CRAY RESEARCH, INC.

CRAY X-MP AND CRAY-1® COMPUTER SYSTEMS

FORTRAN (CFT) INTERNAL REFERENCE MANUAL

SM-0017

Copyright© 1980, 1981, 1983, 1984 by CRAY RESEARCH, INC.
This manual or parts thereof may not be reproduced in any form without permission of CRAY RESEARCH, INC.

RECORD OF REVISION

CRAY RESEARCH, INC.

PUBLICATION NUMBER SM-0017

Each time this manual is revised and reprinted, all changes issued against the previous version in the form of change packets are incorporated into the new version and the new version is assigned an alphabetic level. Between reprints, changes may be issued against the current version in the form of change packets. Each change packet is assigned a numeric designator, starting with 01 for the first change packet of each revision level.

Every page changed by a reprint or by a change packet has the revision level and change packet number in the lower righthand corner. Changes to part of a page are noted by a change bar along the margin of the page. A change bar in the margin opposite the page number indicates that the entire page is new; a dot in the same place indicates that information has been moved from one page to another, but has not otherwise changed.

Requests for copies of Cray Research, Inc. publications and comments about these publications should be directed to:

CRAY RESEARCH, INC.,
1440 Northland Drive,
Mendota Heights, Minnesota 55120

Revision	Description
	October, 1980 - Original printing.
A	June, 1981 - This reprint with revision includes implementation of the character data type and other miscellaneous changes that bring the manual into agreement with the version 1.10 release. All previous printings are obsolete.
A-01	May, 1983 - This change packet brings the manual into agreement with the CFT 1.11 release. Major changes include the addition of intrinsic function processing, the #CL Text Table (TBCLTXT), the Substring Definition Table (TBSB), the Intrinsic Function Name Table (TBJ), the Intrinsic Function Attribute Table (TBK), calling sequence information, and the following subroutines: CCAT, CCLA, CCLO, CCRS, CCTB, CKRF, CLCF, CLGA, CLOF, CLOG, CLRS, CLSZ, CLTG, CPOP, FRTG, FSHD, FSUB, IPRN, IRST, MSAR, MVOP, NARG, SIN, and SOPT. The register numbers were moved to tables 4-1 and B-1. Miscellaneous technical and editorial changes are also included.
A-02	July, 1983 - This change packet, along with A-01, brings the manual into agreement with the CFT 1.11 release. The default of IF optimization is changed from OPT=PARTIALIFCON to OPT=NOIFCON on the CFT control statement.
B	November, 1983 - This reprint with revision incorporates revision A and change packets A-01 and A-02. No other changes have been made.

B-01 February, 1984 - This change packet brings the manual into agreement with the CFT 1.13 release. The CFT release has been numbered 1.13 in conjunction with the 1.13 COS release. Major changes include the addition of reentrancy support, new instruction scheduler information, the Entry/Exit Address Table (TBEE), the Call-by-value Reference Table (TBFR), the Program Description Table (TBH), the Plus Dependency Table (TBPD), the Sequence Number Table (TBSN), the Saved Variable Table (TBSV), the following subroutines: ABRA, ARUS, ASVL, ASVM, BLCN, BTD, COPR, CRAR, CRRG, CRVR, DORP, EVOP, FLVS, FSTK, GCRF, GIXA, GSBS, ISRF, LSOM, LSOV, MCEX, NICV, NOCV, PMRT, PMST, PPDP, RBIN, RBLI, RBRG, RCCK, RSTB, SAST, SA50, SDCO, SPFH, SPFR, TPRU, ZMEM, and Appendixes D and E. Miscellaneous technical and editorial changes are also included.

B-02 December, 1984 - This change packet brings the manual into agreement with the CFT 1.14 release. Major changes include the addition of the Block Definition Table (TBBK), the Register Variables to Restore After a CALL Table (TBCALL), the Character Length Table (TBCLEN), the Conjunctive Term Table (TBCT), the Disjunctive Term Table (TBDT), the Label Usage Table (TBLB), the TBT Index of Variables Not Assignable to B/T Register Table (TBNOBT), the TBX Extension Table (TBXX), and the following subroutines: AIBF, CDPR, CEXP, CFBI, CIDN, CQYL, CRMV, CRNK, CTTY, DOUN, EBSN, EBXR, EBXS, ECNT, ECNU, EDJT, EDJU, ERTX, ESBK, ESNL, ETBX, FPAR, GBAT, GCBS, GLBD, GTCB, IGXF, IVTX, LBLK, NOBTVAR, PBLK, PCIV, PCST, PLDP, RBMV, RDPT, ROSR, SDPF, SFMN, SGES, SPRN, TFBK, and VLAN. Miscellaneous technical and editorial changes are also included.

PREFACE

This publication is part of a set of manuals written for programmers, analysts, and field engineers who have the responsibility of installing, debugging, and modifying the Cray Operating System (COS).

This manual describes the internal design of the Cray FORTRAN Compiler (CFT), Version 1.

Section 1 briefly introduces the compiler and describes CFT conventions.

Sections 2 and 3 provide a basis for understanding the compiler's operation. These sections describe the general flow of the compiler through Passes 1 and 2, respectively.

Section 4 describes table management and gives the specifications for the tables used by the compiler.

Section 5 gives the details of the major subroutines within the compiler.

Section 6 provides information about CFT I/O.

The appendix section gives additional information about the compiler's internal design, including pertinent reference information and sample code.

Other publications in this set are:

SM-0007	IOS Table Descriptions Internal Reference Manual
SM-0040	COS EXEC/STP/CSP Internal Reference Manual
SM-0041	COS Product Set Internal Reference Manual
SM-0042	Front-end Protocol Internal Reference Manual
SM-0043	COS Operational Procedures Reference Manual
SM-0044	COS Operational Aids Reference Manual
SM-0045	COS Table Descriptions Internal Reference Manual
SM-0046	IOS Software Internal Reference Manual
SM-0049	Data General Station (DGS) Internal Reference Manual
SM-0072	COS Simulator (CSIM) Internal Reference Manual
SR-0073	COS Simulator (CSIM) Reference Manual

Manuals designated as internal describe the internal design of the software whereas the other manuals in the set define procedures and external features of tools needed for installing and maintaining CRI software.

The reader is assumed to be familiar with the contents of the CRAY-OS Version 1 Reference Manual, publication SR-0011 and have experience in coding the Cray Assembly Language (CAL) as described in the CAL Version 1 Reference Manual, publication SR-0000.

All values specified in this manual are expressed in octal unless otherwise noted.

CONTENTS

FIGURES

TABLES

INDEX

COMPILER OVERVIEW 1

1.1 GENERAL DESCRIPTION

The CRAY-1 FORTRAN Compiler (CFT) is a 2-pass compiler that converts statements from the FORTRAN language to the binary machine language of the CRAY-1 Computer Systems. CFT constructs CRAY-1 machine-language instruction sequences that cause the full range of CRAY-1 features and capabilities to be applied during program execution.

CFT is written in CRAY-1 Assembly Language (CAL) and executes under control of the CRAY-1 Operating System (COS). It has no hardware requirements beyond those required for the minimum system configuration.

CFT analyzes and compiles source code one program unit (main program, subroutine, function, or block data subroutine) at a time. No information is retained from one program unit to the next.

During compilation, CFT constructs a number of tables in the user area. A *table* is basically a list of information kept for referencing by the compiler. Examples of the tables maintained by CFT are a list of the symbols used in the source program and a list of constants encountered.

The compiler tables are located at the high end of user memory immediately below the user Dataset Parameter Area. Space is allocated to each table as needed. Table length is variable, with most tables expanding as compilation proceeds. If the table area overflows its allotted memory, CFT requests additional memory from the operating system.

Figure 1-1 illustrates the 2-pass philosophy of CFT. The input to Pass 1 of CFT is the source input dataset consisting of the source code file and the accompanying control statement. The principal output from Pass 1 is a copy of the source program translated to an internal format and a symbol table describing the attributes of the symbols encountered in the source program. The Pass 1 output becomes the input to Pass 2. Pass 2 finishes compilation and provides as its output the loader tables, suitable for loading and execution.

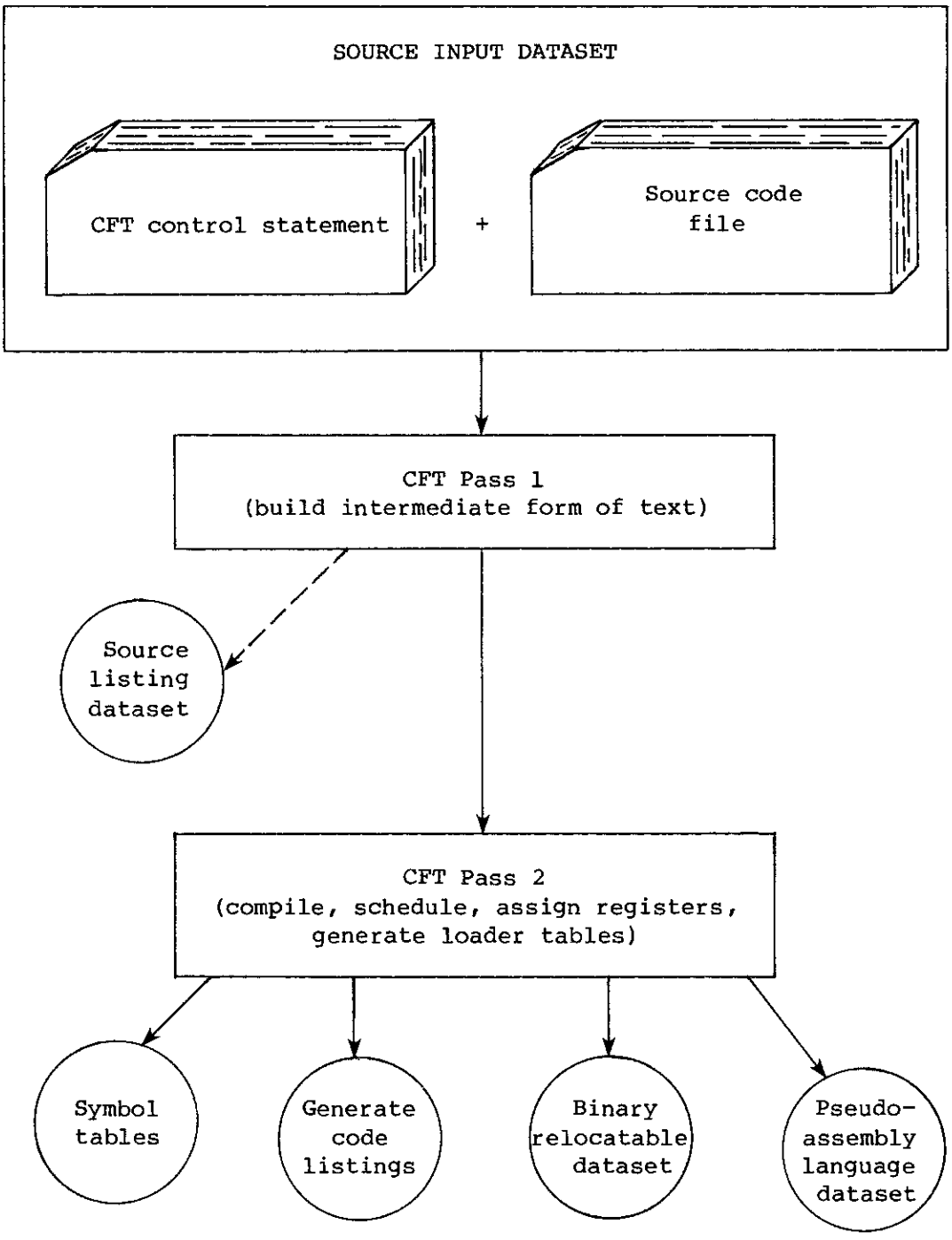


Figure 1-1. CFT's 2-pass philosophy

The compiler is loaded in the user field by the operating system with a CFT control statement. Each user receives a copy of the compiler in the user field. The compiler itself is not re-entrant. Control statement parameters specify characteristics of the compiler run, such as the datasets containing source statements and list output.

1.1.1 PASS 1

Basically, Pass 1 of CFT converts the source code to an intermediate form to facilitate Pass 2 activities. Figure 1-2 illustrates the general flow of Pass 1. A brief summary of Pass 1 activity follows. Section 2 of this manual provides a more detailed description of the flow through Pass 1.

After CFT reads and interprets the CFT control statement, it initializes the tables, presets default values, and opens the datasets required by the job. The preset defaults retain their values for any options not specified by the user on the CFT statement.

At this point, the main loop of the compiler begins. Most pointers are cleared and the tables are set to empty. The statement sequence number is initialized at 0. This procedure is executed at the start of each program unit. Then, compilation begins by calling CFT's card reader driver. The card reader driver reads the source input file statement by statement, checking for continuation cards and comment cards.

As each statement is read, it is copied into the compiler's statement buffer, where it is examined and classified by type. Then, depending on statement type, control transfers to one of the many different unique statement processors. A *unique statement processor* is a handler for only one specific FORTRAN statement; for example, there is a unique statement processor for WRITE statements and another for DIMENSION statements.

Each FORTRAN statement is classified as either executable or non-executable. An *executable* statement specifies an action, while a *non-executable* statement is an inactive descriptor of data (declarative) or program form.

With few exceptions, statements are completely processed as they are encountered. Exceptions to this are the DO and EQUIVALENCE statements. DO statements generate table entries that trigger additional processing when the terminal statement is encountered. EQUIVALENCE statements are packed into a table as they are encountered and are processed when the first executable statement (that is, the last declarative statement) is encountered.

