contact your service representative to report the problem.

Information Passed:

• Offset of error	UBin(4)

- Length of error UBin(2)
- Reason code UBin(2)

Instructions Causing Exception:

- Signal Exception

2A62 Copyright component value not valid.

A value within the copyright component was in error.

Contact your service representative to report the problem.

Information Passed:

- Offset of error UBin(4)
- Length of error UBin(2)
- Reason code UBin(2)

Instructions Causing Exception:

- Signal Exception

2A63 Module limitation exceeded.

An internal part of the module exceeds allowed size.

Contact your service representative to report the problem.

Information Passed:

- Reason code UBin(2)

Instructions Causing Exception:

- Signal Exception

2AA0 Attempt to delete part that may not be deleted.

The module was created with observable data that may not be deleted. An attempt was made to delete that data.

Contact your service representative to report the problem.

Information Passed:

- Reason code UBin(4)

Instructions Causing Exception:

2AB0 Object list referential extension not valid.

An entry in the object list was not valid.

Contact your service representative to report the problem.

Information Passed:

- Pointer to the object list System pointer
- Byte offset UBin(4)
- Entry index UBin(4)
- Field length UBin(4)
- Bit offset UBin(2)
- Reason code UBin(2)

Instructions Causing Exception:

2AB1 Symbol resolution list referential extension not valid.

An entry in the symbol resolution list was not valid.

Contact your service representative to report the problem.

Information Passed:

• Pointer to the symbol resolution list	System pointer
• Byte offset	UBin(4)
• Entry index	UBin(4)
• Field length	UBin(4)
• Bit offset	UBin(2)
• Reason code	UBin(2)

Instructions Causing Exception:

2AB2 Service program export list referential extension not valid.

An entry in the service program export list was not valid.

Contact your service representative to report the problem.

Information Passed:

• Pointer to the service program export list	System pointer
• Byte offset	UBin(4)
• Entry index	UBin(4)
• Field length	UBin(4)
• Bit offset	UBin(2)
• Reason code	UBin(2)

Instructions Causing Exception:

2AB3 Secondary associated spaces list referential extension not valid.

An entry in the secondary associated spaces list was not valid

Contact your service representative to report the problem.

Information Passed:

• Pointer to the secondaray associated spaces list	System pointer
• Byte offset	UBin(4)
• Entry index	UBin(4)
• Field length	UBin(4)
• Bit offset	UBin(2)
• Reason code	UBin(2)

Instructions Causing Exception:

2AB4 Program limitation exceeded.

An internal part of the program exceeds allowed size.

Contact your service representative to report the problem.

Information Passed:

• Reason code	UBin(2)

Instructions Causing Exception:

2AC0 Attempt to delete part that may not be deleted.

The program was created with observable data that may not be deleted. An attempt was made to delete that data.

Contact your service representative to report the problem.

Information Passed:

- Reason code UBin(4)

Instructions Causing Exception:

2C Program Execution

2C01 Return Instruction Invalid

Improper usage of the Return, Transfer Control, or Return From Exception instruction occurred for one of the following reasons:

- A Return From Exception instruction was executed in an invocation that was not defined as an exception handler.
- A Return External or Transfer Control instruction was issued from a first-invocation-level exception handler.
- A Transfer Control instruction was issued from a first-invocation-level event handler.

Instructions Causing Exception:

- Return External
- Return From Exception
- Transfer Control
- Signal Exception

2C02 Return Point Invalid

An attempt was made to use a Return External instruction with a return point that was invalid for one of the following reasons:

- The return point value was outside the range of the return list specified on the preceding Call External instruction.
- A nonzero return point was supplied, but no return list was supplied on the preceding Call External instruction.
- A nonzero return point was supplied when a Return External instruction was issued in the first invocation in the process.
- A nonzero return point was supplied when the Return External instruction was issued by an invocation acting as an event handler.

Instructions Causing Exception:

- Return External
- Signal Exception

<D>

2C04 Branch Target Invalid

An attempt was made to branch to an instruction defined through an instruction pointer, but the instruction pointer was set by a program other than the one that issued the branch.

Information Passed:

- Instruction pointer causing the exception

Instructions Causing Exception:

- All instructions that have a branch form
- Signal Exception

2C05 Activation in Use by Invocation

An attempt was made to de-activate a program that has an existing invocation which is not the invocation issuing the instruction.

Information Passed:

- Program — System pointer

Instructions Causing Exception:

- De-activate Program
- Signal Exception

2C08 Branch target defined by label pointer not valid

The program that was running attempted to branch to a label pointer which points to a label than is not in the current procedure.

Change the program so that it only branches to a label in the currently running procedure.

Information Passed:

- Branch target — Label pointer

Instructions Causing Exception:

- Signal Exception

2C10 Process object destroyed

The process object pointer refers to a process object which has been destroyed.

Information Passed:

None.

Instructions Causing Exception

- Any instruction which has process object pointer operands.

2C11 Process object access invalid

An attempt has been made to reference a process object from another process which is not the owner of that object.

Information Passed:

None.

Instructions Causing Exception

- Any instruction which has process object pointer operands may signal this exception.

2C12 Activation group access violation

An inter-activation group access is not permitted based on the activation group access protection mechanism.

Information Passed:

- Source activation group mark — UBin(4)
- Target activation group mark — UBin(4)

Instructions Causing Exception

- Materialize Activation Attributes
- Materialize Invocation Attributes
- Cancel Invocation
- other instructions which use the activation group access protection mechanism.

2C13 Activation group not found

The activation group specified by an activation group identifier could not be found in the process. The activation group was either destroyed or never existed.

Information Passed:

- activation group number UBin(4)

Instructions Causing Exception:

- Destroy Activation Group
- Materialize Activation Group Attributes
- Materialize Heap Space Attributes

2C14 Activation group in use

The specified activation group contains an invocation (i.e., it is in use) and cannot be destroyed.

Information Passed:

- activation group number UBin(4)

Instructions Causing Exception:

- Destroy Activation Group

2C15 Invalid operation for program

The operation requested is not supported for this model of program. The system supports two distinct "models" of program objects. Not all operations are supported for all models of programs. (E.g., a new model program object cannot be deactivated.)

Information Passed:

- Program System pointer

Instructions Causing Exception

- deactivate program
- call program
- create bound program
- ...

2C16 Program activation not found

The activation specified by an activation mark value does not exist. It was either destroyed or never created.

Information Passed:

- Activation mark Bin(4)

Instructions Causing Exception:

- Materialize Activation Attributes

2C17 Default activation group not destroyed

An attempt was made to destroy a default activation group using the Destroy Activation Group instruction. The operation is not permitted.

Information Passed:

- none

Instructions Causing Exception:

- Destroy Activation Group

2C18 Invalid source invocation

A source invocation was specified that is older than the target invocation. The operation is not permitted.

Information Passed:

• Source invocation offset	Bin(2)
• Operand number	Bin(2)
• Reserved (zeros)	Char(10)
• Base invocation used to determine source	Invocation pointer

Instructions Causing Exception:

- Send Process Message
- Materialize Process Message
- Modify Process Message
- Cancel Invocations

2C19 Invalid origin invocation

An origin invocation was specified that is older than the source or target invocation. The operation is not permitted.

Information Passed:

• Origin invocation offset	Bin(4)
• Operand number	Bin(2)
• Reserved (zeros)	Char(10)
• Base invocation used to determine origin	Invocation pointer

Instructions Causing Exception:

- Send Process Message
- Materialize Process Message
- Modify Process Message
- Cancel Invocations
- Materialize Invocation Attributes
- Modify Invocation Attributes

2C1A Invocation offset outside range of current stack

An offset was specified that attempts to identify an invocation that is either newer than the newest invocation in the process or older than the oldest invocation in the process. The operation is not permitted.

Information Passed:

• Length of invocation identification	Bin(2)
• Bit offset to invalid field	Bin(2)
• Operand number	Bin(2)
• Reserved (zeros)	Char(10)
• Invocation identification value	Char(*)

Instructions Causing Exception:

- Send Process Message
- Materialize Process Message
- Modify Process Message
- Cancel Invocations

- Materialize Invocation Attributes
- Modify Invocation Attributes
- Find Relative Invocation
- Sense Exception Description

2C1B *Invocation not eligible for operation*

An operation was attempted that is not valid for the specified invocation. The operation is not permitted.

Information Passed:

- none

Instructions Causing Exception:

- Modify Invocation Attributes

2C1C *Instruction not valid for invocation type*

An attempt was made to reference an invocation with an instruction that is not valid for the type of the invocation. The operation is not permitted.

Information Passed:

- none

Instructions Causing Exception:

- Materialize Invocation
- Materialize Instruction Attributes

2C1D *automatic storage overflow*

The automatic storage for the specified activation group has been exhausted. Further program execution within the activation group is not possible.

Information Passed:

- activation group mark UBin(4)

Instructions Causing Exception:

- program call instructions
- procedure call instructions
- Modify Automatic Storage Allocation

2C1E *activation access violation*

The program could not be activated within an existing activation group. The program specifies an activation group access protection level which is incompatible with the existing activation group.

Information Passed:

- Program System pointer
- Target activation group mark UBin(4)

Instructions Causing Exception:

- program call instructions

2C1F *program signature violation*

The source program specifies a signature which is not supported by the service program. The service program interface has changed and the source program must be rebound.

Information Passed:

- Source program — System pointer
- Service program — System pointer
- Signature — Char(16)

Instructions Causing Exception:

- program call instructions

2C20 static storage overflow

The static storage allocation for an activation group has been exceeded.

Information Passed:

- Activation group mark — UBin(4)

Instructions Causing Exception:

- program call instructions

2C21 program import invalid

The actual type of a service program export does not agree with the type expected by a client program.

Information Passed:

- Client program — System pointer
- Service program — System pointer
- Export-ID — UBin(4)
- Expected type — UBin(2)
- Actual type — UBin(2)

The type is encoded as:

0001 = procedure
0002 = data

Instructions Causing Exception:

- program call instructions

2C22 data reference invalid

The client program imports a data item from a program which is active in a different activation group. Data items cannot be refernced across activation group boundaries.

Information Passed:

- Client program — System pointer
- Service program — System pointer
- Export-ID — UBin(4)
- Client activation group mark — UBin(4)
- Service activation group mark — UBin(4)

Instructions Causing Exception:

- program call instructions

2C23 *imported object invalid*

The source program attempts to import items from an object which is not a service program.

Information Passed:

- Source program — System pointer
- Object — System pointer

Instructions Causing Exception:

- program call instructions

2C24 *activation group export conflict*

The activation of one or more programs has resulted in conflicting specifications for activation group exports.

Information Passed:

- Reason — UBin(4)
- Activation group mark — UBin(4)
- Name (first 64 characters) — Char(64)

The **Reason** field supplies more information as follows:

1 = multiple strong exports of the specified item were encountered.

2 = the specified item was already allocated within the activation group — a subsequent strong export was encountered for the same item.

3 = the specified item was already allocated within the activation group — a subsequent weak export with a conflciting length specification was encountered.

Instructions Causing Exception:

- program call instructions

2C25 *import not found*

The client program specifies a procedure or data import which is not found in the service program. The *export-ID* specified in the client program exceeds the *export count* of the service program. The client program must be rebound.

Information Passed:

- Client program — System pointer
- Service program — System pointer
- Export-ID — UBin(4)
- Export count — UBin(4)

Instructions Causing Exception:

- program call instructions

2E Resource Control Limit

2E01 User Profile Storage Limit Exceeded

The user profile specified insufficient auxiliary storage to create or extend a permanent object.

Information Passed:

- User profile — System pointer

Instructions Causing Exception:

- All create instructions creating a permanent object
- All instructions extending a permanent object

  Any instruction which references bytes of data within a space object can cause an automatic extension of the space if the space has the attribute of being automatically extendible. Therefore, this exception may be signaled for any instruction which has an operand which references bytes of data in a space.

- Signal Exception

2E02 Security audit journal failure

An entry could not be sent to the security audit journal.

Information Passed:

- Audit journal port — System pointer
- Object being journaled (null for program adopt audits) — System pointer
- Return code — Char(2)

hex 0000 - No error detected
hex 0001 - No journal space attached
hex 0002 - Extend failure
hex 0003 - Damaged journal port
hex 0004 - Damaged journal space
hex 0005 - Maximum sequence number
hex 0006 - Journal failure
hex 0008 - Journal space off line
hex 0009 - Journal port off line
hex 0100 - This object type is not to be audited
hex 0200 - Invalid data length
hex 0300 - Object type table not defined
hex FF00 - Failure in VLIC auditing module

Instructions Causing Exception:

- All instructions performing underlying operations that cause an audit record to be sent (program adopt, object change or reference, etc.).

32 Scalar Specification

3201 Scalar Type Invalid

A scalar operand did not have the following data types required by the instruction:

- Character
- Packed decimal
- Zoned decimal
- Binary
- Floating-point

Instructions Causing Exception:

- Any instruction using a late bound (data pointer) scalar operand
- Signal Exception

3202 Scalar Attributes Invalid

A scalar operand did not have the following attributes required by the instruction:

- Length
- Precision
- Boundary

Instructions Causing Exception:

- Any instruction using a late-bound (data pointer) scalar operand
- Any instruction that verifies the length of a character scalar in a space object operand
- Signal Exception

3203 Scalar Value Invalid

A scalar operand does not contain a correct value as required by the instruction.

Information Passed:

• Length of data passed	Bin(2)
• Bit offset to invalid field (relative to 0)	Bin(2)
• Operand number	Bin(2)
• Invalid data	Char(*)

Instructions Causing Exception:

- Any instruction using a scalar operand
- Signal Exception

36 Space Management

3601 Space Extension/Truncation

A Modify Space Attributes instruction made one of the following invalid attempts to modify the size of the space:

- Truncate the space to a negative size.
- Extend or truncate a fixed size space.
- Extend a space beyond the space allowed in the referenced object.
- An operation which required an automatic extension of a space occurred when the extended space would not fit in the access group which contained it.
- Extend or truncate a space that has a hardware storage protection level of 01 or greater while running in user state.

For information on the maximum size space allowed for a particular object, refer to the *Limitations* topic within the definition of the create instruction for that type of object.

Information Passed

- Space System pointer

Instructions Causing Exception

- Activate Program
- Call External
- Modify Automatic Storage Allocation
- Modify Space Attributes
- Signal Exception
- Transfer Control
- Any instruction that invokes an external exception handler or an external event handler or an invocation exit
- Any instruction which has an operand which references bytes of data in a space.

  Any instruction which references bytes of data within a space object can cause an automatic extension of the space if the space has the attribute of being automatically extendible.

3602 Invalid Space Modification

A Modify Space Attributes instruction made an attempt to modify the attributes of a space but the requested modification is invalid.

Information Passed:

- System pointer to the object
- Error code Char(2)

  Error codes and their meanings are as follows:

Code	Meaning
0001	An attempt was made to modify the performance class attribute of the system object containing the space and the space was not a fixed length of size zero.
0002	An attempt was made to modify a system object to or from the state of having a fixed length space of size zero and the operation is invalid for that type of system object.

0003	An attempt was made to modify a system object to the state of having a fixed length space and the automatic extend attribute. These are mutually exclusive.
0004	An attempt was made to modify the associated space of a program while running in user state on a system running at security level 40 or above. This combination is invalid.

Instructions Causing Exception:

- Modify Space Attributes
- Signal Exception

38 Template Specification

3801 Template Value Invalid

A template did not contain a correct value required by the instruction.

Information Passed:

- Addressability to the template — Space pointer
- Offset to invalid field in bytes — Bin(2)

  A value of 0 is the first byte in the template. An invalid field is considered to be the lowest-level character or numeric template entry that contains the information that is in error.

- Bit offset in invalid field or 0 — Bin(2)

  A 0 value indicates the leftmost bit in the invalid field.