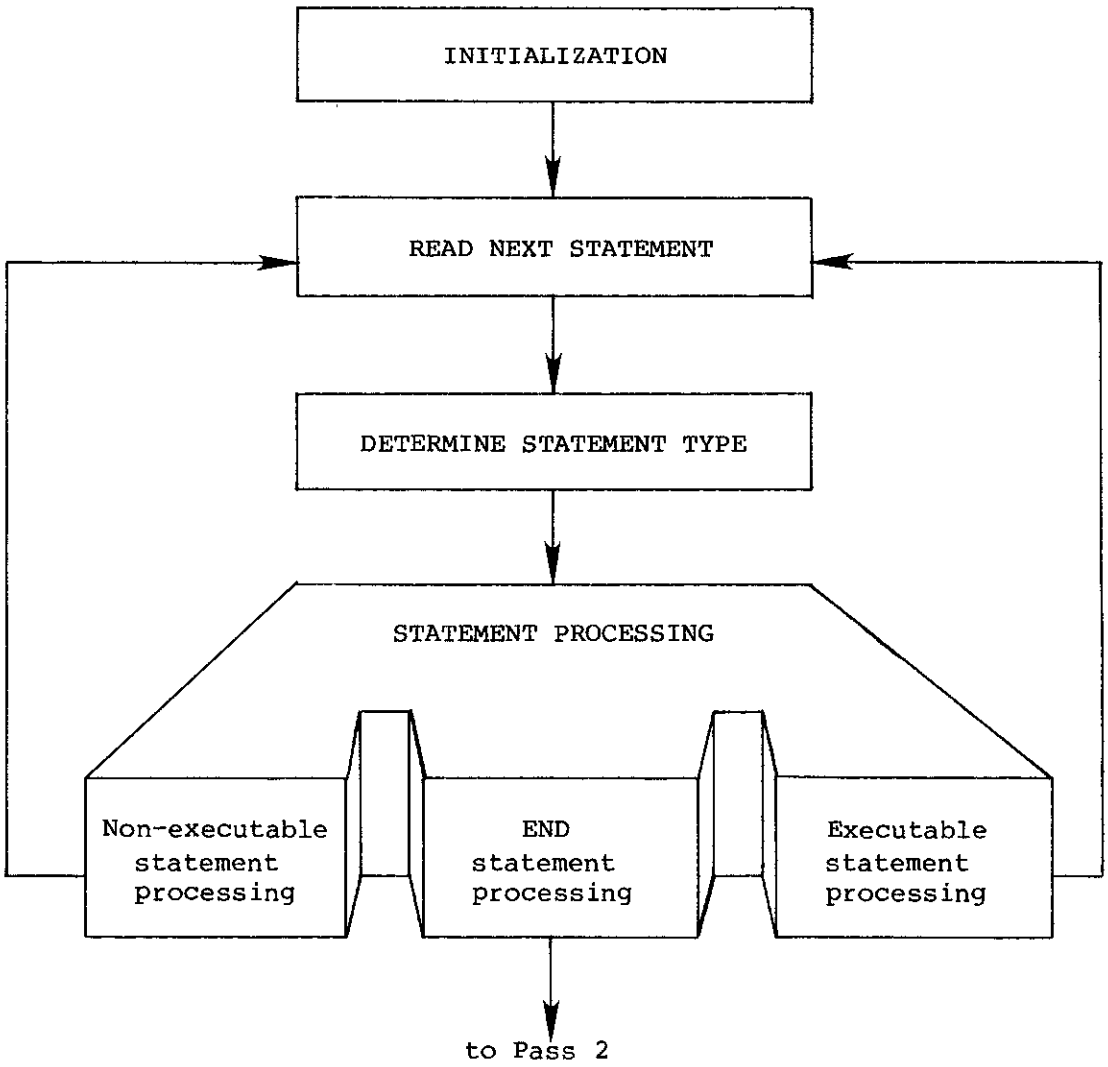


Figure 1-2. Pass 1 overview

Many FORTRAN statements have similar syntax. For example, the syntaxes of READ, WRITE, and PRINT statements are similar, as are those of REAL, DIMENSION, and COMMON statements. In such cases, one of the unique statement processors is called to process the initial keyword. This unique statement processor then branches to an appropriate *common syntax processor* to handle the syntax held in common by the statements.

Generally, each declarative statement makes entries in the Symbol Table specifying attributes for the named entity.

Each executable statement is converted by CFT to an internal notation and stored in a separate table called the Tag Buffer Table. The internal notation used in the Tag Buffer Table consists of a 1-word statement header entry containing information on the characteristics of the statement, followed by a series of 1-word tag and operator entries.

CFT constructs a *tag* for each lexical entity in the statement. This tag describes the attributes of the entity (variables, statement numbers, and constants, for example).

Each *operator* entry specifies the processing precedence of that operator within the FORTRAN statement. The Tag Buffer represents each operator by a precedence-level-related numerical value. (See section 4 for a list of these values.) The entry for a parenthesis points to the matching open or close parenthesis.

At the end of Pass 1, the Tag Buffer contains a nearly literal one-for-one translation of the input statement in CFT internal notation.

Statement functions are translated into tag-and-operator sequences and copied to a separate table. When a reference to the statement function occurs in an executable statement, the tags and operators representing the statement function are copied from the table to the Tag Buffer, the actual arguments being substituted for the dummy arguments.

When the END statement is encountered, all variables are assigned addresses. Statement number references are linked and the contents of the Tag Buffer are transferred to a Tag Buffer Table. Finally, the compiler initializes the loader tables and initializes parameters for Pass 2.

1.1.2 PASS 2

Pass 2 cycles through the program unit, identifying code blocks. These code blocks are handled one at a time, in order of occurrence. Each block is compiled individually, registers are assigned, and the loader tables for that block are generated. When an END statement is encountered, END statement processing for the program unit takes place. Unless an end of file is encountered, control transfers back to Pass 1 to reinitialize for the next program unit.

Figure 1-3 illustrates the basic flow of Pass 2. A brief summary of Pass 2 activity follows; section 3 of this manual provides a more detailed description of the flow through Pass 2.

The first task of Pass 2 is to identify a code block. A block is delimited by a loop boundary, the subprogram boundary, an ENTRY statement, or an active label with a transfer from outside the block. A block has only one entry point, but may have more than one exit point. Code blocks are defined this way so that the code generator and optimizer can analyze them more easily.

When a code block has been defined, the compiler determines if it is an innermost DO-loop and therefore subject to vector analysis. If vector analysis is indicated, CFT checks for proper subscripting and for calls to library functions that have known vector versions. CFT also makes sure there are no dependencies. If all of these conditions are met, vectorization within the code block may be possible.

If there are ambiguous dependencies and CFT can determine a condition for safe compilation of the loop, vectorization may be possible. These loops are called conditional vector loops and are compiled as two loops, one vector loop and one scalar loop. A run-time test determines which loop will be executed.

Next, code is generated for the code block. A pass is made on the block, statement-by-statement. Operations are parsed as triads, and hardware instruction codes are selected. Register assignment at this point, however, is internal to the compiler. CFT assigns pseudo registers where required by the operation code. References to intrinsic functions are expanded to intermediate instructions.

Scheduling occurs after a code block is translated to intermediate code. Scheduling involves breaking the block into smaller, separate groups, determining the instruction dependencies, and determining which instruction sequences take the most time to execute.

After an instruction group is ordered on instruction time, hard registers are assigned and the final instruction order is determined. When all instructions are assigned hard registers, the completed instructions are packed and transferred to loader text tables. The next group is selected from the block and the scheduling and register assignment repeated. When there are no more groups in the block, control transfers back to the selection of the next block. At the end of the program unit, END statement processing occurs and control transfers back to Pass 1 to begin processing the next program unit.

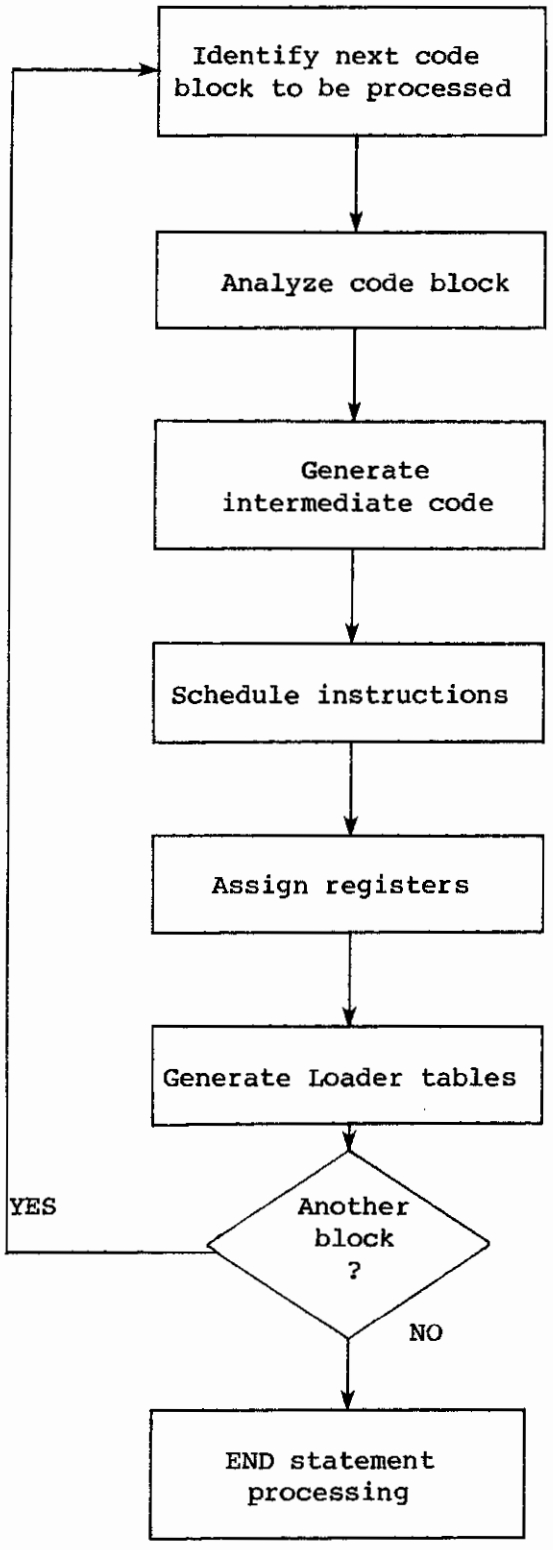


Figure 1-3. Pass 2 overview

1.2 TABLE NAMES AND INDEXES

CFT maintains nearly 50 tables during compilation. Initially, all the tables are empty. Then, as compilation proceeds, the tables expand in memory.

Several conventions exist for naming tables. Each table has a unique name, often a mnemonic for the function the table performs. The form is TBx, where x is the table identifier. For example, a table that maintains array information is TBA, the Array Table.

The index to a table is of the form KTx. For example, the index to TBA is KTA. Pointer words to the CFT tables are maintained in register V7. The pointer word contains the first word address (FWA) and the last word address + 1 (LWA+1) for the table. (Refer to Appendix B for more information.)

Table indexes are relative to the beginning address of the appropriate table. Except for the Symbol Table, the Tag Buffer Table (TBG), and the Program Unit Name Table (TBPN), the FWA of a table is always a multiple of 100_8, while the LWA+1 changes as necessary.

Refer to the section entitled Compiler Tables for table descriptions and a summary of table management.

1.3 CFT MEMORY ORGANIZATION

The compiler is loaded for each CFT control statement and is reinitialized rather than reloaded for each program unit. In a multiprogramming environment, several copies of the compiler can be in memory at one time because a copy goes in the field of each CFT user.

Figure 1-4 illustrates the organization of the memory area occupied by one FORTRAN user job. CFT code is at the low end of the user field, with the various routines comprising CFT arranged in approximately alphabetical order. Immediately above that is an area allocated for two compiler tables, the Library Name Table (TBL) and the Library Macro Table (TBM). Above that is the Record Image Buffer (RIB), which contains one source line at a time, followed by the library routines. During Pass 1, the statement currently being processed by CFT is stored in the Character Buffer (CHB), which is at the low end of blank common.

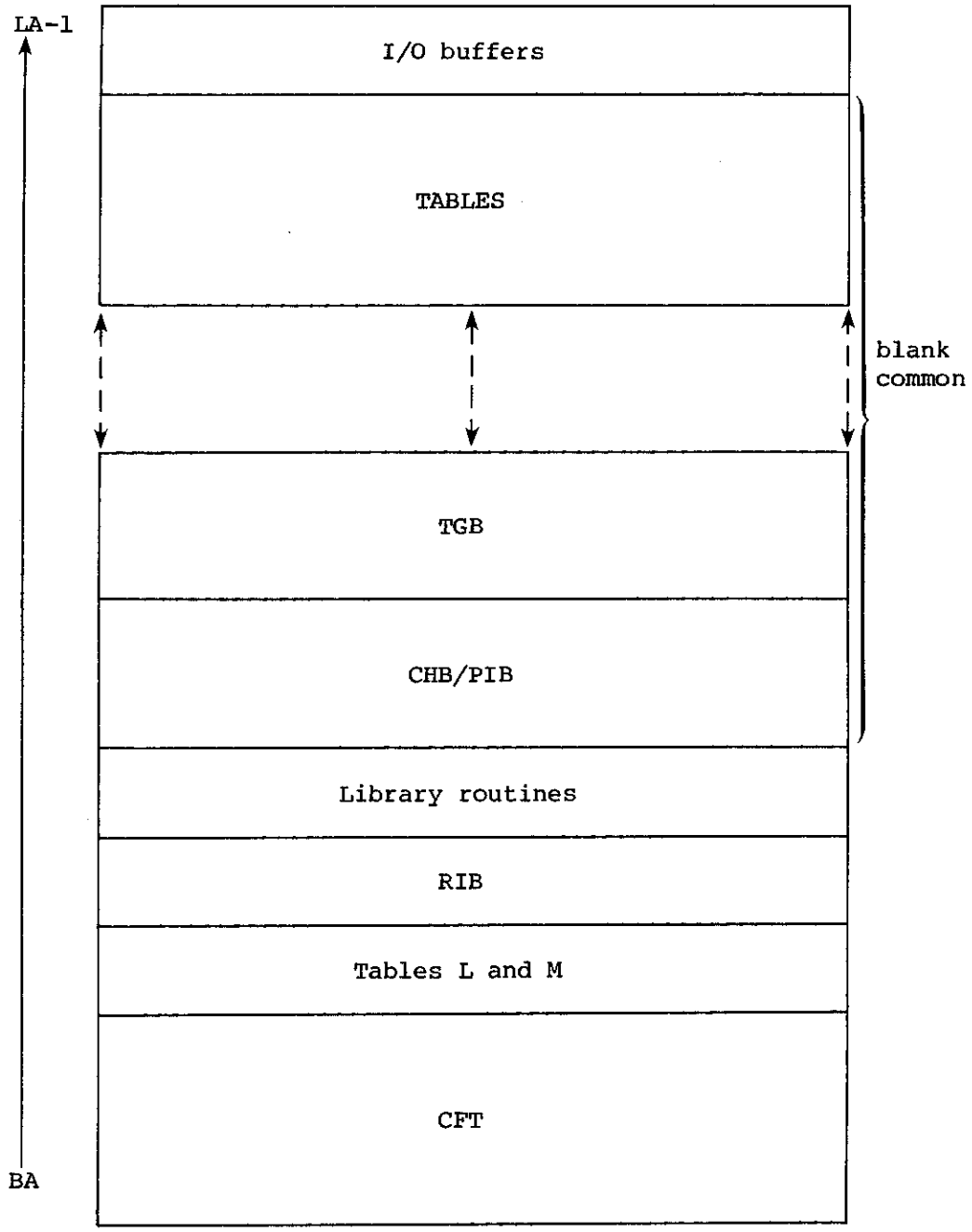


Figure 1-4. CFT memory organization

Immediately above CHB is the Intermediate Tag Buffer (TGB), containing the intermediate form of the text during Pass 1. At the end of Pass 1, the contents of TGB are moved to the Tag Buffer Table (TBG). Then, during Pass 2, the Pseudo Instruction Buffer (PIB) is built in the area previously occupied by CHB and TGB.

I/O buffers are allocated at the high end of the user field. TGB and the other tables maintained by the compiler are located just below the I/O buffer area in blank common. As each new statement is processed, the tables expand downward in memory and TGB expands upward. If the available memory is exhausted, CFT requests more memory from the operating system. The additional memory is inserted between the tables and TGB so, in theory, the two never meet.

Experience shows that, exclusive of I/O buffers, a typical program can be compiled in 64K words ($200{,}000_8$). Since compilation occurs one program unit at a time, each program unit is completely generated and the loader tables written out before a new unit is begun. The source listing is produced as Pass 1 is read. The generated code is listed at completion of compilation for each code block. Compiler code, TBPG, and TBPN, are saved from one program unit to the next. All of the loader tables for a given program unit are saved in memory and written out at one time. Nothing else is saved from one program unit to the next.