- The number of bytes in the invalid field — Bin(2)
- Instruction operand number — Bin(2)
  (The first operand in an instruction is 1.)
- Reason code — Char(2)

  The meaning of the reason code which may or may not be returned is specific to the definition of each instruction. Refer to the instruction definition to find out whether or not a meaningful value is returned. Unless otherwise stated, this field has no meaning.

  The following instructions support setting the reason code.

  - Create Dictionary
  - Create Module
  - Create Bound Program

Instructions Causing Exception:

- Any instruction that has a space pointer as a source operand
- Convert BSC to Character
- Convert Character to BSC
- Convert Character to MRJE
- Convert MRJE to Character
- Signal Exception
- Scan with Control

3802 Template Size Invalid

A source template was not large enough for this instruction.

Information Passed:

- Addressability to the template — Space pointer

Instructions Causing Exception:

- Any instruction that has a space pointer that addresses a source template operand
- Signal Exception

3803 Materialization Length Exception

Less than 8 bytes was specified to be available in the receiver operand of a materialize instruction.

Instructions Causing Exception:

- Any materialize instruction
- Any retrieve instruction
- Signal Exception

3A Wait Time-Out

3A01 Dequeue

A specified time period elapsed, and a Dequeue instruction was not satisfied.

Information Passed:

• The queue waited for	System pointer
• Time-out value	Char(8)

Instructions Causing Exception:

- Dequeue
- Signal Exception

3A02 Lock

A specified time period elapsed, and a Lock Object instruction was not satisfied.

Information Passed:

• System pointer to the object waited for	
• Time-out value	Char(8)

Instructions Causing Exception:

- Lock Object
- Signal Exception

3A04 Space Location Lock Wait

A specified time period has elapsed and a Lock Space Location instruction has not been satisfied.

Information Passed:

• Space location	Space pointer
• Time-out value	Char(8)

Instructions Causing Exception:

- Lock Space Location
- Signal Exception

3C Service

3C01 Invalid Service Session State

The process is not in the proper service session for the request service command because of one of the following conditions:

- No service session exits for the process, and the command is other than start service session.
- The process is in service session, and the command is to start service session.
- The process is in service session, but a previous stop service session command was issued.

Instructions Causing Exception:

- Signal Exception

3C02 Unable to Start Service Session

The machine was unable to start a valid service session.

Instructions Causing Exception:

- Signal Exception

3E Commitment Control

3E01 Invalid Commit Block Status Change

An attempt was made to modify (to an invalid status) the status of a commit block attached to the issuing process. The exception data defines the attempted change to the commit block status.

Information Passed:

- Commit block — System pointer
- Attempted status change — Char(2)
  (as defined in the modifications options in the modification template for the Modify Commit Block instruction)

Instructions Causing Exception:

- Modify Commit Block
- Signal Exception

3E03 Commit Block Is Attached to Process

The identified commit block is attached to a process making the operation requested impossible. The process that has the commit block attached is identified in the exception data.

Information Passed:

- Commit block — System pointer
- Process control space — System pointer

Instructions Causing Exception:

- Destroy Commit Block
- Signal Exception

3E04 Commit Blocks Control Uncommitted Changes

The identified commit block controls uncommitted changes, and an attempt was made to detach the commit block from the issuing process.

Information Passed:

- Commit block — System pointer

Instructions Causing Exception:

- Modify Commit Block
- Signal Exception

3E05 Operation Not Valid on Commit Block in Prepared State

The identified Commit Block is in a prepared state. The operation attempted is therefore not allowed. Signalled from .#COCHECK module.

Information Passed:

- Commit block — System pointer

Instructions Causing Exception:

- Modify Commit Block
- Activate Cursor
- Deactivate Cursor
- Delete Data Space Entry

- Destroy Commit Block
- Destroy Cursor
- Insert Data Space Entry
- Insert Sequential Data Space Entries
- Release Data Space Entries
- Retrieve Data Space Entry
- Update Data Space Entry

3E06 Commitment Control Resource Limit Exceeded

One of the resource limits for the commitment control functions has been reached.

Information Passed:

- Commit block — System pointer
- Condition code — Char(1)

  Hex 01 = Lock limit exceeded
  Hex 02 = Object list size exceeded
  Hex 03 = Limit of attached commit blocks in the system exceeded
  Hex 04 = Reposition cursor data limit exceeded

Instructions Causing Exception:

- Insert Data Space Entry
- Insert Sequential Data Space Entries
- Modify Commit Block
- Set Cursor
- Signal Exception

3E08 Object Under Commitment Control Being Journaled Incorrectly

All objects under commitment control must have their changes journaled through the same journal port as the commit block. An attempt was made to place an object under commitment control that did not meet this condition.

Information Passed:

- Object — System pointer
- Journal port that must be used — System pointer
- Journal port currently being used (binary 0 if not currently being journaled) — System pointer

Instructions Causing Exception:

- Modify Commit Block
- Signal Exception

3E10 Operation Not Valid Under Commitment Control

An operation was attempted on an object or through an object that was currently under commitment control. The operation is not supported under commitment control.

Information Passed:

- Object under commitment control — System pointer

Instructions Causing Exception:

- Copy Data Space Entries

- De-activate Cursor
- Destroy Cursor
- Retrieve Sequential Data Space Entries
- Signal Exception

3E11 Process Has Attached Commit Block

An attempt was made to attach a second commit block to a process.

Information Passed:

• Commit block (attached)	System pointer
• Commit block (attempted to attach)	System pointer

Instructions Causing Exception:

- Modify Commit Block
- Signal Exception

3E12 Objects Under Commitment Control

An attempt was made to detach a commit block from a process that has objects under commitment control.

Information Passed:

• Commit block	System pointer

Instructions Causing Exception:

- Modify Commit Block
- Signal Exception

3E13 Commit Block Not Journaled

An attempt was made to attach a commit block to a process, and the commit block was not being journaled.

Information Passed:

• Commit block	System pointer

Instructions Causing Exception:

- Modify Commit Block
- Signal Exception

3E14 Errors During Decommit

Errors were detected during an execution of a Decommit instruction. The exception data indicates the type of errors detected.

Information Passed:

• Commit block	System pointer
• Reserved (binary 0)	Char(1)
• Decommit status	Char(4)
— Damaged	Bit 0
0 = Commit block is not damaged 1 = Commit block is damaged	
— Reserved (binary 0)	Bits 1-2
— Partially damaged	Bit 3

0 = Not partially damaged
1 = Partially damaged

- Reserved (binary 0) — Bits 4-15
- Decommit — Bit 16

  0 = All changes were decommitted
  1 = Not all changes were decommitted

- Journal read errors — Bit 17

  0 = No journal read errors
  1 = Journal read errors occurred during decommit

- Journal write errors — Bit 18

  0 = No journal write errors
  1 = Journal write errors occurred during decommit

- Partial damage to data space — Bit 19

  0 = No damage encountered
  1 = Damage encountered on one or more data spaces

- Damage to data space — Bit 20

  0 = No damage encountered
  1 = Damage encountered on one or more data spaces

- Function check — Bit 21

  0 = No function check encountered
  1 = Function check encountered

- Reserved (binary 0) — Bits 22-23
- Constant = 100 — Bits 24-26
- Reserved (binary 0) — Bits 27-31

• Reserved (binary 0) — Char(7)

• Journal entry sequence number of start commit journal entry — Bin(4)

Instructions Causing Exception:

• Decommit

• Signal Exception

3E15 *Object Ineligible for Commitment Control*

The specified object is not eligible to be placed under commitment control.

Information Passed:

• Object — System pointer

• Reason code — Char(1)

Hex 01 = Object is a type that is not supported under commitment control
Hex 02 = Object is a cursor that was not activated under the issuing process
Hex 03 = Object is already under commitment control to this another commit block or to commit block
Hex 04 = Object is a cursor that holds data space entry locks
Hex 05 = Object is a join cursor

Instructions Causing Exception:

- Modify Commit Block
- Signal Exception

3E16 Object Ineligible for Removal from Commitment Control

The specified object cannot be removed from commitment control.