PASS 1 FLOW 2

2.1 INTRODUCTION

Pass 1 of the CRAY-1 FORTRAN Compiler performs the following functions:

- Compiler initialization
- Statement-by-statement processing of the source file:
  - Read source statement
  - Determine statement type
  - Process statement
- END processing for Pass 1

The input to Pass 1 is the source input file. The main output from Pass 1 is the information contained in three tables:

- The Symbol Table (TBS) - Contains the names of all symbols in the source program
- The Tag Table (TBT) - Holds the attributes of all of the symbols listed in TBS
- The Tag Buffer Table (TBG) - Contains a copy of the source program translated into an internal tag-and-operator format

2.2 INITIALIZATION

Routine BGIN (begin compilation) handles both initialization at the beginning of processing and initialization at the start of each program unit. Control transfers to BGIN at the start of processing to initialize the compiler. Then, each time processing begins on a new program unit, control transfers to BG10, a location within the routine BGIN, to reinitialize for that program unit.

2.2.1 INITIALIZATION AT BGIN

BGIN calls the control card cracking routine CARD to read the CFT control statement. A copy of the CFT statement is stored in words 5 through 77 of the operating system's Job Communication Block. Once the CFT statement is decoded, CARD determines what datasets are required and opens them.

CARD also collects the list options and error processing options in the CFT statement and sets indicator bits in register T.OCW to show whether the corresponding options are on or off. Any options not specified by the user on the CFT control statement are set to the default values. The default options are generally contained in the block of constants at the front of the compiler and can be changed at compiler assembly time.

2.2.2 INITIALIZATION AT BG10

The main loop of the compiler begins at label BG10. Control returns to BG10 at the start of each program unit to reinitialize CFT. Pointers are cleared and table pointers are reset to indicate that the tables are empty except for the Program Unit Name Table and the Page Number Table (TBPN and TBPG), which are saved from pass to pass. Storage registers are zeroed and the statement sequence number is initialized at 0.

Compilation begins with a call to routine RNXT.

2.3 READ SOURCE STATEMENT

Routine RNXT (read next statement) is the compiler's card reader driver. RNXT reads the input dataset ($IN or its equivalent) statement by statement, checking for statement continuation and comments. An internal buffer builds one complete FORTRAN statement at a time into a buffer. If a source listing is requested, lines are written to the listing file as they are read.

Two buffer areas are used to assemble a statement. The Record Image Buffer (RIB) can accommodate one card image and contains the image of the next card to be processed. The Character Buffer (CHB) contains one complete FORTRAN statement character image including an initial line and up to 19 continuation lines.

The First Card Buffer (FCB) is maintained if no source listing was requested. The FCB receives the first line of each statement so that the line can be printed out if an error occurs during processing.

RIB holds columns 1 through 96 of the next card to be processed. Characters beyond 96 are discarded. Upon entry, RNXT examines the contents of RIB. If RIB contains a blank line or a comment line (indicated by an asterisk or the letter C in column 1), the line is copied to the output dataset ($OUT or its equivalent), if requested. Each comment line is copied out in turn until the first non-comment line is encountered.

Columns 1 through 72 of this initial non-comment line are copied from RIB into CHB. The next line is read into RIB. RNXT disregards any comments contained in the comment field (columns 73 through 96) of this initial line. RNXT also checks to see if the initial line in CHB has a statement number. (A statement number is contained in columns 1 through 5.) If a statement number is present, it is stripped from the card image and saved for later processing where it is then assembled, eliminating blanks and leading zeros. Statement number entries are made in the Symbol Table and the Tag Table (TBS and TBT).

If the initial non-comment line is not an END statement, RNXT determines if RIB contains a continuation line. Columns 7 through 72 of any continuation lines are copied line by line until a non-continuation line or a comment line is in RIB. Any comment lines encountered within a continued statement are skipped and are copied to the output dataset, if requested. Finally, CHB contains one complete FORTRAN statement, including any continuation lines but excluding the statement number, the comments fields, and any continuation characters. A zero word terminates the string in CHB.

RNXT must treat several statements as special cases during Pass 1, as described below. END and FORMAT statements and the CDIR$ compiler directives are such statements.

RNXT identifies the special CDIR$ compiler directive cards. The characters 'CDIR$' are stripped off each of these cards. Comment cards are not allowed in a CDIR$ continuation so that CDIR$ NOLIST can operate properly.

The FORTRAN standard prescribes special treatment for the END statement. An END statement cannot be continued to a subsequent line; therefore, RNXT does not look for a continuation line if the initial line encountered contains only the characters 'END'. No other statement can contain only the characters 'END' on its initial line. Identification of an END statement terminates the reading of any more lines for that program unit; thus, comment lines following an END statement are listed before the next program unit rather than after the END statement.

FORMAT statements are special cases because the asterisk character is allowed as a text delimiter and because commas are not required within FORMAT statements. Thus, the statement

```
FORMAT (I3 * I4) = (I5 * I6)
```

is ambiguous. A statement is identified as a FORMAT statement if it has a statement number and if it begins with the characters FORMAT(.

For statements other than FORMAT, RNXT makes a separate pass through CHB for the following reasons:

- To eliminate Hollerith/character constant notation, quote fields, and spaces not in a Hollerith/character field
- To convert any lowercase characters not in a Hollerith or character string to uppercase

For all statements, CFT indicates any character that is part of a Hollerith/character string by setting its sign bit. Setting of this bit means that Hollerith/character strings can be easily recognized and will not be confused with other characters. The high-order 41 bits are also set so that the character is negative in both A and S registers. A suffix character at the end of each Hollerith string indicates whether it is an H, L, or R type Hollerith constant. If no suffix is present, a C suffix is inserted into CHB to indicate a character constant.

The end of each statement is indicated by a zero word.

2.4 DETERMINE STATEMENT TYPE

After RNXT has assembled a statement into CHB, routine STTP (determine statement type) examines the statement and identifies it by type. First, it checks for the presence of DO in a replacement statement because an expression such as

```
DO 11 I= . . .
```

is potentially ambiguous when compared with a replacement statement. Then, depending on an address contained in the corresponding Statement Type Table entry, STTP transfers to the appropriate common syntax processor or unique statement processor.

Table 2-1 provides the Statement Type Table information as it appears at the front of the compiler.

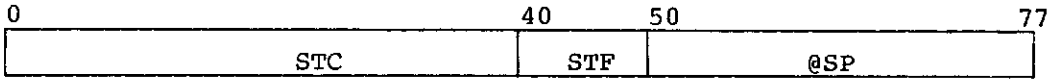
Table 2-1. Statement Type Table

GROUP	STATEMENT TYPE	
0	BLOCK DATA FUNCTION	PROGRAM SUBROUTINE
1	IMPLICIT	
2	CHARACTER COMMON COMPLEX DIMENSION DOUBLE DOUBLE PRECISION EQUIVALENCE	EXTERNAL GENERIC INTEGER INTRINSIC LOGICAL POINTER REAL
3	PARAMETER	
4	Statement function definition	

5	ASSIGN
	BACKSPACE
	BUFFER IN
	BUFFER OUT
	CALL
	CLOSE
	CONTINUE
	DECODE
	DO
	ELSE
	ENCODE
	ENDFILE
	ENDIF
	GO TO
		IF
		INQUIRE
		NAMELIST READ
		NAMELIST WRITE
		OPEN
		PAUSE
		PRINT
		PUNCH
		READ
		Replacement statement
		RETURN
		REWIND
		STOP
		WRITE

6	DATA	NAMELIST
7	ENTRY	FORMAT
8	END	

Each entry in the Statement Type Table has the following format:



Field	Bits	Description
STC	0-37	Initial 1 to 4 ASCII characters of statement; provides a unique statement type identifier. The content of this field is left-justified with zero fill. These identifiers are listed at the front of the compiler.
STF	40-47	Statement type flags
@SP	50-77	Address of appropriate statement processor

Because CFT specifies that statements must be in a certain order, types in the Statement Type Table are organized into groups according to these rules. For example, DIMENSION statements must precede all executable statements.

When a statement has been identified by type, the previous statement type is checked to ensure that the sequence is legal. Statements within a program must be ordered by group, (i.e., group numbers must be in increasing order), except for group 6 and group 7 statements, which can appear anywhere after group 0 statements. A group 0 statement need not be present. Figure 2-1 illustrates the required order of lines and statements in a CRAY-1 FORTRAN program.

2.5 STATEMENT PROCESSING

The compiler has nearly 50 *unique statement processors* that handle only one FORTRAN statement each. Additionally, there are *common syntax processors* that handle groups of statements. These handlers work on similar statements after they have been partially processed by unique statement processors.

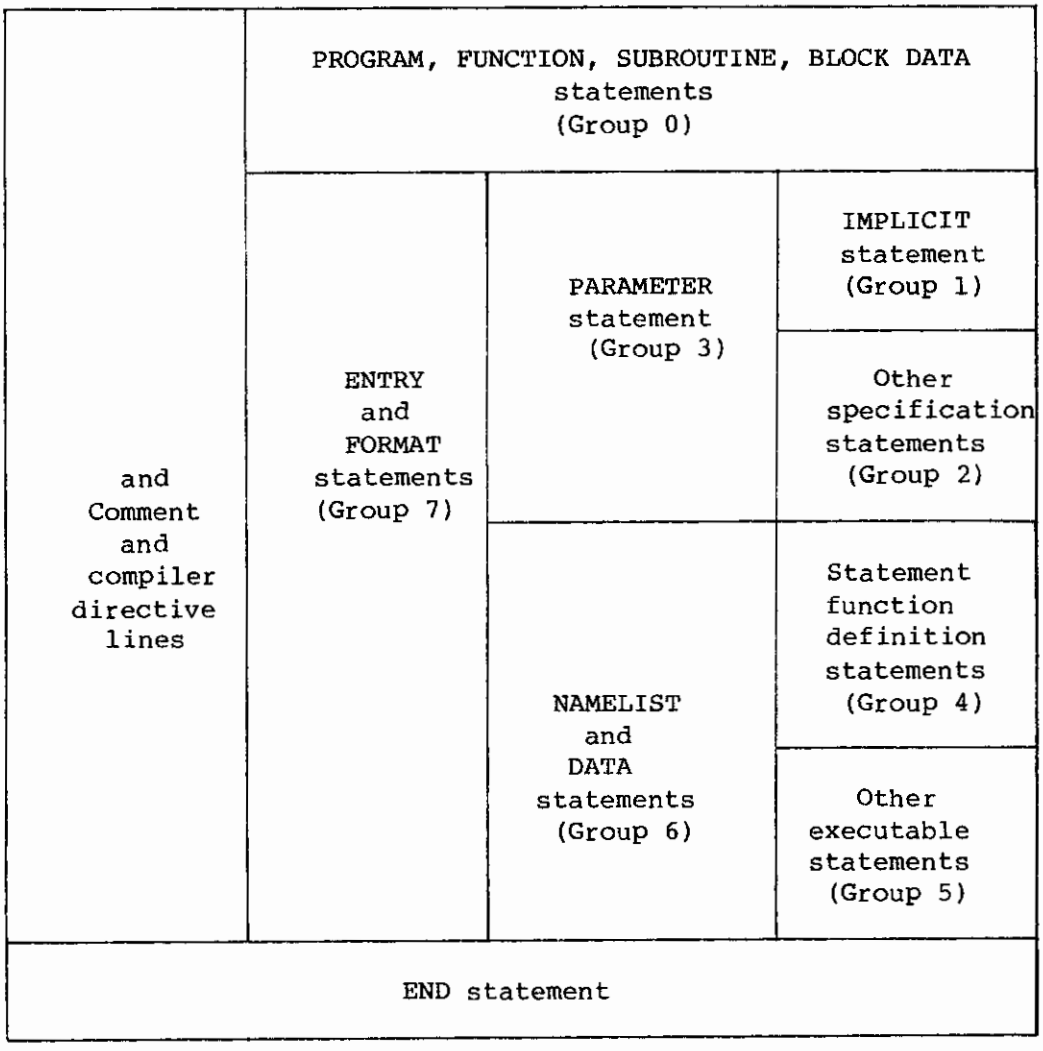


Figure 2-1. Required order of lines and statements

By convention, most of the statement processors have names of the form *xx*ST, where *xx* is a mnemonic for the statement or statement group handled by the processor. For example, WRST is the name of the unique statement processor that handles FORTRAN WRITE statements initially, while routine IOST handles final processing of the FORTRAN I/O operations statements as a group (including WRITE statements).

A major task of the statement processing section of the compiler is to reduce the number of different FORTRAN statements handled by CFT. Many FORTRAN statements provide the same result as one or more basic FORTRAN statements. For example, the statement

```
ENDFILE 10
```

serves the same purpose as (and therefore is replaced by):

```
CALL $EOFW (10)
```

Thus, a goal of Pass 1 of the compiler is to reduce the number of different FORTRAN statements in a program to only the most basic FORTRAN statements. Pass 2 of the compiler, therefore, needs very few different statement processors. By the end of Pass 1, the compiler has restructured the program unit so that it contains only five types of statements: replacement statements, CALLs, IFs, GO TOs, and ENTRY statement headers.

2.6 NON-EXECUTABLE STATEMENT PROCESSING

CRAY FORTRAN includes a number of non-executable statements. Non-executable statements contain I/O editing information, specify statement functions, classify program sections, and provide data specification. Non-executable statements and their processors are shown in table 2-2.

Once STTP determines that a statement is non-executable, control transfers to the appropriate unique statement processor. Unless the statement is a declarative, this unique statement processor completely processes the statement.

Declarative statements are handled in two steps. First, a declarative statement is handled by a unique statement processor, which handles items unique to that statement and then transfers control to a routine named DCLR (process declarative statement). DCLR is a group statement processor that handles list processing, taking variables and making entries for them in TBS and TBT. Additionally, DCLR makes TBA and TBQ entries for arrays.

EQUIVALENCE statements are processed differently from other non-executable statements. All EQUIVALENCE statements encountered in a program unit are packed eight characters per word in TBR. Then, after all other non-executable statements have been processed, the set of EQUIVALENCE statements is handled.

Table 2-2. Non-executable statement processors

Group	Non-executable statement	Statement processor(s)
0	BLOCK DATA FUNCTION PROGRAM SUBROUTINE	BDST FNST PGST SRST
1	IMPLICIT	IMST
2	CHARACTER COMMON COMPLEX DIMENSION DOUBLE DOUBLE PRECISION EQUIVALENCE EXTERNAL INTEGER INTRINSIC LOGICAL POINTER REAL	CHST, DCLR CMST, DCLR CPST, DCLR DMST, DCLR DBST, DCLR DBST, DCLR EQST EXST INST, DCLR INFN LGST, DCLR PTST, DCLR REST, DCLR
3	PARAMETER	PRST
4	Statement function definition	SFST
7	ENTRY	NTRY

DATA statements also are processed in two steps. First, as they are encountered, DATA statements are translated to a tag-operator sequence (refer to section 4 of this manual for more information) and entered in TBG. Then, during END statement processing, this information is processed into the Loader tables. Constants within DATA statements are handled differently from other constants. Normally, constants are entered in the Constant Table (TBB); however, constants within DATA statements are stored into TBE instead.

Dummy arguments in FUNCTION and SUBROUTINE statements are entered into TBS and are given a tag in TBT. This tag identifies an argument as a dummy argument, and the argument's type is determined from the Intrinsic Type Table maintained in register V7. When the remaining non-executable statements are processed, the arguments may become arrays or may have their types changed by typing statements.

Once a statement has been completely processed, control returns to RNXT for the next statement to be processed. The process loops until STTP determines that the next statement to be processed is an executable statement rather than another non-executable statement. When an executable statement is encountered, STTP branches to label EQ10 to finish processing the EQUIVALENCE statements.

The EQUIVALENCE statements are removed from the Block Relocation Table (TBR) and processed. The information gained in processing is then stored in the Dummy Argument Address Table (TBP) until the end of Pass 1, when all variables are assigned addresses.

2.7 EXECUTABLE STATEMENT PROCESSING

CFT converts executable statements to an intermediate form and puts them into the intermediate Tag Buffer (TGB). Each executable FORTRAN statement is converted to a sequence of tags, operators, and separators. Thus, at the end of Pass 1, TGB contains the entire executable string derived from the program unit; during Pass 2, the machine code is generated from this string.

A TGB statement entry begins with a statement header word. The header word is the only word of a statement entry having its sign bit set. A quick search through the sign bits in TGB can locate the beginning of each statement entry.

The statement entry header word cannot be completed until the entire statement entry has been built because some of the information kept in the header word, such as backward and forward indexes, are not known until further processing has taken place.

Following the header word for a statement entry is a 1-word entry for each element in the statement. This word may be a tag, an operator, or a separator.

A tag is derived from the Tag Table (TBT) entry for the corresponding symbol or statement number. Generally, the tag contains the following information:

- Tag type, a 3-octal-digit descriptor that identifies the function of an element in a statement (for example, external function name, statement number, constant, variable in a common block)
- Mode and particular attributes (such as call-by-value or function)
- Data type (such as integer, real) and data length (single-precision, double-precision, and so on)

It also contains an index into TBT for the corresponding entry.

Operators and separators are translated to 6-bit codes that reflect compiler processing precedence. The precedence scheme is detailed at the front of the compiler (and in the description of TGB in this publication). CFT shifts the code for an operator or separator to the leftmost 20 bits so that it sits in the exponent portion of the field. The rest of the word is all zeros for an operator; however, for a separator, parenthesis group type flags are set to indicate subscript or expression mode. Thus, the compiler can quickly spot which TGB entries are for tags (those with a tag value in bits 1-11) and which are for operators (those with zeros in bits 1-11).

The Tables section of this publication includes a detailed description of the TGB statement entry header and the types of statement entry words possible.

CFT automatically surrounds the right-hand side of a replacement statement with a pair of parentheses. The extra parentheses force the parser through the statement one more time to finish evaluation. In other words, the statement processor can terminate on a close parenthesis rather than looking for both a close parenthesis and an end-of-statement (the all-zeros word).

Two routines act as handlers for expressions: AT*xx* (assemble tag) and OP*xx* (assemble operator). Each unique statement processor uses these two routines to translate any expressions in the corresponding statement to TGB format. Each time the start of a new tag or operator is encountered, control is transferred to AT*xx*. AT*xx* then determines whether the next element is a symbol, a number, an operator, or a separator.

If the next element is a numeric constant, AT*xx* calls routine $NICV (numeric input conversion) to convert the number from ASCII to CRAY-1 internal format. If the next element is a character or Hollerith constant, routine EHOL packs the character or Hollerith constant. The converted constant is then entered into the Constant Table (TBB). AT*xx* also generates a tag for the constant and makes an entry in TGB.

If the next element encountered is a symbol, AT*xx* calls routine ACAN (assemble alphanumeric character group) to assemble the symbol name. A tag is generated for the symbol name and entered in TBT, if one does not already exist. Entries are also made in the Symbol Table (TBS) and in TGB.

If the next element encountered is an operator or a separator, OP*xx* converts it to a precedence code and enters it in TGB. (These precedence codes are listed in the description of TGB in the Compiler Tables section.) OP*xx* then returns control to AT*xx*. If AT*xx* detects an array, it branches to the section of OP*xx* that handles open parentheses and begins subscript evaluation.

OP*xx* sets flags in register S7, indicating that the statement is in subscript mode. The subscript is then expanded by AT*xx* and OP*xx* and entered in TGB.

OP*xx* maintains a parenthesis stack in registers S7 and V6. Each time an open parenthesis is encountered, the current parenthesis type (subscript, expression, argument list) is pushed from S7 onto V6 and the new parenthesis type is entered into S7. When a close parenthesis is found, V6 is popped back to S7. Thus, S7 always describes the current type of parenthesis being processed.

Finally, AT*xx* determines that the end of the statement has been reached and branches to terminate the statement processing (STTR).

Executable statements are of three types: those that accomplish input/output operations, those that provide program control, and those that handle data assignment. Because these types are handled differently by the compiler, they are discussed individually.

2.7.1 INPUT/OUTPUT OPERATIONS STATEMENTS

Each input/output unique statement processor performs initial functions specific to the statement. The I/O operations statement processors include:

Statement	Processor
READ	RDST
WRITE	WRST
PRINT	PIST
PUNCH	PUST
ENDFILE	EFST
REWIND	RWST
BACKSPACE	BKST
BUFFER IN	BFST
BUFFER OUT	BFST
ENCODE	ECST
DECODE	DCST
NAMELIST READ	NLST
NAMELIST WRITE	NLST

Each of these unique statement processors (except for EFST, RWST, and BKST) then calls routine IOST (process I/O statement) to process the list. IOST generates entries in TBS and TBT for variables and for external I/O library routine calls. The compiler converts all I/O operations statements to a series of CALL statements.

READ, WRITE, PRINT, and PUNCH statements involve three types of calls: an initial call, a data processing call, and a final call. The internal statement function facility is used extensively to compile library calling sequences. The processors for other statements generate direct calls to the corresponding I/O routines.

The DO statement processor, DOST, is called to process implied DO-loops in an I/O list.

2.7.2 PROGRAM CONTROL STATEMENTS

Each program control statement has a unique statement processor associated with it. These statements and their unique processors include:

Statement	Processor
CALL	CLST
RETURN	RTST
END IF	IEND
ELSE	IELS
GO TO	GTST
IF	IFST
PAUSE	PAST
STOP	STST
CONTINUE	CNST
DO	DOST

Each of these unique statement processors performs initial functions specific to the statement, including generation of particular test and required branch instructions. Program control statements are converted to IF statements. The same routines used for replacement statements process expressions in program control statements.

2.7.3 ASSIGNMENT STATEMENTS

Data assignment statements define variables and array elements during execution of a program. These statements and the unique statement processors associated with them in the compiler include:

Statement	Processor
Replacement statements	RPST
ASSIGN	ASST

RPST initializes for a replacement and then transfers to AT*xx* and OP*xx* to process subscripts on the left and the expressions on the right.

2.7.4 STATEMENT TERMINATION

Every executable statement terminates through routine STTR (statement terminate). STTR terminates processing for a particular statement.

If the statement is a statement function definition, STTR copies the definition from TBG to the Statement Function Skeleton Table (TBF), where all statement function skeletons are kept, and then resets TBG to empty. A statement function definition does not directly generate any code. The definition is copied from TBF to TBG whenever the statement function is actually referenced.

STTR examines statement numbers to check for DO-loop termination. If STTR detects loop termination, it first checks for a special case where the DO-loop consists of one statement. If the loop is a 1-line DO-loop, STTR calls the routine DORP to conditionally replace a 1-line DO-loop with a call to a SCILIB routine. If the replacement was successful and the statement number does not terminate additional DO-loops, STTR branches to RNXT to read the next statement. If the DO-loop is not a 1-line DO-loop, STTR branches to process the DO termination sequence.

Finally, STTR handles the actual insertion of backward and forward pointers in the statement entry header word. Normally, control then transfers to RNXT to read the next statement. However, if a statement such as a DO statement is expanding, STTP may branch to the main driver for that statement to handle the expansion processing.

2.7.5 INTRINSIC FUNCTION PROCESSING

The resolution of an intrinsic function name occurs in several locations between AT23 and AT28. The Intrinsic Function Name Table (TBJ) is searched for the intrinsic function name currently processing. If the intrinsic function name is found in TBJ, it is determined if the name refers to an intrinsic function or another construct (the NLI bit in the corresponding Intrinsic Function Attribute Table (TBK) entry). If the name refers to another construct, processing continues. If the name refers to a valid intrinsic function name, processing continues at AT50.

At AT50, an entry is made in the intrinsic stack INTST for the function reference. Each INTST entry contains three words: a function name in 8-bit ASCII, right-justified and zero-filled, the addresses of the last comma pointer and function tag in TGB, and the address of the TBK entry. All functions except MAX and MIN functions (those with a variable number of arguments) are treated as internal function macros by leaving a space before the function tag

for comma pointers. After storing the function in TBG and allowing space for the open parenthesis, the sign bit in S7 is set indicating intrinsic function processing and processing continues at OP71. Zero-argument functions have the parameter list inserted into TGB at AT55 and processing continues at OP90.

When the closing parenthesis of the function reference is located (indicated by the set sign bit of the parenthesis level), function name resolution is done by OP90 and the stack is popped. If the TBK entry indicates a special processing need, control transfers to the special processor. Special processing is done for CMPLX, CVMGT, MAX and MIN type functions, zero-argument functions, shift functions, and Boolean result functions, such as AND. Except for CMPLX and the MAX and MIN type functions, the REAL bit in TBK is set and is followed by the TBM index. Processing special cases is as close to processing general cases as possible.

In general processing, the arguments are counted (the counter is incremented at each comma tag) and each argument type is compared to the first argument type (the type is found in the comma tag). If all argument types are not alike, an error message is output. If the number of arguments differs from the function number in the TBK entry, an error message is output. The argument type is used as the index into a computed $GOTO transferring control to the appropriate type processor. The type processor checks the flag for the argument type. If the flag is not set, an error message is output, otherwise the Library Macro Table (TBM) is extracted from the TBK and control is transferred to OP114. At OP114, the specifier name is found in TBM and inserted into the Symbol Table (TBS). If the generic name is referenced, an ! is added to the specific name entered into TBS. The specific name can now be used as a name other than an intrinsic function name. The ! is removed from the TBS name at EN08. The function tag is found in TBM and updates the Tag Table (TBT).

If the function tag is an internal function tag, control is transferred to OP81 for expansion. Function tags and parameters, except MAX or MIN type functions, are moved over the comma pointers. S7 is adjusted accordingly and parsing continues at AT03.

2.8 END PROCESSING

When the compiler finds an END statement, Pass 1 ends. Routine ENST (process END statement) is called to handle the END statement.

Unless the previous statement was an unconditional transfer, ENST generates a call to $END in a main program or a RETURN in a subprogram. (There is no generation if the previous statement was an unconditional transfer.) Once the END of the program unit is

encountered, variables are assigned addresses. By the end of Pass 1, all variables are known. Variables occurring in EQUIVALENCE statements have already been given addresses by this time, since they are handled at the end of nonexecutable statement processing. Local variables, however, must be assigned addresses. ENST resolves all EQUIVALENCEs and assigns addresses to all variables in TBT.

DATA statement entries are also made in TBB, the Loader's Text Table. Then ENST begins building TBH, the Loader Program Description Table. TBH holds program name, common block name and length information. Following this are ENTRY names and EXTERNAL names.

Each statement having a statement number also has a pointer in bits 1 through 17 of the statement header entry in TGB that points to the corresponding TBT statement number entry. At the end of Pass 1, all statement number references are linked together, and unreferenced statement numbers are deleted. The index to the TBT entry for an unreferenced statement number is cleared from the TGB statement header entry. Although the statement number still exists in TBS and TBT, no pointers to these table entries exist. An unreferenced statement number is transparent to CFT Pass 2.

ENST copies intermediate code generated as a result of variable dimension declarators from TBQ to immediately after each ENTRY statement header. ENST also links all statement numbers and references. Any statement number having no reference is deleted. If a statement number is never referenced, the pointer to TBT in the statement entry header word is cleared and, although the statement number is still in TBT, there is no pointer to it. This means statement numbers cannot be used only to break up blocks of code. Finally, the Intermediate Tag Buffer (TGB) is moved to the Tag Buffer Table (TBG) and actual addresses relative to the appropriate block are filled in for all symbols. The move is done word-for-word. As each item is moved, its tag is examined and looked up in TBT and the actual address or offset is put into the tag.

At EN81, some of the tables are cleared and released in preparation for Pass 2.

PASS 2 FLOW 3

3.1 INTRODUCTION

Basically, Pass 2 of the CRAY-1 FORTRAN compiler performs the following functions:

- Initialization of loader tables
- Processing of code blocks one at a time:
  - Locate and analyze code block
  - Generate intermediate code
  - Schedule instructions
  - Assign registers
  - Generate loader tables
  - Optionally produce generated code listing
- END processing for Pass 2
  - Link actual statement number addresses in generated code
  - Put B and T register counts in EXIT/ENTRY sequences
  - Produce Symbol Table (TBS) with cross-references on the output file after each program unit (if called for)
  - Produce Debug Symbol Table (if called for)
  - Write loader tables out on binary output dataset

The primary input to Pass 2 is the information contained in the Tag Buffer Table (TBG). TBG is the translation of the original program into intermediate text made during Pass 1. Auxiliary tables produced during Pass 1 and used as input to Pass 2 include TBS, the Tag Table (TBT), and the Array Table (TBA).

The principal output from Pass 2 is the compiled FORTRAN program in relocatable binary form. Secondary output from Pass 2 includes a printable listing of the generated code, including TBS with the cross-reference lists, if called for.

3.2 LOCATE AND ANALYZE CODE BLOCK

Analysis of a code block is the key to optimization for CFT. The code block is the basic optimizing unit that CFT handles. Once a code block is defined, block analysis includes the following functions:

- Mark constant increment integers
- Analyze array references for dependencies
- Promote constants within subscript expressions
- Examine array and function references

Once block analysis is complete, actual code generation can occur.

3.2.1 DEFINE NEXT CODE BLOCK TO BE PROCESSED

Routine ABLK (analyze block) is the main driver for Pass 2. During this pass, the Tag Buffer string in TBG is processed one code block at a time. The first section of ABLK prepares for and then defines the next block to be processed.

Actual code is generated in PIB, the Pseudo Instruction Buffer (see Appendix E for a detailed description of PIB). PIB is allocated and released on a block-by-block basis as Pass 2 progresses, taking up the memory area occupied by the Character Buffer Table (CHB) and TGB during Pass 1 processing. The Constant Table (TBB) expands as the binary is constructed.

ABLK initializes the following tables by setting their LWA+1 equal to their FWA: the Block Definition Table (TBBK), the Label Usage Table (TBLB), the Triad Table (TBW), the Variable Reference Table (TBX), the Dependent Reference Table (TBY), and the Defined Variable Table (TBZ). These tables can be marked empty because they are built for each code block during Pass 2.

ABLK searches through the statements in TBG building a TBBK for each block. The block is delimited by a loop boundary, ENTRY statement, subprogram begin, subprogram end, or an active label with a reference outside the block.

Statement numbers encountered during Pass 2 are not limited to those that were included in the original source code. During Pass 1 processing, CFT inserts made-up statement numbers for processing logical IF and block IF statements when it is necessary to jump from a block. CFT also inserts made-up statement numbers at both the beginning and the end of a DO-loop when processing a DO. A made-up statement number is also inserted immediately following an ENTRY other than the primary. These made-up statement numbers go through Pass 2 analysis the same as any programmer-defined statement numbers.

3.2.2 MARK CONSTANT INCREMENT INTEGERS

The section of code beginning at AB20 builds TBZ. TBZ contains an entry for each variable defined within the code block. The information in TBZ is used later in Pass 2 in handling constant increment integers and in building the Cross Reference Overflow Table (TBV).

Starting at AB40 is a section of code that finds and marks all constant increment integers in TBG. In loop mode, a *constant increment integer* (CII) is an integer that is incremented by an invariant expression at only one point in the loop and is not conditionally executed in the block. An example of a CII is the DO control variable in a DO block where the index is an integer. CIIs are located for two reasons: subscript optimization in general and vectorization in particular.