Information Passed:

• Object	System pointer
• Reason code	Char(1)

Hex 01 = Object is a type that is not supported under commitment control
Hex 02 = Object is not under commitment control of this commit block
Hex 03 = Object is a cursor holding data space entry locks

Instructions Causing Exception:

- Modify Commit Block
- Signal Exception

44 Domain Violation Exceptions

4401 Object Domain Violation

An attempt was made to use a blocked instruction or to access a system domain object from a user state program. The system must be running at security level 40 to get this exception.

The execution of the instruction is suppressed when the exception is signalled.

Information Passed:

- Object. System pointer

Instructions Causing Exception:

- All blocked instructions.
- Any instruction that contains an operand that can be a pointer or be represented in storage (i.e. the value for the operand is to retrieved from storage).

Appendix A. Instruction Summary

This appendix provides an abbreviated format of all the instructions. The instructions are listed alphabetically by instruction mnemonic.

The summary list includes the following items for each instruction.

- *Operation Description*-The name of the instruction.
- *Mnemonic*-The mnemonic assigned to the instruction.
- *Operation Code*-The operation code assigned to the instruction.
- *Number of Operands*-The number of operands (excluding the extender) in the instruction.
- *Extender*-A description of the use of the extender field.
- *Operand Syntax*-The objects allowed as operands in the instruction.
- *Resulting Conditions*-The conditions that can be set at the end of the standard operation in order to perform a conditional branch or set a conditional indicator.
- *Optional Forms*-A notation for the optional forms that are allowed for the computational instructions.

Note: This summary list can also be used as an index to identify the page where a complete description of each instruction can be found in this manual. The page number is the last item included with each instruction in this summary.

The following paragraphs further describe the summary list format of the last five items in the previous list.

Number Of Operands

Certain computational instructions allow a variable number of operands and are identified in the summary list by the following form:

number + B

The number defines the number of fixed operands. The B indicates the existence of variable operands (branch targets or indicator operands). A pair of braces around the letter indicates that the variable operands are optional.

Extender Usage

Instructions that use an extender field have a brief description of the use of the extender. Hyphens indicate that the extender is not used. Brackets indicate that the extender is optional. The abbreviation BR/IND is used to mean branch or indicator options. The extender field defines the use of the branch or indicator operands with respect to the resulting conditions of the instruction.

Resulting Conditions

Resulting conditions are the status result of the operation that is used for determining a branch target, if any.

The following conditions are indicated in the instruction summary.

P, N, Z	Positive, negative, zero
Z, NZ	Zero, not zero

© Copyright IBM Corp. 1991, 1993

H, L, E	High. low, equal
E, NE	Equal, not equal
P, Z	Positive, zero
H, L, E, U	High, low, equal, unequal
Z, O, M	Zero, ones, mixed
[N]Z[N]C	Zero and no carry, not zero and no carry, zero and carry, not zero and carry
S, NS	Signaled, not signaled
DE, I	Exception deferred, exception ignored
DQ, NDQ	Dequeued, not dequeued

Optional Forms

All instructions are classified as computational or noncomputational format. The format determines how the operation code is interpreted and whether optional forms of the instruction are allowed. (See "Instruction Format" in Chapter 1. "Introduction").

Certain computational instructions allow optional forms. The following optional forms can be specified:

- *B (Branch Form)*-The resulting conditions of the operation are compared with the branch options specified in the extender field. If one of the options is satisfied, a branch is executed to the branch target corresponding to the branch option.
- *I (Indicator Form)*-The resulting conditions of the operation are compared with the indicator options specified in the extender field. If one of the options is satisfied, the indicator corresponding to that option is assigned a value of hex F1. The other indicators referred to by the operation are assigned a value of hex F0.
- *S (Short Form)*-The operand that acts as a receiver in the instruction can also be one of the source operands.
- *R (Round Form)*-If the result of the operation is to be truncated before being placed in the receiver, rounding is performed.

Instruction Stream Syntax

In this instruction summary, the following metalanguage is used to describe the machine interface instruction set operand syntax.

Metasymbol	Meaning
{}	Choose from a series of alternatives
[ ]	Enclose an optional entry or entries
|	OR - used to separate alternatives
.N.	Repeat previous entry, up to N times
::=	Is defined as - define a metavariable Metavariable ::= Metadefinition
DESC-{}	Description of a metavariable in English

Notes:

1. Some of the computational op codes require an extender field while on other op codes an extender field is optional. Some computational op codes may be optionally short, or round.

Program Object Definitions

ARG-LIST ::= DESC-{operand list which defines an argument list}

B-ARRAY ::= DESC-{array of binary variables} B-PT ::= DESC-{branch point} BIN ::= DESC-{binary} BIN[N] ::= DESC-{binary object with precision N} BT ::= DESC-{instruction number | relative instruction number | instruction pointer | branch pointer | IDL element | null}

C-ARRAY ::= DESC-{array of character string variables} CHAR ::= DESC-{character string which is either variable or constant} CHAR[N] ::= DESC-{string at least N bytes long} CHARV ::= DESC-{char variable} CHARC ::= DESC-{char constant}

D-PTR ::= DESC-{data pointer}

EXCP-DESC ::= DESC-{exception description}

F-BT ::= DESC-{instruction number | relative instruction number | branch point} F-P ::= DESC-{floating-point value}

IDL ::= DESC-{instruction definition list} IT ::= DESC-{char|numeric variable used as an indicator target} I-ENT PT ::= DESC-{internal entry point} I-PTR ::= DESC-{instruction pointer}

NULL ::= DESC-{indicates a null operand [X'0000']} NUMERIC ::= DESC-{binary | zoned | packed | numeric scalar} N-ARRAY ::= DESC-{array of numeric variable}

OP-LIST ::= DESC-{operand list}

PROCESS ::= DESC-{character string that names a process} PTR ::= DESC-{a 16-byte, 16-byte-boundary-aligned pointer element} P-ARRAY ::= DESC-{an array of 16 bytes, 16-byte-boundary-aligned pointer(s)}

SPDO ::= DESC-{space pointer data object} S-PTR ::= DESC-{system pointer} SPP ::= DESC-{space pointer} SPP-ARRAY ::= DESC-{an array of space pointer variables}

Notes:

1. NUMERIC, CHAR, BIN, and UBIN may be followed by the special characters S, C, V. CHAR, BIN, and UBIN may also be followed by the special character I. These characters further qualify the object as being scalar, constant, variable or immediate, respectively.
2. All array objects are variable.