In a replacement statement, a CII can be either a function of itself (for example, I=I+1) or a function of another integer that is a CII (for example, I=J+1, where J is a CII). The only operators allowed in a CII expression are + and -. One operand may be variant, but all others must be invariant within the loop.

When a CII variable is identified, a flag is set in the TBZ entry for the variable. All references in TBG made to that variable also have a flag set to indicate that that variable is a CII.

3.2.3 ANALYZE ARRAY REFERENCES FOR DEPENDENCIES

Routine ADEP (analyze dependencies) checks for dependencies within arrays. Vectorization is inhibited if a dependency exists; however, the programmer can override this with the CDIR$ IVDEP directive.

ADEP builds the Plus Dependency Table (TBPD). TBPD is used in code generation to move a vector load before a vector store.

ADEP is a double loop. The outer loop, which drives ADEP, takes each successive definition entry from the Defined Variable Table (TBZ). A definition entry is one in which the defined item appears on the left-hand side of a replacement statement, or the item is used in an input statement or is an argument in a subroutine or function that might have side effects. For each definition entry, the inner loop of the routine searches the entire block for references to the item defined. ADEP compares the definition with subsequent definitions and other references made to it in the block, looking for dependencies (EQUIVALENCE overlapping, for example). If ADEP finds an ambiguous dependency with a condition for safe vectorization, the condition is entered into the Conjunctive Term Table (TBCT). If ADEP finds an unambiguous dependency or an ambiguous dependency with no condition for safe vectorization, the Vector Loop flag (VLF) is turned off. The VLF is global to a block and is located in register S7.

ADEP builds TBY, which is used by the optimizer in load-and-store operations. Each variable within a loop has a definition entry followed by an entry for each reference made to the variable. ADEP proceeds through the entire block, even if not in loop mode or if a dependency has been found, because TBY must be completely built for the instruction scheduler and the load/store generation routines.

3.2.4 PROMOTE CONSTANTS WITHIN SUBSCRIPT EXPRESSIONS

Basically, routine PCON (promote constants) has as its task the cleanup of all subscripts within the code block. PCON cycles through the code block looking for array references. Each time PCON finds an array reference, it cycles through the reference looking for constants within subscripts. Each constant occurring within a subscript is multiplied by the appropriate dimension multiplier and then added into the initial term of a subscript expression where it acts like a bias.

For example, if the dimension is A(10,10,10), the subscript

```
A(I,3,J+2)
```

is processed as follows:

Pass 1 expands the subscript to

```
@ A + 0 + 1 * (I-1) + 10 * (2) + 100 * (J+1).
```

PCON extracts the constants, leaving

```
@ A + 119 + 1 * (I) + 100 * (J).
```

As a result of PCON, an array reference looks like a base address plus a constant plus terms that involve variables within the subroutine for the rest of the subscript. The constant is the sum of all constants from all subscripts. PCON calls routine SVEC to locate possible scalar temporary definitions in a vector loop. SVEC then sets the vector array flag (VAF) in subsequent references to the scalar temporary within the TBG block.

3.2.5 EXAMINE ARRAY REFERENCES AND FUNCTION REFERENCES

Array and function references are examined by routine EAFR. For each statement in a block, EAFR does a backward scan looking for array and function references. Within a statement, it scans from right to left so that it can sort out the parentheses within the statement.

Array references are checked by EAFR to determine whether they are vectorizable. An array element is a candidate for vectorization if its subscripts meet the following general rules:

1. All subscripts within a loop must be invariant within that loop except one.

2. The one variant subscript can contain only one linear reference to a CII variable.

3. The only operators allowed in the variant subscript are +, -, and * on either side of the CII; otherwise, +, -, *, /, and ** are allowed.

If all of these conditions are met, then EAFR sets a flag called the vector array flag (VAF) for this particular array reference. This flag is set on a term-by-term basis.

EAFR also sets the variant subscript flag (DSF) in the array tag if the subscript has any variants. A vectorizable array has this flag set, but a non-vectorizable array might also have a variant subscript. A subscript is *invariant* if it is not changed within a loop and there are no stores anywhere into the array. A subscript is *variant* if it is changed within the loop or if there is any store into the array.

EAFR looks at function references to ensure all arguments are proper vector arguments. It checks a flag in the function tag for a vector version available for the function. If a vector version exists, EAFR looks at each argument of the function to see if it is invariant and to see if its VAF is set. If either condition is met, then the function reference is vectorizable.

3.2.6 TRANSFER TO VECTOR CONTROL

If processing is in loop mode, control is transferred next to routine VCTL (vector control). VCTL consists of three main sections.

The first section of VCTL copies several flags including the VAF from TBG into TBZ.

The second section of routine VCTL searches each tag in order. Whenever VCTL finds a condition that turns off vectorization, it turns off the vector loop mode flag (VLF) and returns to the Compile Block routine (CBLK). If the tag is that of a variant (the variant bit is set from ABLK), then it is a vector.

A scalar temporary, even though it is not an array, has had its VAF set by ADEP; VCTL thus treats it as an array. VCTL also looks for recursive sums and checks to ensure that at least one array reference appears in the loop.

When VCTL is done, it generates the necessary calculations at the beginning of the loop to set the vector length register and then returns to CBLK. CBLK generates vector instructions because the VLF is set.

The third part of VCTL generates the incrementation for the CIIs found in the code block. It is called when CBLK finishes the loop. The information in TBZ is used for this purpose.

3.3 GENERATE INTERMEDIATE CODE

As a result of Pass 1, the number of different FORTRAN statements in a program is reduced to very few. By the end of the pass, CFT has restructured the program unit so that it contains the following types of statements: replacement statements, CALLs, IFs, GO TOs, and ENTRY statement headers.

Generating intermediate code is driven by routine CBLK (compile block). CBLK handles CALL statements and replacement statements. IF statements are handled by the code beginning at IF50 and GO TO statements cause control to transfer to GT30; however, this transfer does not occur until after CBLK generates the code for all expressions associated with the statements.

All expressions are handled by CBLK as general-purpose expressions even if they are basic 1-term expressions (for example, each argument of a CALL statement or the expression in the parentheses in an IF statement).

CBLK calls PBLK to find the next statement to be compiled. CBLK tries to compile the located statement by finding the innermost set of parentheses. The innermost set of parentheses is compiled, the parentheses are removed, and the process is repeated until all parentheses have been removed. Final statement processing occurs when the parentheses processing ends. Then, any remaining stores are compiled and IF statements or GO TO statements are completed.

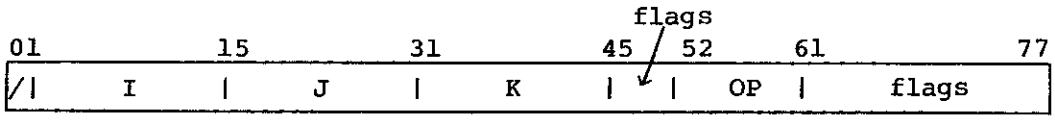
When CBLK finds an open parenthesis, it begins a series of forward scans within the parentheses. By definition, the first open parenthesis found in a backward scan is the innermost. CBLK puts itself in a loop and calls routine OLEV (operator level), which makes repeated scans through the expression, looking for operators in precedence order.

CBLK calls PTRI (process triad) for each operator OLEV finds in the expression. PTRI checks for index processing, and either calls CTRI or extracts the index increment.

When CBLK determines it is processing an intrinsic function reference, it evaluates each of the function parameters and calls routine INFN (intrinsic function generator) to expand the skeleton for the function and generate instructions. INFN returns to CBLK.

Intermediate code is generated one word at a time in TGB, the Tag Buffer. Most of the code inserted in TGB is generated by routine CTRI. As each instruction is generated, one word is stored into TGB. Register A7 contains the address where the next instruction can be stored; A7 is incremented by 1 after each store to prepare for the next instruction.

The format of a stored instruction is:



Field	Bits	Description
I	1-14	I field of instruction, pseudo register number, or parameter number
J	15-30	J field of instruction, pseudo register number, or parameter number
K	31-44	K field of instruction, pseudo register number, or parameter number
Flags:	45-50	
	45	If set, I field is result or is unused; if clear, I field is operand pseudo register or constant.
	46	If set, J field is result or is unused; if clear, J field is operand pseudo register or constant.
	47	If set, K field is result or is unused; if clear, K field is operand pseudo register or constant.
	50	If set, I is both an operand and a result (as in shift operations)
OP	52-60	Opcode for instruction to be generated

Two special cases exist for the FLG and OP fields. If FLG bits 45, 46, and 47 are set and OP=005, the entry represents an entry or exit sequence. If FLG=0, OP=0, and I=0, the entry represents a special-case instruction sequence.

Flags:	61-77	
	75	If set, instruction can be delayed to postamble
	77	If set, J and K are invariants and instruction can be removed to preamble

When instructions are generated, pseudo registers are assigned instead of real registers. A *pseudo register* is an imaginary register not corresponding to any *hard (real) register*. Pseudo

register assignment begins at register 7375 and goes downward; pseudo registers 7400 and above are special-purpose registers reserved for use by the compiler. For example, pseudo registers 7470-7477 represent V0-V7. The lowest numbered pseudo register assigned is 100. Approximately 4000 pseudo registers are available in each block. Generally, individual pseudo registers are not used more than once in a block.

3.4 SCHEDULING

CBLK transfers to SKED (instruction scheduler) after compiling the last statement in a block. SKED separates the preamble, main body, and postamble instructions and returns to ABLK. If more blocks can be added to the compiled blocks, ABLK transfers to CBLK to compile the next block. If code cannot be added to the current block, ABLK transfers to SKED to schedule the compiled code.

SKED separates the block of code into groups. The groups are separated by statement numbers, return jump instructions, entries, or exits. Each group is scheduled separately with the backup registers holding values which need to be passed between instruction groups.

Processing an instruction group begins with a backward pass to assign times ignoring the limitations of the real machine. The times given by the first pass are used to assign instruction issue priorities. When all the priorities are assigned, a final pass is made to assign registers and issue the instructions in the correct order. This final pass takes the hardware limitations into account when issuing instructions.

The final pass first processes statement numbers or return jumps at the start of a group, then takes the highest priority instruction which is ready to issue, assigns registers, and issues it. The issue process normally ensures an operand is in the register when the instruction using it comes up for issue. If an operand is not in the register, the value is reloaded from a backup register or the constant definition is reissued. If there is no register free for the result register, a register is made free by transmitting the lowest priority register to a backup register.

During the final pass, memory transfers are examined for overlap. If an overlap occurs and the machine for which code is being generated can get the operations out of sequence, code is inserted to force proper ordering.

If an instruction is partially processed and cannot be issued, processing is undone and the next eligible instruction is selected for issue.

Special processing occurs when the group processing contains a complete loop. Constant definitions and eligible instructions are moved to the preamble and reissued at the end of the loop. Recursive register definitions are assigned to a register for the life of the loop.

At the end of a group, remaining transfers to specific registers are compiled. These transfers may not need compiling because register assignment attempts to assign a result which is later transmitted to a specific register.

Routine RASN (register assignment) maintains a mapping between the CRAY-1 system's hard registers and the assigned pseudo registers. The mapping is:

- The A registers and their backup registers, the B registers;
- The S registers and their backups, the T registers; and
- The V registers and their backups, the W registers.

The W registers do not really exist; they are included only to preserve symmetry. A reference to a W register is actually taken as a reference to memory.

RASN cannot assign all B and T registers. For example, by convention B70-77 and T70-77 are used as scratch registers by library routines, while several other B registers are dedicated to the calling sequence and stack support. The first B register RASN can assign is given by the value in memory word BFIRST, and the last B and T registers RASN can assign are coded in memory word BTLIM.

RASN produces the Final Instruction Buffer (FIB). RASN and SKED reformat the assigned hardware registers entries into the FIB format (see Appendix E for a detailed description of FIB). FIB holds the generated code with assigned hardware registers. Instructions and statement numbers are unpacked and stored one to a word. FIB is packed into the relocatable binary table by LTGN.

3.5 GENERATE LOADER TABLES

The three parts of routine LT (loader table generation) handle all aspects of building the loader tables. The code beginning at LTST is called at the start of Pass 2 initializing the loader tables. LTGN takes a block of instructions after RASN and packs up the code into the loader tables. LTND does the final processing; it does some statement number processing for the loader tables after the entire program unit is compiled, then it closes the tables.

LTGN moves through a code block one instruction at a time. It determines if an instruction is 16 or 32 bits and packs the instructions into the Text Table (TBB). The 32-bit instruction requires care in packing so it is correctly broken up across a word boundary in TBB. LTGN makes other table entries necessary for a given instruction, such as making an entry in the External Reference Table (TBE) if the instruction refers to an external or making an entry in the Block Relocation Table (TBR) if it is relocatable. Statement label references and ENTRY/EXITs are linked for processing by LTND after all addresses are known.

Each time LTGN encounters a statement number definition, that is, the location where a statement is labeled with that statement number, it puts the address of the statement number in a table. LTGN maintains a linked list through the code in TBB of all references to a statement number.

LTGN is also the driver for the routine OUT. OUT produces the generated code and/or the pseudo-CAL code for the program.

Control then returns to ABLK to access the next code block. This loop is repeated until an END statement is encountered.

3.6 END PROCESSING

When LTND is called, TBR and TBB are closed, that is, the size of each table is determined and the header word is built for each.

LTND cycles through TBB, the generated code, and finds all statement number references. Before end processing, all references to a given statement number are linked. LTND goes through this link and fills in each link term with the actual relative location of the corresponding statement number.

LTND also determines how many B and T registers the program unit uses. It cycles through all the ENTRY and EXIT sequences LTGN linked together and adjusts the code, inserting the exact number of B and T registers into the instructions saving B and T registers.

After LTND finishes the loader tables, it writes them out to the binary file. The Debug Symbol Table is created and written to the binary output file if requested (ON=Z on the CFT control statement) at this point. Then, if called for, the following tables are formatted and listed on the output: the DO-loop Table (TBD; ON=D), the common block table (ON=C), the statement number table (TBS and TBT; ON=T or X) and the Symbol Table (TBS and TBT; ON=T or X).

This completes compilation of a program unit. Unless end of file is encountered, control returns to BG10 to begin Pass 1 for the next program unit. When end of file is detected, control transfers to EN78 to terminate compilation.

COMPILER TABLES 4

4.1 INTRODUCTION

CFT constructs a number of tables in the user area. As compilation proceeds and tables are built, the table area grows downward in memory.

To minimize storage moves during compilation, those tables not changing greatly in size during processing are located near the top of the table area, while those expanding significantly are positioned near the bottom of the area. For example, the Symbol and Tag Tables (TBS and TBT) are located near the bottom of the table area, while the skeleton for DIMENSION statements is near the top, because it is built early in compilation and does not change thereafter.

Four tables are exceptions to the above: the Intrinsic Function Name Table (TBJ), the Intrinsic Function Attribute Table (TBK), the External Library Tag Table (TBL), and the Library Macro Table (TBM). These tables are assembled with the compiler and do not expand in size during program compilation. TBJ, TBK, TBL, and TBM are not true compiler tables but are treated as true compiler tables because the mechanisms for table searches and general table handling can then be used on these tables. TBJ, TBK, TBL, and TBM are stored immediately following the compiler's executable code.

Figure 4-1 illustrates the relative locations of the compiler tables in the user area.

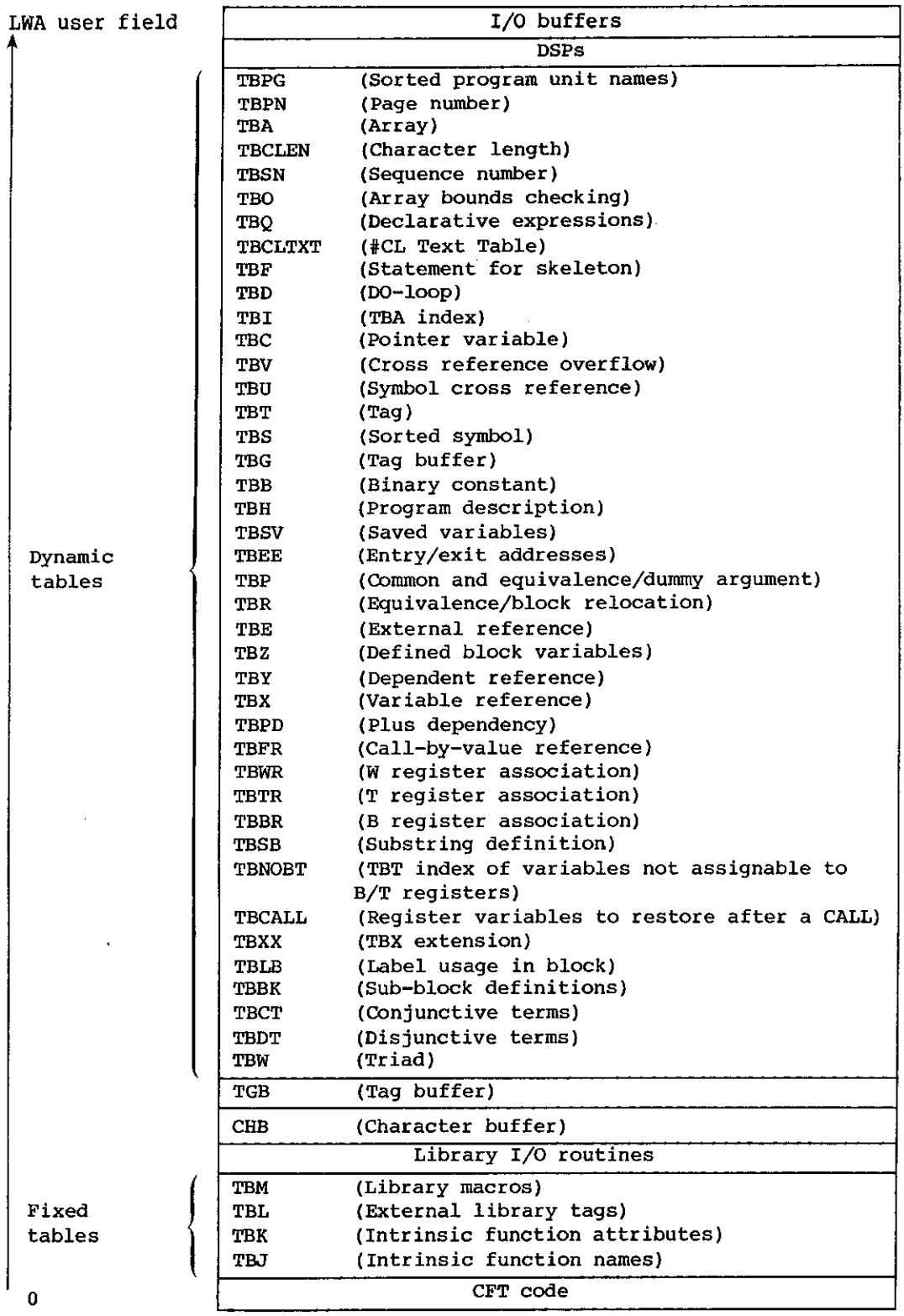
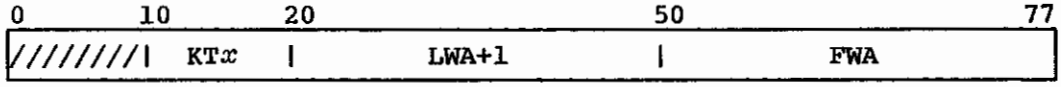


Figure 4-1. Compiler table memory locations

4.2 TABLE MANAGEMENT

Part of register V7 contains CFT table parameter words (see Appendix B for a detailed description of register V7). Each table maintained by CFT has one word of V7 set aside as a pointer. The format of this word follows.



Field	Bits	Description
KTx	10-17	Table pointer; index to the corresponding table in V7.
LWA+1	20-47	Last word address + 1 of the table; LWA+1 is used so the difference between it and FWA equals the true table length.
FWA	50-77	First word address of the table

KTx is not used during compilation and is in V7 only for use by the postmortem debugger, CFTDUMP.

Initially, all the table pointers for the dynamic tables point to the top word of memory and all the tables are empty. For each table, LWA+1=FWA, which indicates a null table. As compilation proceeds, the tables grow in size, with memory allocated as needed. Each time there is a memory move, all table parameter words are adjusted accordingly. In Pass 2, many tables are emptied and reused from code block to code block.

Most compiler tables are sequential tables. A *sequential table* has entries made to it as the entries are encountered; new entries are made at LWA+1.

TBS and TBPN are maintained in sorted order. A *sorted table* requires more time to enter a new element, but it can be searched rapidly because every element does not have to be examined. A sequential table is easy to add entries to, but searches must examine every element until a match is found.

4.2.1 SEQUENTIAL TABLE MANAGEMENT

The FWA for most sequential tables is a multiple of 100, because space for sequential tables is allocated 100 words at a time.

All entries to sequential tables are made by the routine ELWD (enter last word). Each time an entry is made to a sequential table, it is checked to ensure that the table did not overflow the adjacent table; that is, LWA+1 is equal to FWA of the next table.

When entering ELWD, register A1 contains the index into V7 of the appropriate table pointer word enabling examination of both the current and adjacent table pointer words, causing difficulty if the table pointer words were not stored in a vector register. The value to be entered into the table is in register S4.

When exiting from ELWD, register S3 contains the updated current table pointer word, register A4 contains the index to the new table entry, and register A3 contains the actual address of the new table entry.

If a table is full, ELWD calls routine MTAB (move table) to allocate another 100 words for the table. MTAB searches through the table pointer words in V7 looking for a gap (a table whose size has been previously decreased). The search proceeds from high memory (TBA) through low memory. If a gap is found, tables between the table needing space and the gap are moved 100 words into the gap. If no gap exists, all lower tables are moved down 100 words.

If the table area must grow downward, a check is made to see if the table is within 4000 words of the top of TGB/PIB. If it is, 20000 more words are requested from the system by GMEM. 4000 words is an arbitrary pool and an estimate allowing TGB/PIB to grow one word at a time and probably not overflow the tables. While a statement is being processed, no checks are made for table overflow when entries are made in TGB/PIB. If MTAB does a memory move, it adjusts all of the table pointers in V7.

The actual memory move is done by the routine MMEM (move memory). When entering MMEM, register A1 contains the address of the source FROM, register A2 contains the destination address TO, and register A3 contains a count of the number of words to move. MMEM is a general memory management routine that moves up or down and handles overlapping of the FROM and TO area. The actual move uses vector loads and stores.

A sequential table can be released or collapsed by setting its LWA+1 equal to its FWA. If another table has to be expanded later, ELWD and MTAB use this space.

Ten routines can be used to search sequential tables. The four primary table search routines are:

- SBLT - Search backward, shift left, table
- SBRT - Search backward, shift right, table
- SFLT - Search forward, shift left, table
- SFRT - Search forward, shift right, table

These routines search tables in the forward or backward direction. The backward search is important because often the last reference to an item is needed rather than the first. These four routines act as drivers for four general-purpose search routines that can be used to search any area of memory. These four are:

- SBLS - Search backward, shift left, string
- SBRS - Search backward, shift right, string
- SFLS - Search forward, shift left, string
- SFRS - Search forward, shift right, string

A normal search does the following steps:

1. Loads up a vector of table elements.
2. Optionally, shifts them right or left to shift off the bits not going to be search targets.
3. Subtracts the search target from the vector.
4. Generates a vector mask.
5. If a match is not found, loops until table is empty.

This procedure allows everything to chain together; thus, it takes about 1 clock period per element. Most tables are organized so likely search targets are entered in the uppermost or lowermost portion of a word. This allows shifting instead of masking to obtain the search targets.

When it is not possible to find the search target using a shift operation, a masked search occurs and proceeds as follows:

1. Loads up a vector of table elements.
2. Subtracts the search target from the vector.
3. Masks to get at the middle bits of the word.
4. Generates a vector mask.
5. If a match is not found, loops until table is empty.

Because this uses the logical unit both for the masking and formation of the vector mask, this is a 2-clock-period operation per element.

Two general-purpose masked search routines are available:

- SBMS - Search backward, masked, string
- SFMS - Search forward, masked, string

No masked table search routines exist.

4.2.2 SORTED TABLE MANAGEMENT

Entries in the Symbol Table (TBS) and the Program Unit Name Table (TBPN) are arranged alphabetically. Thus, every time an entry is made, the table is rearranged, inserting the new entry in alphabetical order.

Initially, the largest possible positive value ($2^{63}-1$) is entered in the table. Subsequent entries are made above or below the initial entry, depending on whether they come before or after the initial entry alphabetically. Thus, no more than half of the table need be moved to make any one entry. On the average, no more than one-fourth is moved.

The FWAs of these tables are not, in general, multiples of 100 words. Initially, FWA points to the first element inserted in the table. Each time a symbol name or a program unit name is encountered, the compiler searches TBS or TBPN, respectively. If the name is found, the compiler uses the information provided in the corresponding entry. However, if the name is new, the compiler locates the slot where the new element must be inserted; that is, it finds the two elements that the new entry belongs between, and then makes the entry.

Entries in TBS and TBPN are in 6-bit ASCII, packed eight characters per word. At the end of the entry is a pointer to the corresponding entry in TBT or TBPG, respectively. SSTB converts from 8-bit to 6-bit ASCII before searching either table.

Routine SSTB (search sorted table) bases its search techniques on the fact that a sorted table is in alphabetical, and thus numerical, order. If the table has 64 or fewer elements, SSTB effectively performs a normal sequential search. However, if there are more than 64 elements, SSTB loads every 63rd element into a vector register. This procedure ensures that there are no memory bank conflicts on the vector read. By performing a search along this vector, it is possible to determine which block of 63 the match is in. This block is then searched sequentially.

SSTB follows a general-purpose algorithm. SSTB searches tables longer than 64 x 64 words by first loading up every 64 x 63rd word and determining the block where the target is. This block is then searched in multiples of 63. This search allows up to a 256,000-word table, although FORTRAN symbol tables are typically smaller. If the element is in the table, SSTB finds the match; if it is not, the search finds the spot in the table where the element should be entered.

Routine ESTB (enter sorted table) makes the entry. It first checks if the new entry will be made in the upper or the lower half of the table. If the addition is to be made in the upper half, entries above the addition are moved up; if it is in the lower half, entries below it are moved down. ESTB also checks to see if additional space is needed before making the entry. TBS, TBG, and TBPN expand in either direction; ESTB does a memory move, moving everything one word above or below the appropriate slot and inserting the new entry.

4.3 TABLE DESCRIPTIONS

The compiler tables listed in table 4-1 are described in the following section. Table 4-1 contains the table prefix, table name, and the register numbers (in octal) of the associated table parameter word in register V7 for CFT compiler versions 1.11, 1.13, and 1.14 calling sequences, respectively. The table descriptions in the following section are arranged alphabetically according to table prefixes. Pages are also numbered according to the table prefixes. Tables with different functions in Pass 1 and Pass 2 have their dual role descriptions separated by a slash.

Table 4-1. Table descriptions

Prefix	1.11 Version Register Number	1.13 Version Register Number	1.14 Version Register Number	Table name
TBA	44	51	57	Array Table
TBB	27	33	40	Constant Binary Table
TBBK	None	None	16	Block Definition Table
TBBR	7	16	23	B-register Associates Table
TBC	35	41	46	Pointer Variable Table
TBCALL	None	14	21	Register Variables to Restore After a CALL Table
TBCLEN	None	None	56	Character Length Table
TBCLTXT	41	45	52	#CL Text Table
TBCT	None	None	15	Conjunctive Term Table
TBD	37	43	50	DO-loop Table
TBDT	None	None	14	Disjunctive Term Table
TBE	23	26	33	External Reference Table
TBEE	None	31	36	Entry/Exit Address Table
TBF	40	44	51	Statement Function Skeleton Table
TBFR	None	21	26	Call-by-value Reference Table
TBG	30	34	41	Tag Buffer Table
TBH	26	32	37	Program Description Table
TBI	36	42	47	TBA Index Table
TBJ	12	7	7	Intrinsic Function Name Table
TBK	13	10	10	Intrinsic Function Attribute Table
TBL	14	11	11	External Library Tag Table
TBLB	None	None	17	Label Usage Table
TBM	15	12	12	Library Macro Table

Table 4-1. Table descriptions (continued)

Prefix	1.11 Version Register Number	1.13 Version Register Number	1.14 Version Register Number	Table name
TBNOBT	None	None	21	TBT Index of Variables Not Assignable to B/T Register Table
TBO	43	47	54	Array Bounds Checking Table
TBP	25	30	35	Common and Equivalence / Dummy Argument Address Table
TBPD	None	22	27	Plus Dependency Table
TBPG	46	53	61	Page Number Table
TBPN	45	52	60	Program Unit Name Table
TBQ	42	46	53	Variable Declarator Table
TBR	24	27	34	Packed EQUIVALENCE / Block Relocation Table
TBS	31	35	42	Symbol Table
TBSB	17	15	22	Substring Definition Table
TBSN	45	50	55	Sequence Number Table
TBSV	26	31	36	Saved Variable Table
TBT	32	36	43	Tag Table
TBTR	10	17	24	T-register Associates Table
TBU	33	37	44	Symbol Cross Reference Table
TBV	34	40	45	Cross Reference Overflow Table
TBW	16	13	13	Triad Table
TBWR	11	20	25	W-register Associates Table
TBX	20	23	30	Variable Reference Table
TBXX	None	None	20	TBX Extension Table
TBY	21	24	31	Dependent Reference Table
TBZ	22	25	32	Defined Variable Table
TSIFS	None	None	None	Special Intrinsic Function Table

4.3.1 NOTATIONAL CONVENTIONS

Several conventions are followed in the presentation of table information. Unless otherwise noted, the following conventions are observed throughout the section.

When the field value is known, the value is given in the field description (in octal). The format for the field description is:

Field description; *xxxx*

where *xxxx* is the content of the field being described.

If a field contains an index to another table, the field is given the abbreviation %TB*x*. An address in another table is abbreviated @TB*x*.

If a field contains zeros, the figure shows diagonal lines for that field.

The bit positions are specified in octal.

4.3.2 TAG DEFINITIONS

Each entity (variable, statement number, external name, etc.) within a statement is converted to a descriptive tag during Pass 1 processing. The first three octal digits (9 bits) of a tag describe the basic type of entity.

Tags are used in many compiler tables. Table 4-2 describes tag types.