System Object Declarations

ACTV ENTRY ::= DESC-{SPP that addresses an activation}

AG ::= DESC-{S-PTR that addresses an access group}

AL ::= DESC-{S-PTR that addresses an authorization list}

CD ::= DESC-{S-PTR that addresses a controller description}

CSD ::= DESC-{S-PTR that addresses a class of service description}

CONTEXT ::= DESC-{S-PTR that addresses a context}

CURSOR ::= DESC-{S-PTR that addresses a cursor}

DATA SPACE ::= DESC-{S-PTR that addresses a data space}

DCT ::= DESC-{S-PTR that addresses a dictionary}

DS-INDEX ::= {S-PTR that addresses a data space index}

INDEX ::= DESC-{S-PTR that addresses an index}

LUD ::= DESC-{S-PTR that addresses a logical unit description}

MD ::= DESC-{S-PTR that addresses a mode description}

MODULE ::= DESC-{S-PTR that addresses a module}

ND ::= DESC-{S-PTR that addresses a network description}

PCS ::= DESC-{S-PTR to process control space}

PROGRAM ::= DESC-{S-PTR that addresses a program}

SPACE ::= DESC-{a system pointer pointing to a space object}

QUEUE ::= DESC-{S-PTR that addresses a queue}

QUEUE SPACE ::= DESC-{S-PTR that addresses a queue space}

USER PROFILE ::= DESC-{S-PTR that addresses a user profile}

Resulting Conditions Definitions

ZC ::= DESC-{zero with carry}

[N]ZC ::= DESC-{[not] zero with carry}

Z[N]C ::= DESC-{zero with [no] carry}

[N]Z[N]C ::= DESC-{[not] zero with [no] carry}

CR ::= DESC-{completed record}

DE ::= DESC-{deferred}

DEN ::= DESC-{denormalized}

[N]DQ ::= DESC-{[Not]dequeued}

ECE ::= DESC-{escape code encountered}

E ::= DESC-{equal}

[N]F ::= DESC-{[Not]found}

H ::= DESC-{high}

I ::= DESC-{ignored}

IN ::= DESC-{infinity}

L ::= DESC-{low}

M ::= DESC-{mixed}

N ::= DESC-{negative}

NaN ::= DESC-{symbolic not-a-number}

NCO ::= DESC-{null compare operand}

NE ::= DESC-{not equal}

NRN ::= DESC-{normalized real number}

NS ::= DESC-{not signaled}

NZ ::= DESC-{not zero}

O ::= DESC-{ones}

P ::= DESC-{positive}

RO ::= DESC-{receiver overrun}

S ::= DESC-{signaled}

SE ::= DESC-{source exhausted}

TR ::= DESC-{truncated record}

U ::= DESC-{unequal}

UN ::= DESC-{unordered}

Z ::= DESC-{zero}

Instruction Summary (Alphabetical Listing by Mnemonic)

Operation Description	Mnemonic	Op Code	Number of Operands	Operand Syntax	Resulting Condi-tions	Optional Forms	Page
Activate Program	ACTPG	0212	2	{ACTV ENTRY | PROGRAM},PROGRAM	-	-	9-3
Add Logical Character	ADDLC	1023	3+[B]	CHARV, CHARS.2., [BT.4. | IT.4.]	[N]Z[N]C	[B | I, S]	2-3
Add Numeric	ADDN	1043	3+[B]	NUMERICV, NUMERICS.2., [BT.4. | IT.4.]	P, N, Z, UN	[B | I, S, R]	2-6
Add Space Pointer	ADDSPP	0083	3	SPP.2., BINS	-	-	5-3
Allocate Heap Space Storage	ALCHSS	03B3	3	SPP, {BIN | NULL}, BIN			7-3
And	AND	1093	3+[B]	CHARV, CHARS.2., [BT.3. | IT.3.]	Z, NZ	[B | I, S]	2-10
Branch	B	1011	1	BT	-	-	2-13
Compute Array Index	CAI	1044	4	BINV, BINS.3.	-	-	2-30
Call Internal	CALLI	0293	3	I-ENT PT, {ARG LIST | NULL}, I-PTR	-	-	9-9
Call External	CALLX	0283	3	PROGRAM | SPP, {ARG LIST | NULL}, {IDL | NULL}	-	-	9-5
Concatenate	CAT	10F3	3	CHARV, CHARS.2.	-	-	2-46
Compute Date Duration	CDD	0424	4	NUMERICV, CHAR, CHAR, SPP	-	-	3-13
Clear Bit in String	CLRBTS	102E	2	{CHARV | NUMERICV}, BINS	-	-	2-15
Clear Invocation Exit	CLRIEXIT	0250	0	-	-	-	9-11
Compute Math Function Using One Input Value	CMF1	100B	3+[B]	NUMERICV, CHARS[2], NUMERICS, [BT.4. | IT.4.]	P, N, Z, UN	[B | I]	2-32
Compute Math Function Using Two Input Values	CMF2	100C	4+[B]	NUMERICV, CHARS[2], NUMERICS, NUMERICS, [BT.4. | IT.4.]	P, N, Z, UN	[B | I]	2-41
Compare Bytes Left-Adjusted	CMPBLA	10C2	2+B	{CHARS | NUMERICS}.2., {BT.3. | IT.3.}	H, L, E	{B | I}	2-17

Operation Description	Mnemonic	Op Code	Number of Operands	Operand Syntax	Resulting Condi- tions	Optional Forms	Page
Compare Bytes Left-Adjusted With Pad	CMPBLAP	10C3	3+B	{CHARS | NUMERICS}.3., {BT.3. | IT.3.}	H, L, E	{B | I}	2-19
Compare Bytes Right-Adjusted	CMPBRA	10C6	2+B	{CHARS | NUMERICS}.2., {BT.3. | IT.3.}	H, L, E	{B | I}	2-21
Compare Bytes Right-Adjusted With Pad	CMPBRAP	10C7	3+B	{CHARS | NUMERICS}.3., {BT.3. | IT.3.}	H, L, E	{B | I}	2-23
Compare Numeric Value	CMPNV	1046	2+B	NUMERICS.2., {BT.4. | IT.4.}	H, L, E, UN	{B | I}	2-25
Compare Pointer for Space Address-ability	CMPPSPAD	10E6	2+B	{SPP | D-PTR}, {NUMERICV | CHARV | C-N-ARRAY | SPP | D-PTR}, {BT.4. | IT.4.}	H, L, E, U	{B | I}	4-5
Compare Pointer for Object Address-ability	CMPPTRA	10D2	2+B	{D-PTR | SPP | S-PTR | I-PTR}.2., {BT.2. | IT.2.}	E, NE	[B | I]	4-3
Compare Pointers for Equality	CMPPTRE	1012	3+[B]	{ Any PTR }.2.	E, NE	[B | I]	4-7
Compare Pointer Type	CMPPTRT	10E2	2+B	{D-PTR | SPP | S-PTR | I-PTR}, {CHARS[1]NULL}, {BT.2. | IT.2.}	E, NE	{B | I}	4-9
Compare Space Addressability	CMPSPAD	10F2	2+B	{CHARV | C-ARRAY | NUMERICV | N-ARRAY | PTR | P-ARRAY.2.}, {BT.4. | IT.4.}	H, L, E, U	{B | I}	5-5
Compress Data	CPRDATA	1041	1	Space Pointer	-	-	2-28
Copy Bytes to Bits Arithmetic	CPYBBTA	104C	4	{NUMERICV | CHARV}, BINI.2., {NUMERICV | CHARV}	-	-	2-124
Copy Bytes to Bits Logical	CPYBBTL	103C	4	{NUMERICV | CHARV}, BINI.2., {NUMERICV | CHARV}	-	-	2-126

Operation Description	Mnemonic	Op Code	Number of Operands	Operand Syntax	Resulting Condi- tions	Optional Forms	Page
Copy Bytes Left-Adjusted	CPYBLA	10B2	2	{NUMERICV | CHARV}, {NUMERICS | CHARS}	-	-	2-110
Copy Bytes Left-Adjusted With Pad	CPYBLAP	10B3	3	{NUMERICV | CHARV}, {NUMERICS | CHARS}.2.	-	-	2-112
Copy Bytes Overlap Left-Adjusted	CPYBOLA	10BA	2	{NUMERICV | CHARV}.2.	-	-	2-114
Copy Bytes Overlap Left-Adjusted With Pad	CPYBOLAP	10BB	3	{NUMERICV | CHARV}.2., {NUMERICS | CHARS}	-	-	2-116
Copy Bytes Right-Adjusted	CPYBRA	10B6	2	{NUMERICV | CHARV}, {NUMERICS | CHARS}	-	-	2-120
Copy Bytes Right-Adjusted With Pad	CPYBRAP	10B7	3	{NUMERICV | CHARV}, {NUMERICS | CHARS}.2.	-	-	2-122
Copy Bytes Repeatedly	CPYBREP	10BE	2	{NUMERICV | CHARV}, {NUMERICS | CHARS}	-	-	2-118
Copy Bits Arith-metic	CPYBTA	102C	4	{NUMERICV | CHARV}.2., BINI.2.	-	-	2-100
Copy Bits Logical	CPYBTL	101C	4	{NUMERICV | CHARV}.2., BINI.2.	-	-	2-102
Copy Bits With Left Logical Shift	CPYBTLLS	102F	3	{CHARV | NUMERICV}, {CHARS | NUMERICS}, CHARS[2]	-	-	2-104
Copy Bits With Right Arithmetic Shift	CPYBTRAS	101B	3	{CHARV | NUMERICV}, {CHARS | NUMERICS}, CHARS[2]	-	-	2-106
Copy Bits With Right Logical Shift	CPYBTRLS	103F	3	{CHARV | NUMERICV}, {CHARS | NUMERICS}, CHARS[2]	-	-	2-108
Copy Bytes With Pointers	CPYBWP	0132	2	{CHARV | PTR}, {CHARV | PTR | NULL}	-	-	4-12