Table 4-2. TGB tag descriptions

Tag	Description
100	Pseudo tag. Used primarily in DO statement processing when a tag is desired but no memory needs to be assigned. Only used in TGB.
101	Statement number tag
102	External function tag
103	Inline function tag
104	Statement function tag; arithmetic statement function.
106	Subroutine entry name tag
107	Function entry name tag; also used for implied-DO variables in DATA statements.
110-577	Dummy argument tags. Assigned in consecutive order; 110 is assigned the first dummy argument allowing for over 300 arguments per subroutine or function, or Pointee tags. Assigned in consecutive order; 110 is assigned the first pointee tag.
600*q*	Constant tag; refers to a constant rather than a variable. The digit immediately following the tag (*q*) is the subtype for the type of constant. Subtypes are as follows: *q* value — Explanation 0 — Refer to TBB; the offset field is an index into TBB. 1 — Constant can be machine-generated using an 071 machine instruction 4 — Immediate constant; 22-bit constant is supplied. 6 — Shifted constant; 22-bit constant entered in S register and shifted left 51 bits.

Table 4-2. TGB tag descriptions (continued)

Tag	Description
601	Used for dummy argument addresses. The offset field gives the offset from the address in B01 to the address passed in for a dummy argument.
602	Program block; positive relocation with respect to the origin of the current program. The program block is used for generated code, static variables and arrays, and constants.
603 (#TB)	In static mode, #TB holds temporary variables local to a code block; space in #TB is reused from block to block. #TB is not used in stack mode.
604 (#CL)	In static mode, #CL holds argument lists and the space into which passed-in argument lists are copied for multiple-entry routines. In stack mode, #CL holds argument list headers only (the headers are built as compile-time constants).
605 (#ST)	#ST is not used in static mode. In stack mode, all stacked entities except the B/T save area are in #ST (including entities in #TB and most entities in #CL in static mode). The offset field gives the offset from the run time address in B03 to the first word of the stacked entity.
606 (#RG)	Used for variables globally assigned to the B and T registers
607† (#DA)	Holds data (constants and static arrays and variables not in common)
	Tags 601, 605 (#ST), and 606 (#RG) are used internally by CFT; no loader tables are generated for them. Tags 602, 603 (#TB), 604 (#CL), and 607 (#DA) are treated as local blocks by the loader; a program unit using one of these blocks is assigned a unique (nonshared) area of memory.
610-777	User-declared common block tags. Assigned in consecutive order; 610 is assigned the first common block. This assignment allows up to 120 common blocks per subroutine or function. The loader treats these tags as common blocks; references to the same common block by two program units are treated as references to a single (shared) area of memory.

† Deferred implementation

4.3.3 MODE FLAGS

Several different groups of flags are referenced during compilation. These groups include:

- Statement type flags
- Parenthesis group type flags
- Generation mode flags
- TBT and Tag Buffer flags

If a flag bit is set (equal to 1), the entity has the named attribute.

4.3.4 TL FIELD

Tags also usually have a field called TL in bits 64 through 77. This field contains data type and length information. The TL field contains four octal digits. The first digit indicates data type and the remaining three denote data length, as follows:

Data type codes currently used by CFT include:

Type	Description
0	Typeless, Boolean
1	Logical
2	Character ASCII
4	Integer binary
6	Real
7	Complex

These data types are hierarchical; if operands of different types are combined, the lower type is converted to the higher type. However, not all type conversions are allowed. Type 7 (complex) is the highest type.

Data length is equal to the number of bits minus 1 of a quantity, unless Hollerith notation is used. In Hollerith notation, data length is equal to the number of characters. Data length values include:

Length	Description
027	24-bit
077	64-bit
177	128-bit (double-precision)

A typical TL field value might be 4027 for a 24-bit integer value or 6077 for a single-precision real value.

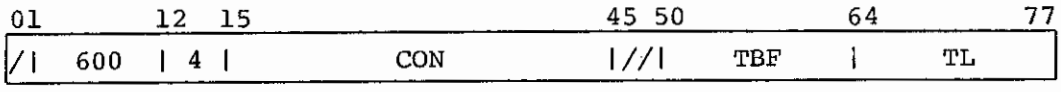
4.TBA TBA - ARRAY TABLE

Each array has a corresponding entry in TBA. An Array Table index in the Tag Table (TBT) points to each TBA entry. Arrays of the same dimensions use the same TBA entry.

For an array dimensioned N, the TBA has an N+1 word entry. The header word for this entry differs for fixed dimension and variable dimension arrays. The N word entries are the same regardless of array type.

Header word for fixed dimension array entry

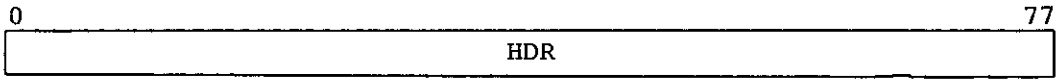
Format:



Field	Bits	Description
TAG	1-11	Tag; 600 (constant tag)
TYP	12-14	Constant tag subtype; 4 (integer)
CON	15-44	22-bit immediate constant (preceded by 2 sign bits); total array size
TBF	50-63	Tag Buffer flags (refer to 4.TGB)
TL	64-77	Type and length; 4027 for 22-bit integer, 4077 for 64-bit integer

Header word for variable dimension array entry

Format:

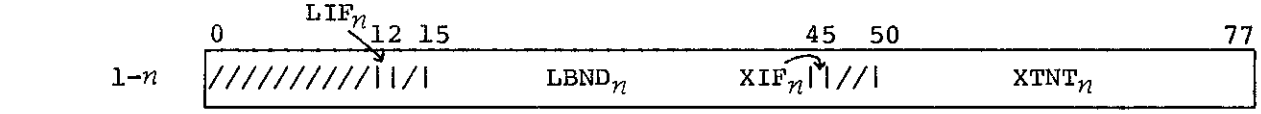


Field	Bits	Description
HDR	0-77	Header word; zero-filled

Entry word

The fields for the entry word are in word 1-n.

Format:



Field	Bits	Description
LIF_1 - LIF_n	12	Lower bound indirect flag; clear for a constant array
$LBND_1$ - $LBND_n$	15-44	Lower bound. If corresponding LIF is clear, contains actual lower bound. If LIF is set, contains pointer to the TBT entry for the compiler-generated local variable whose value is the actual lower bound
XIF_1 - XIF_n	45	Indirect extent flag; clear for a constant array
$XTNT_1$ - $XTNT_n$	50-77	Dimension length. If corresponding XIF is clear, contains the actual dimension length (or extent). If XIF is set, contains a pointer to the TBT entry for the compiler-generated local variable whose value is the actual dimension length

4.TBB TBB - CONSTANT, BINARY TABLE

TBB contains the loader TXT, Text Table (type 16). Refer to the COS Table Descriptions Internal Reference Manual, CRI publication SM-0045, for more information.

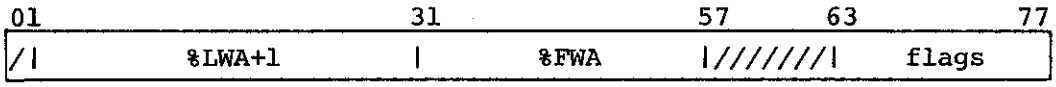
Constants are inserted into TBB during Pass 1. However, constants within DATA statements are forced into another table, TBE; then, at the end of Pass 1 the output from DATA statements is inserted in TBB.

Instructions are inserted during Pass 2.

4.TBBK TBBK - BLOCK DEFINITION TABLE

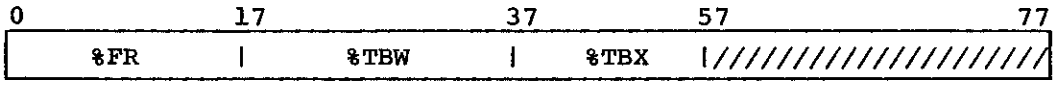
TBBK describes the characteristics of each statement in a block, such as the beginning of the block, the end of the block, and factors that can inhibit vectorization.

Format before compilation:



Field	Bits	Description
%LWA+1	1-30	%LWA+1 statement in TBG (before compilation)
%FWA	31-56	%FWA statement in TBG (before compilation)
Flags:	63-77	
BVL	63	Vectorizable function reference
BLS	64	ELSE block
BUB	65	Unconditional branch
BBR	66	Change in flow of control
BET	67	Block is executed every time
BVF	70	Nonvectorizable function reference
BLF	71	Block in loop
BFE	72	Forward entry within group
BND	73	End of group
BXE	74	Enter from outside group
BEX	75	Calls function or subroutine
BSP	76	Sub-block has a plus dependency
BSM	77	Sub-block has a minus dependency

Format after compilation:

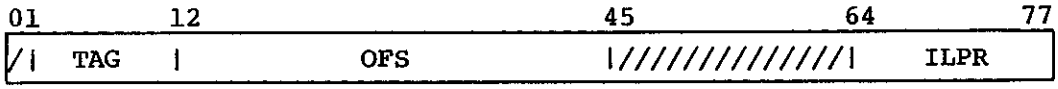


Field	Bits	Description
%FR	0-16	Length of TBFR (after compilation)
%TBW	17-36	Length of TBW (after compilation)
%TBX	37-56	Length of TBX (after compilation)

4.TBBR TBBR - B-REGISTER ASSOCIATES TABLE

TBBR is used in Pass 2 by CSRD and CSWR for loop mode variant scalars referenced before definition. CSRD forms the following entry.

Format:



Field	Bits	Description
TAG	1-11	Tag; one of the following: 601 Offset from B01 (dummy argument addresses) 602 Program block, positive relocation 603 (#TB) Temporary block tag 604 (#CL) Argument list block tag 605 (#ST) Offset from B03; stack tag. 606 (#RG) B/T register tag 607 (#DA) Data block tag 610-777 Common block tag (assigned in ascending order)
OFS	12-44	Bit offset from the base of the block
ILPR	64-77	Initial loop pseudo register

This entry is updated by CSWR to the following entry.

Format:

Field	Bits	Description
ILPR	45-60	Initial loop pseudo register
FIPR	64-77	Final loop pseudo register

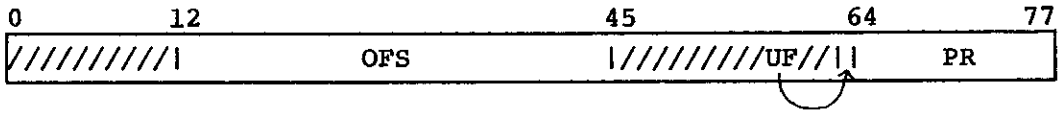
RASN uses TBBR passed on by CSWR and generates the following temporary entries as required.

Format:

Field	Bits	Description
UF	63	Previously used flag
PR	64-77	Pseudo registers

In addition, if the temporary variables overflow the available secondary registers, the entry is as follows.

Format:



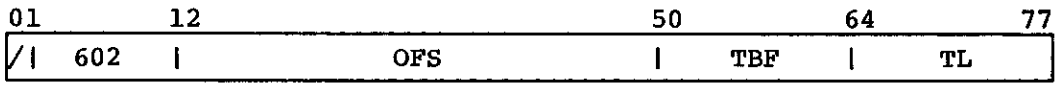
Field	Bits	Description
OFS	12-44	Bit offset from the base of the block
UF	63	Previously used flag
PR	64-77	Pseudo registers

This entry gives the block location assigned to the temporary variables.

4.TBC TBC - POINTER VARIABLE TABLE

TBC contains one entry for each pointer variable. Their format is similar to the argument format, but each pointer variable has its Dummy Argument flag (bit 50) and its Equivalence flag (bit 51) set making them identifiable as pointers.

Format:

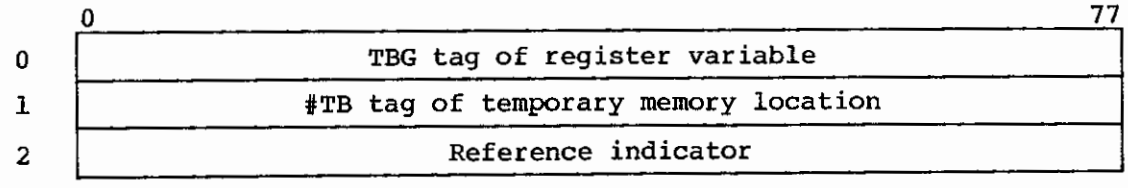


Field	Bits	Description
TAG	1-11	Tag; 602 (program block, positive relocation tag).
OFS	12-47	Index of TBT entry (Pass 1); bit offset within the block (Pass 2).
TBF	50-63	Tag Buffer flags (refer to 4.TGB)
TL	64-77	Type and length

4.TBCALL TBCALL - REGISTER VARIABLES TO RESTORE AFTER A CALL TABLE

TBCALL is used to restore T register variables after a CALL statement which may have side effects on the variable. CB21 copies the variable value to memory and uses the memory address as the actual argument. A 3-word entry is made for each register variable used as a call-by-address argument. After the CALL statement is compiled, CB48 scans TBCALL and generates code restoring the value from memory into the T register. TBCALL is set to empty by ABLK at the beginning of each block. CB21 and CB48 treat TBCALL as a stack and TBCALL is emptied by CB48 when the restore code is complete.

Format:



Field	Word	Bits	Description
	0	0-77	TBG tag of register variable
	1	0-77	Temporary block tag (#TB) of temporary memory location; 603.
	2	0-77	Reference indicator: 0 Multiple references to the T register variable in this statement @TBG of reference Only one reference to the T register variable If there are multiple variable references, the called routine cannot (legally) change the variable value and CB48 will not generate the restore code. If there is only one variable reference, the @TBG value is used by CSRE and CSWD to build TBY.

4.TBCLEN TBCLEN - CHARACTER LENGTH TABLE

During passes 1 and 2, TBCLEN contains the lengths, in characters, of all character entities declared in a program unit. TBCLEN will never contain more than 511 entries because the offset field (for indexing into TBCLEN) in a character tag is limited to 9 bits. The maximum length entered in this table cannot be larger than 16383.

Format:

Field	Bits	Description
LEN	0-77	Length in characters

4.TBCLTXT TBCLTXT - #CL TEXT TABLE

During pass 2, TBCLTXT contains the loader TXT, Text Table (type 16) for #CL. See the COS Table Descriptions Internal Reference Manual, CRI publication SM-0045 for more information.

#CL contains argument lists. See the Macros and Opdefs Reference Manual, CRI publication SR-0012 for argument list format. All argument lists begin with an argument list header. The header contains the sequence number of the source code line that generated the call, a number-of-arguments field, and flags indicating if the call is for a character function or call-by-value.

Argument transmission is either call-by-value or call-by-address mode. In call-by-value mode, a header is built and the number-of-arguments field is the number of registers set up for the call. In call-by-address mode, one word is saved for each argument in addition to the header word. Call-by-address character functions are an exception to the general call-by-address format and save one additional word reflecting the address-of-result argument invented by the compiler.

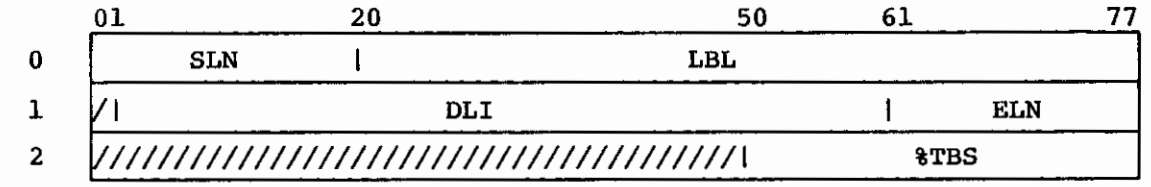
4.TBCT TBCT - CONJUNCTIVE TERM TABLE

TBCT contains the conditions for safe vectorization of a conditional vector loop in the form of an IF statement in TGB format.