Operation Description	Mnemonic	Op Code	Number of Operands	Operand Syntax	Resulting Condi-tions	Optional Forms	Page
Copy Extended Characters Left-Adjusted With Pad	CPYECLAP	1053	3	D-PTR CHARS,D-PTR CHARS,CHAR	-	-	2-128
Copy Hex Digit Numeric to Numeric	CPYHEXNN	1092	2	{NUMERICV | CHARV}, {NUMERICS | CHARS}	-	-	2-132
Copy Hex Digit Numeric to Zone	CPYHEXNZ	1096	2	{NUMERICV | CHARV}, {NUMERICS | CHARS}	-	-	2-134
Copy Hex Digit Zone to Numeric	CPYHEXZN	109A	2	{NUMERICV | CHARV}, {NUMERICS | CHARS}	-	-	2-136
Copy Hex Digit Zone to Zone	CPYHEXZZ	109E	2	{NUMERICV | CHARV}, {NUMERICS | CHARS}	-	-	2-138
Copy Numeric Value	CPYNV	1042	2+[B]	NUMERICV, NUMERICS, [BT.4. | IT.4.]	P, N, Z, UN	[B | I, R]	2-140
Create Heap Space	CRTHS	03B2	2	BINV, SPP			7-6
Compute Time Duration	CTD	0454	4	NUMERICV, CHAR, CHAR, SPP	-	-	3-16
Compute Timestamp Duration	CTSD	043C	4	NUMERICV, CHAR, CHAR, SPP	-	-	3-19
Convert BSC to Character	CVTBC	10AF	3 + [B]	CHARV, CHARV[3], CHARS, {BT.3. | IT.3.}	CR, SE, TR	[B | I]	2-48
Convert Char-acter to BSC	CVTCB	108F	3 + [B]	CHARV, CHARV[3], CHARS, {BT.2. | IT.2.}	SE, RO	[B | I]	2-52
Convert Char-acter to Hex	CVTCH	1082	2	CHARV, CHARS	-	-	2-55
Convert Char-acter to MRJE	CVTCM	108B	3 + [B]	CHARV, CHARV[13], CHARS, {BT.2. | IT.2.}	SE, RO	[B | I]	2-57
Convert Char-acter to Numeric	CVTCN	1083	3	NUMERICV, CHARS, CHARS[7]	-	-	2-62
Convert Char-acter to SNA	CVTCS	10CB	3 + [B]	CHARV, CHARV[15], CHARS, {BT.2. | IT.2.}	SE, RO	[B | I]	2-65

Operation Description	Mnemonic	Op Code	Number of Operands	Operand Syntax	Resulting Condi- tions	Optional Forms	Page
Convert Decimal Form to Floating-Point	CVTDFFP	107F	3	F-PS, NUMERICS, NUMERICS	-	-	2-74
Convert External Form to Numeric Value	CVTEFN	1087	3	NUMERICV, CHARS, {CHARS[3] | NULL}	-	-	2-76
Convert Floating-Point to Decimal Form	CVTFPDF	10BF	3	NUMERICV, NUMERICV, F-PS	-	Round	2-79
Convert Hex to Character	CVTHC	1086	2	CHARV, CHARS	-	-	2-82
Convert MRJE to Character	CVTMC	10AB	3 + [B]	CHARV, CHARV[6], CHARS, {BT.2. | IT.2.}	SE, RO	[B | I]	2-84
Convert Numeric to Character	CVTNC	10A3	3	CHARV, NUMERICS, CHARS[7]	-	-	2-88
Convert SNA to Character	CVTSC	10DB	3 + [B]	CHARV, CHARV[14], CHARS, {BT.3. | IT.3.}	SE, RO, ECE	[B | I]	2-90
Convert Date	CVTD	040F	3	CHAR, CHAR, SPP	-	-	3-22
Convert Time	CVTT	041F	3	CHAR, CHAR, SPP	-	-	3-25
Convert Timestamp	CVTTS	043F	3	CHAR, CHAR, SPP	-	-	3-28
Decompress Data	DCPDATA	1051	1	SPP	-	-	2-143
De-activate Program	DEACTPG	0225	1	PROGRAM | NULL	-	-	9-12
Decrement Date	DECD	0414	4	CHAR, CHAR, NUMERICV, SPP	-	-	3-31
Decrement time	DECT	0444	4	CHAR, CHAR, NUMERICV, SPP	-	-	3-35
Decrement timestamp	DECTS	040C	4	CHAR, CHAR, NUMERICV, SPP	-	-	3-38
Dequeue	DEQ	1033	3+[B]	CHARV, SPP, QUEUE, [BT.2. | IT.2.]	DQ, NDQ	[B | I]	12-3
Destroy Heap Space	DESHS	03B1	1	BINV			7-11

Operation Description	Mnemonic	Op Code	Number of Operands	Operand Syntax	Resulting Conditions	Optional Forms	Page
Divide	DIV	104F	3+[B]	NUMERICV, NUMERICS.2., [BT.4. | IT.4.]	P, N, Z, UN	[B | I, S, R]	2-146
Divide With Remainders	DIVREM	1074	4+[B]	NUMERICV, NUMERICS.2., NUMERICV	P, N, Z	[B | I, S, R]	2-150
Extended Character Scan	ECSCAN	10D4	4+[B]	B-ARRAY, CHARS, CHARS, CHARS[1], {BT.3. | IT.3.}	P, Z, ECE	[B | I]	2-167
Edit	EDIT	10E3	3	CHARV, NUMERICS, CHARS	-	-	2-154
End	END	0260	0	-	-	-	9-14
Enqueue	ENQ	036B	3	QUEUE, CHARS, SPP	-	-	12-9
Ensure Object	ENSOBJ	0381	1	S-PTR	-	-	19-3
Exchange Bytes	EXCHBY	10CE	2	{CHARV | NUMERICV}.2.	-	-	2-162
Extract Exponent	EXTREXP	1072	2+[B]	BINV, F-PS, {BT.4. | IT.4.}	NRN, DEN, IN, NaN	[B | I]	2-171
Extract Magnitude	EXTRMAG	1052	2+[B]	NUMERICV, NUMERICS, [BT.3. | IT.3.]	P, Z, UN	[B | I, S]	2-174
Find Independent Index Entry	FNDINXEN	0494	4	SPP, INDEX, SPP.2.	-	-	11-12
| Find Relative Invocation Number	FNDRINVN	0543	3	BIN, {CHARS[48] | NULL}, SPP	-	-	21-3
| Free Heap Space Storage	FREHSS	03B5	1	SPP	-	-	7-13
| Free Heap Space Storage from Mark	FREHSSMK	03B9	1	SPP	-	-	7-15
Insert Independent Index Entry	INSINXEN	04A3	3	INDEX, SPP.2.	-	-	11-16
Lock Object	LOCK	03F5	1	SPP	-	-	13-3
Lock Space Location	LOCKSL	03F6	2	SPP, CHARS[1]	-	-	13-8
Materialize Access Group Attributes	MATAGAT	03A2	2	SPP, AG	-	-	19-5
| Materialize Activation Group Attributes	MATAGPAT	02D3	3	SPP, UBIN, CHARS[1]	-	-	9-19
Materialize Authority List	MATAL	01B3	3	SPP, AL, SPP	-	-	17-7

Operation Description	Mnemonic	Op Code	Number of Operands	Operand Syntax	Resulting Condi-tions	Optional Forms	Page
Materialize Allo-cated Object Locks	MATAOL	03FA	2	SPP, {S-PTR | SPDO}	-	-	-- Heading 'MATAOL' unknown --
Materialize Authority	MATAU	0153	3	SPP, S-PTR, {USER PROFILE | NULL}	-	-	17-3
Materialize Authorized Objects	MATAUOBJ	013B	3	SPP, USER PROFILE, CHARS[1]	-	-	17-12
Materialize Authorized Users	MATAUU	0143	3	SPP, S-PTR, CHARS[1]	-	-	17-20
Materialize Bound Program	MATBPGM	02C6	2	SPP, S-PTR	-	-	8-3
Materialize Context	MATCTX	0133	3	SPP, {CONTEXT | NULL}, CHARS	-	-	16-3
Materialize Dump Space	MATDMPS	04DA	2	SPP, S-PTR	-	-	20-3
Materialize Data Space Record Locks	MATDRECL	032E	2	SPP, SPP	-	-	13-13
Materialize Exception Description	MATEXCPD	03D7	3	SPP, EXCP-DESC, CHARS[1]	-	-	14-3
Materialize Instruction Attri-butes	MATINAT	0526	2	SPP, CHARS	-	-	21-8
Materialize Invocation	MATINV	0516	2	SPP.2.	-	-	21-14
Materialize Invocation Attri-butes	MATINVAT	0533	3	SPP, {CHARS[48] | NULL}, SPP	-	-	21-18
Materialize Invocation Entry	MATINVE	0547	3	CHARV, {CHARV.1. | NULL}, CHARS.1. | NULL}	-	-	21-28
Materialize Invocation Stack	MATINVS	0546	2	SPP, {S-PTR | NULL}	-	-	21-32
Materialize Machine Attri-butes	MATMATR	0636	2	SPP, CHARS[2] | SPP	-	-	22-4
Materialize Machine Data	MATMDATA	0522	2	CHAR, {CHAR [2] | NUMERICV }	-	-	22-30
Materialize Program	MATPG	0232	2	SPP, PROGRAM	-	-	8-24

Operation Description	Mnemonic	Op Code	Number of Operands	Operand Syntax	Resulting Conditions	Optional Forms	Page
Materialize Process Activation Groups	MATPRAGP	0331	1	SPP	-	-	18-3
Materialize Process Attributes	MATPRATR	0333	3	SPP, {PCS | NULL}, CHARS[1]	-	-	18-5
Materialize Process Record Locks	MATPRECL	031E	2	SPP, SPP	-	-	13-20
Materialize Process Locks	MATPRLK	0312	2	SPP, {PCS | NULL}	-	-	13-17
Materialize Process Message	MATPRMSG	039C	4	SPP, SPP, {SPP | NULL}, SPP	-	-	15-3
Materialize Pointer	MATPTR	0512	2	SPP, {S-PTR | D-PTR | SPP | I-PTR}	-	-	21-37
Materialize Pointer Locations	MATPTRL	0513	3	SPP.2., BINS	-	-	21-46
Materialize Queue Attributes	MATQAT	0336	2	SPP, QUEUE	-	-	12-12
Materialize Queue Messages	MATQMSG	033B	3	SPP, S-PTR, CHARS[16]	-	-	12-16
Materialize Resource Management Data	MATRMD	0352	2	SPP, CHARS[8]	-	-	19-9
Materialize Space Attributes	MATS	0036	2	SPP, S-PTR	-	-	6-11
Materialize Selected Locks	MATSELLK	033E	2	SPP, {S-PTR | SPP}	-	-	13-24
Materialize System Object	MATSOBJ	053E	2	SPP, S-PTR	-	-	21-48
Materialize User Profile	MATUP	013E	2	SPP, USER PROFILE	-	-	17-25
Modify Automatic Storage Allocation	MODASA	02F2	2	{SPP | NULL}, BINS	-	-	9-23
Modify Exception Description	MODEXCPD	03EF	3	EXCP-DESC, SPP, CHARS[4]	-	-	14-6
Modify Independent Index	MODINX	0452	2	S-PTR, CHARS[4]	-	-	11-23
Modify Space Attributes	MODS	0062	2	S-PTR, BINS	-	-	6-15
Multiply	MULT	104B	3+[B]	NUMERICV, NUMERICS.2., [BT.4. | IT.4.]	P, N, Z, UN	[B | I, S, R]	2-177

Operation Description	Mnemonic	Op Code	Number of Operands	Operand Syntax	Resulting Condi-tions	Optional Forms	Page
Negate	NEG	1056	2+[B]	NUMERICV, NUMERICS, [BT.4. | IT.4.]	P, N, Z, UN	[B | I, S]	2-181
No Operation	NOOP	0000	0	-	-	-	10-3
No Operation and Skip	NOOPS	0001	1	UBINI	-	-	10-4
Not	NOT	108A	2+[B]	CHARV, CHARS, [BT.2. | IT.2.]	Z, NZ	[B | I, S]	2-184
Or	OR	1097	3+[B]	CHARV, CHARS.2., [BT.2. | IT.2.]	Z, NZ	[B | I, S]	2-187
Override Program Attri-butes	OVRPGATR	0006	2	UBINI.2.	-	-	10-5
Remainder	REM	1073	3+[B]	NUMERICV, NUMERICS.2., [BT.3. | IT.3.]	P, N, Z	[B | I, S]	2-190
Retrieve Excep-tion Data	RETEXCPD	03E2	2	SPP, CHARS[1]	-	-	14-9
Remove Inde-pendent Index Entry	RMVINXEN	0484	4	{SPP | NULL}, INDEX, SPP.2.	-	-	11-26
Resolve Data Pointer	RSLVDP	0163	3	D-PTR, {CHARS[32] | NULL}, {S-PTR | NULL}	-	-	4-14
Resolve System Pointer	RSLVSP	0164	4	S-PTR, {CHARS[34] | NULL}, {S-PTR | NULL}, {CHARS[2" | NULL}	-	-	4-17
Return From Exception	RTNEXCP	03E1	1	SPP	-	-	14-12
Return External	RTX	02A1	1	{BINS | NULL}	-	-	9-25
Scale	SCALE	1063	3+[B]	NUMERICV, NUMERICS, BINS, [BT.4. | IT.4.]	P, N, Z, UN	[B | I, S]	2-194
Scan	SCAN	10D3	3+[B]	{BINV | B-ARRAY}, CHARS.2., [BT.3. | IT.3.]	P, Z, NCO	[B | I]	2-198
Scan With Control	SCANWC	10E4	4+[B]	SPP, CHARV[8], CHARS[4], [BT.4. | IT.4.]	H, L, E, NF	[B, I]	2-201

Operation Description	Mnemonic	Op Code	Number of Operands	Operand Syntax	Resulting Conditions	Optional Forms	Page
Search	SEARCH	1084	4+[B]	{BINV | B-ARRAY}, {N-ARRAY | C-ARRAY}, {CHARV | NUMERICV}, BINS, {BT.2. | IT.2.}	P, Z	[B | I]	2-209
Set Access State	SETACST	0341	1	SPP	-	-	19-31
Set Argument List Length	SETALLEN	0242	2	ARG-LIST, BINS	-	-	9-27
Set Bit in String	SETBTS	101E	2	{CHARV | NUMERICV}, BINS	-	-	2-212
Set Data Pointer	SETDP	0096	2	D-PTR {NUMERICV | N-ARRAY | CHARV | C-ARRAY}	-	-	5-7
Set Data Pointer Addressability	SETDPADR	0046	2	D-PTR {NUMERICV | N-ARRAY | CHARV | C-ARRAY}	-	-	5-9
Set Data Pointer Attributes	SETDPAT	004A	2	D-PTR, CHARS[7]	-	-	5-11
Set Heap Space Storage Mark	SETHSSMK	03B6	2	SPP, BIN	-	-	7-25
Set Invocation Exit	SETIEXIT	0252	2	S-PTR, ARG LIST NULL	-	-	9-29
Set Instruction Pointer	SETIP	1022	2	I-PTR, F-BT	-	-	2-214
Set System Pointer From Pointer	SETSPFP	0032	2	S-PTR, {D-PTR | SPP | S-PTR | I-PTR}	-	-	4-24
Set Space Pointer	SETSPP	0082	2	SPP, {CHARV | C-ARRAY | NUMERICV | N-ARRAY | PTR | P-ARRAY}	-	-	5-14
Set Space Pointer with Displacement	SETSPPD	0093	3	SPP, {CHARV | C-ARRAY | NUMERICV | N-ARRAY | PTR | P-ARRAY}	-	-	5-16
Set Space Pointer From Pointer	SETSPPFP	0022	2	SPP, {S-PTR | D-PTR | SPP}	-	-	4-22

Operation Description	Mnemonic	Op Code	Number of Operands	Operand Syntax	Resulting Condi-tions	Optional Forms	Page
Set Space Pointer Offset	SETSPPO	0092	2	SPP, BINS	-	-	5-18
Signal Exception	SIGEXCP	10CA	2+[B]	SPP.2., [BT.2. | IT.2.]	I, DE	[B | I]	14-19
Sense Exception Description	SNSEXCPD	03E3	3	SPP.3.	-	-	14-15
Store and Set Computational Attributes	SSCA	107B	3	CHARV[5], {CHARS[5] | NULL}, {CHARS[5] | NULL}	-	-	2-216
Store Param-eter List Length	STPLLEN	0241	1	BINV	-	-	9-31
Store Space Pointer Offset	STSPPO	00A2	2	BINV, SPP	-	-	5-20
Subtract Logical Character	SUBLC	1027	3+[B]	CHARV, CHARS.2., [BT.4. | IT.4.]	Z, NZ, C, NC	[B | I, S]	2-220
Subtract Numeric	SUBN	1047	3+[B]	NUMERICV, NUMERICS.2., [BT.4. | IT.4.]	P, N, Z, UN	[B | I, S, R]	2-223
Subtract Space Pointer Offset	SUBSPP	0087	3	SPP.2., BINS	-	-	5-22
Subtrace Space Pointer For Offset	SUBSPPFO	0033	3	BINS, SPP.2.	-	-	5-24
Test Authority	TESTAU	10F7	3	{CHARV[2] | NULL}, {S-PTR | SPDO}, CHARS[2]	-	-	17-29
Test Extended Authorities	TESTEAU	10FB	3	{CHARV[8] | NULL}, CHARS[8], {BINS[2] | NULL}	-	-	17-34
Test Exception	TESTEXCP	104A	2+[B]	SPP, EXCP-DESC, [BT.2. | IT.2.]	S, NS	[B | I]	14-24
Trim Length	TRIML	10A7	3	NUMERICV, CHARS, CHARS[1]	-	-	2-240
Test Bit in String	TSTBTS	100E	2+B	{CHARS | NUMERICS}, BINS, {BT.2. | IT.2.}	Z, O	{B | I}	2-229
Test Bits Under Mask	TSTBUM	102A	2+B	{CHARS | NUMERICS}.2., {BT.3. | IT.3.}	Z, O, M	{B | I}	2-231
Test and Replace Char-acters	TSTRPLC	10A2	2	CHARV, CHARS	-	-	2-227

Operation Description	Mnemonic	Op Code	Number of Operands	Operand Syntax	Resulting Condi- tions	Optional Forms	Page
Unlock Object	UNLOCK	03F1	1	SPP	-	-	13-30
Unlock Space Location	UNLOCKSL	03F2	2	SPP, CHARS[1]	-	-	13-33
Verify	VERIFY	10D7	3+[B]	{BINV | B-ARRAY}, CHARS.2. [BT.2. | IT.2.]	P, Z	[B | I]	2-242
Wait On Time	WAITTIME	0349	1	CHARS.16.	-	-	18-18
Transfer Control	XCTL	0282	2	PROGRAM | SPP, {ARG LIST | NULL}	-	-	9-33
Transfer Object Lock	XFRLOCK	0382	2	PCS, SPP	-	-	13-27
Translate	XLATE	1094	4	CHARV, CHARS, {CHARS | NULL}, CHARS	-	-	2-233
Translate With Table	XLATEWT	109F	3	CHARV, CHARS, CHARS	-	-	2-235
Exclusive Or	XOR	109B	3+[B]	CHARV, CHARS.2., [BT.2. | IT.2.]	Z, NZ	[B | I, S]	2-164

Index

A

absolute instruction number 1-1
activation entry 9-3
activation group 9-3, 9-7, 9-20, 9-21, 9-35, 18-3
  activation count 9-21
  heap space count 9-21
  mark 9-21
  name 9-21
  root program 9-21
  storage address recycling key 9-21
activation group heap list 9-20
activation group mark 18-3
activation mark 9-3
actvation entry 9-7, 9-35
array 1-4
authorization management instructions 17-1
authorization required 1-3
automatic storage frame (ASF) 9-7, 9-35

B

byte string 1-1

C

Character 1-3
compound operands 1-1
  explicit base 1-2
  subscript 1-1
  substring 1-2
computation and branching instructions 2-1
context management instructions 16-1

D

data pointer 1-4
data pointer defined scalar 1-4
dump space management instructions 20-1

E

Exceptions 1-3, 14-1
  management instructions 14-1
  specifications 23-1
external entry point 9-7, 9-35

H

heap management instructions 7-1

I

immediate operands 1-1
IMPL (initial microprogram load) 22-5
IMPLA (initial microprogram load abbreviated) 22-5
independent index instructions 11-1
initial microprogram load (IMPL) 22-5
initial microprogram load abbreviated (IMPLA) 22-5
instruction definition list element 1-4
instruction format
  authorization required 1-3
  Exceptions 1-3
  limitations 1-3
  lock enforcement 1-3
  resultant conditions 1-3
instruction forms
  number 1-4
  pointer 1-4
instruction summary A-1
invocation count 9-7, 9-13, 9-35
invocation exit and deactivation 9-13
invocation mark 9-7, 9-35
invocation number 9-7, 9-35
invocation type 9-7, 9-35
IPL (initial program load) 22-11

L

LEAR (lock exclusive allow read) 13-4
LENR (lock exclusive no read) 13-4
limitations 1-3
lock enforcement 1-3
lock exclusive allow read (LEAR) 13-4
lock exclusive no read (LENR) 13-4
lock management instructions 13-1
lock shared read (LSRD) 13-4
lock shared read only (LSRO) 13-4
LSRD (lock shared read) 13-4
LSRO (lock shared read only) 13-4
LSUP (lock shared update) 13-4

M

machine initialization status record (MISR) 22-5, 22-11
machine interface support functions instructions 22-1
machine observation instructions 21-1
mark count 9-7, 9-35
Materialize Invocation Attributes 9-7, 9-35
MISR (machine initialization status record) 22-5, 22-11

N

null operands 1-1

© Copyright IBM Corp. 1991, 1993

O

object
object mapping table 8-25
ODT object 1-1
Operand
  Syntax 1-3

P

pointer 1-4
pointer/name resolution addressing instructions 4-1
process management instructions 18-1
program
  execution instructions 9-1
  management instructions 8-1
program creation control instructions 10-1
program entry procedure 9-7, 9-35

Q

queue management instructions 12-1
queue space management instructions 15-1

R

relative instruction number 1-4
resource management instructions 18-19
resultant conditions 1-3

S

Scalar 1-3
signed binary 1-1
simple operands 1-1
space addressing instructions 5-1, 5-3
space management instructions 6-1
space pointer 1-4
static storage 9-3
static storage initialization 9-3
Syntax definition
  array 1-4
  Character 1-3
  data pointer 1-4
  data pointer defined scalar 1-4
  instruction definition list element 1-4
  instruction number 1-4
  instruction pointer 1-4
  Numeric 1-3
  pointer 1-4
  relative instruction number 1-4
  Scalar 1-3
  space pointer 1-4
  space pointer data object 1-4
  system pointer 1-4
  variable scalar 1-4
system pointer 1-4

U

unsigned binary 1-1

V

variable scalar 1-4
Vital Product Data (VPD) 22-17
VPD (vital product data) 22-17

Printed in U.S.A.