4.TBD TBD - DO LOOP TABLE

TBD contains a 3-word entry for each DO-loop encountered by the compiler.

Format:



Field	Word	Bits	Description
SLN	0	0-17	Starting line number; keyed to the source listing line number.
LBL	0	20-77	Label that ends the DO-loop; 6 ASCII characters, right-justified, and zero-filled.
DLI	1	1-60	DO-loop index; 8 compressed (6-bit) ASCII characters, left-justified. A bias of 40B is used for compression.
ELN	1	61-77	Ending line number, keyed to the source listing line number
%TBS	2	50-77	Index into TBS for the label that is the start of the DO-loop. The TBS entry for the end-of-loop label is always the next TBS entry.

4.TBDT TBDT - DISJUNCTIVE TERM TABLE

TBDT contains the conditions for safe vectorization of a subscript dependency in the form of a logical expression in TGB format. TBDT is copied to TBCT at the end of dependency analysis for a single subscript dependency.

4.TBE TBE - EXTERNAL REFERENCE TABLE

In Pass 1, TBE contains the converted constant values from the constant lists in the DATA statements.

During Pass 2, TBE contains the loader XRT, the External Relocation Table (type 14).

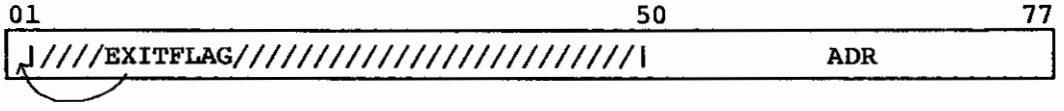
See the COS Table Descriptions Internal Reference Manual, CRI publication SM-0045 for more information.

4.TBEE TBEE - ENTRY/EXIT ADDRESS TABLE

Several quantities used in entry and exit sequences are unknown until the end of Pass 2. These quantities include the starting addresses and sizes of the B and T register save areas and the size of the stack frame in the stack sequence.

TBEE is cleared at the beginning of Pass 2 by LTST. During Pass 2, LT50 adds one item to TBEE for each entry and exit sequence encountered. After Pass 2, LT48 uses the addresses in TBEE to locate the entry and exit sequences and inserts the correct counts and addresses into the sequences.

Format:



Field	Bits	Description
EXITFLAG	0	Set for an exit sequence and clear for an entry sequence
ADR	50-77	Address of the entry or exit sequence; a parcel address relative to the program code base.

4.TBF TBF - STATEMENT FUNCTION SKELETON TABLE

During Pass 1, this table contains statement function skeletons. The statement function definitions are stored in TBF as they are encountered and are expanded as macros when they are referenced during Pass 1.

A skeleton consists of a tag-operator string similar to that compiled in TGB for an expression.

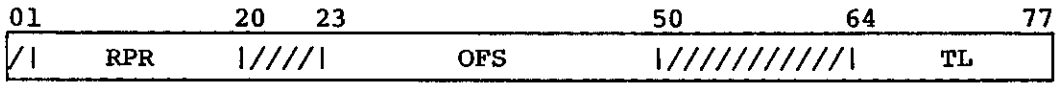
The parameters are given a tag with a bias of 100.

TBF is not used during Pass 2.

4.TBFR TBFR - CALL-BY-VALUE REFERENCE TABLE

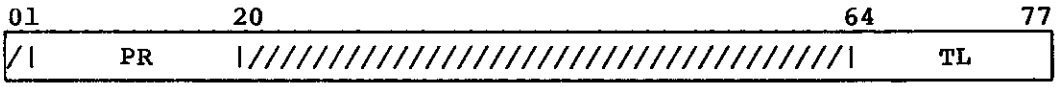
This table contains a primary entry for each unique call-by-value function reference in a block. Associated with each primary entry are one to six secondary entries corresponding to the arguments of the function reference.

Primary entry format:



Field	Bits	Description
RPR	1-17	Result pseudo register
OFS	23-47	Offset
TL	64-77	Type and length

Secondary entry format:



Field	Bits	Description
PR	1-17	Pseudo register
TL	64-77	Type and length

4.TBG TBG - DATA STATEMENT / TAG BUFFER TABLE

During Pass 1, TBG contains the DATA statements in Tag Buffer (TGB) format. (Refer to the TGB format description in this section for more information.)

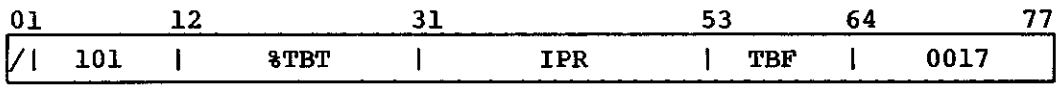
During Pass 2, TBG contains the Tag Buffer built during Pass 1. Descriptions of the different kinds of entries made to TBG follow.

The CFT statement type flags (STF field) set in the statement header of TBG are:

Bit	Flag	Description
37	ENF	ENTRY statement
40	DBF	Beginning of a DO
41	CSF	Conditional statement
42	RPF	Replacement statement
43	CAF	CALL statement
44	ISF	IF statement
45	GTF	GO TO statement
46	CNF	CONTINUE statement
47	IDF	Ignore Vector Dependency flag

Statement number reference entry

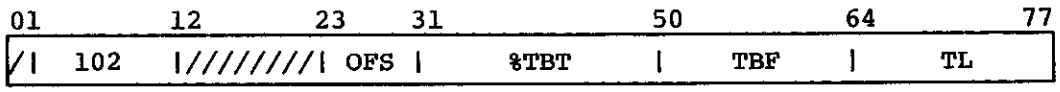
Format:



Field	Bits	Description
TAG	1-11	Tag; 101 (statement number tag).
%TBT	12-30	Index into TBT entry
IPR	31-52	Index into previous reference in TBG. This index is a backward index relative to the LWA+1 of TBG.
TBF	53-63	Tag Buffer flags (refer to 4.TGB)
TL	64-77	Type and length; 0017.

External function tag

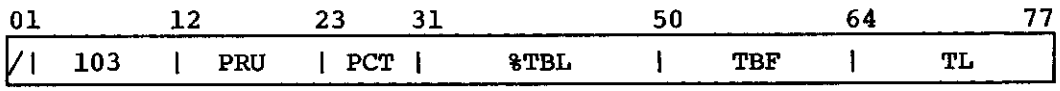
Format:



Field	Bits	Description
TAG	1-11	Tag; 102 (external function tag).
OFS	23-30	Parcel offset from primary entry
%TBT	31-47	%TBT of external function
TBF	50-63	Tag buffer flags
TL	64-77	Type and length

Intrinsic function entry

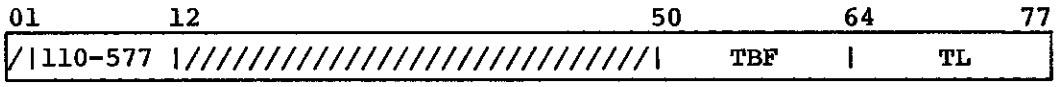
Format:



Field	Bits	Description
TAG	1-11	Tag; 103 (intrinsic function tag).
PRU	12-22	Pseudo registers used
PCT	23-30	Parameter count; number of arguments to the function.
%TBL	31-47	Index into TBL entry
TBF	50-63	Tag Buffer flags (refer to 4.TGB)
TL	64-77	Type and length

Dummy argument entry

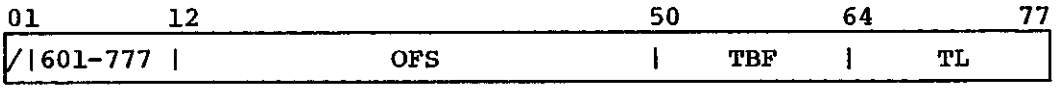
Format:



Field	Bits	Description
TAG	1-11	Tag; a number from 110 to 577 (dummy argument tag).
TBF	50-63	Tag Buffer flags (refer to 4.TGB)
TL	64-77	Type and length

Program and common block entry

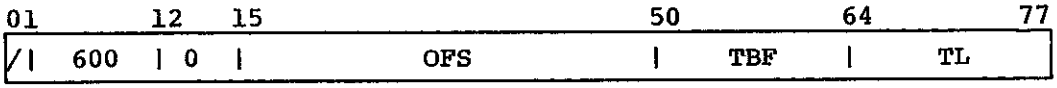
Format:



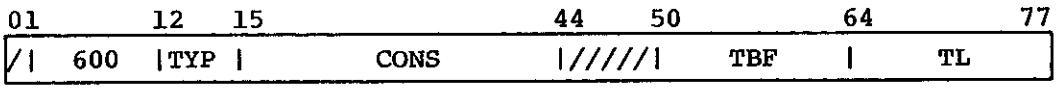
Field	Bits	Description
TAG	1-11	Tag; one of the following: 601 Offset from B01 (dummy argument addresses) 602 Program block, positive relocation 603 (#TB) Temporary block tag 604 (#CL) Argument list block tag 605 (#ST) Offset from B03; stack tag. 606 (#RG) B/T register tag 607 (#DA) Data block tag 610-777 Common block tag (assigned in ascending order)
OFS	12-47	Bit offset from the base of the block
TBF	50-63	Tag Buffer flags (refer to 4.TGB)
TL	64-77	Type and length

Constant entry

Format:



Field	Bits	Description
TAG	1-11	Tag; 600 (constant tag).
TYP	12-14	Constant type; 0 (offset field is index into TBB).
OFS	15-47	Bit offset into block
TBF	50-63	Tag Buffer flags (refer to 4.TGB)
TL	64-77	Type and length

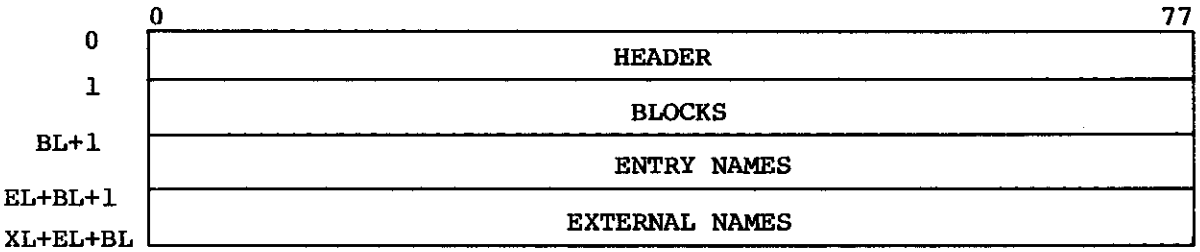


Field	Bits	Description
TAG	1-11	Tag; 600 (constant tag).
TYP	12-14	Constant type (6 if shifted; 4 if immediate.)
CONS	15-44	Constant value
TBF	50-63	Tag Buffer flags (refer to 4.TGB)
TL	64-77	Type and length

4.TBH TBH - PROGRAM DESCRIPTION TABLE

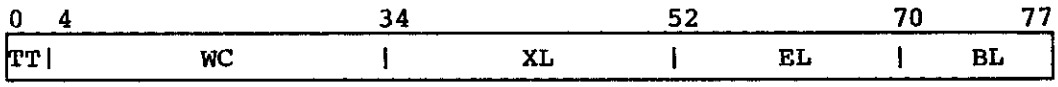
TBH is a simplified form of the loader's Program Description Table (PDT). See the COS Table Descriptions Internal Reference Manual, CRI publication SM-0045, for more information on PDT. The format of TBH follows.

Format:



Header word

Format:

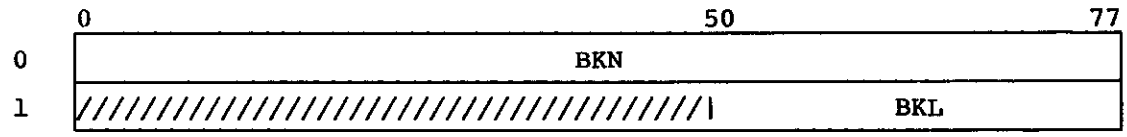


Field	Bits	Description
TT	0-3	Table type; 17.
WC	4-33	Word count; number of words in table.
XL	34-51	Word count for external names
EL	52-67	Word count for entry names
BL	70-77	Word count for blocks

Blocks

Each block has a 2-word descriptor in TBH. The first block is the program block, the second block is #TB, the third block is #CL, and so on. Following the blocks created by the compiler are descriptors for common blocks declared by the user.

Format:



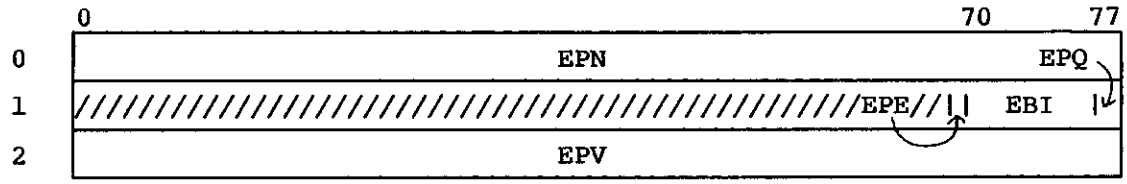
Field	Word	Bits	Description
BKN	0	0-77	Block name; in ASCII, left-justified and zero-filled. Blank common; if present, has name //L.
BKL	1	50-77	BLock length; length in words of the named block.

The program block (the first block) has many flags set in the second word of the descriptor. See the COS Table Descriptions Internal Reference Manual, publication SM-0045, for more information on the PDT flags.

Entry names

Each entry point to a program unit has a 3-word descriptor in TBH.

Format:



Field	Word	Bits	Description
EPN	0	0-77	Entry name; in ASCII, left-justified and zero-filled.
EPE	1	67	Primary entry flag; set for a program and clear for a subroutine or function subprogram.
EBI	1	70-76	Block index; 2 (program block).
EPQ	1	77	Relocation mode; 1 (parcel address).

Field	Word	Bits	Description
EPV	2	0-77	Entry value; parcel address of the entry point.

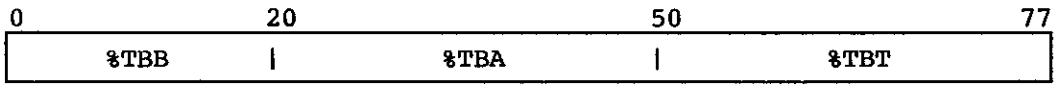
External names

Each external routine called by a program unit has a 1-word entry in TBH. The entry is the external routine name in ASCII, left-justified and zero-filled. The external index of an external routine is the origin-0 index into the external names section in TBH of the external routine's name. The first external name listed in TBH has an external index of 0, the second external name listed has an external index of 1, and so on.

4.TBI TBI - TBA INDEX TABLE

TBI is built when option Z (Debug Symbol Table) is specified on the CFT control statement. TBI contains a 1-word entry for each array in the FORTRAN routine. This table retains pointers to the array dimension descriptors during Pass 2 so that they may be included in the Debug Symbol Table.

Format:

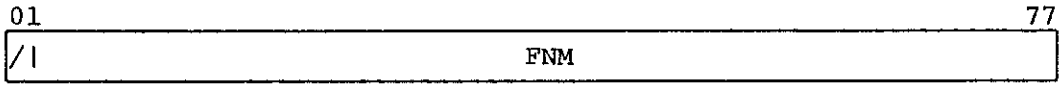


Field	Bits	Description
%TBB	0-17	Index into TBB of array name
%TBA	20-47	Index into TBA
%TBT	50-77	Index into TBT

4.TBJ TBJ - INTRINSIC FUNCTION NAME TABLE

TBJ contains the names of all intrinsic functions, ANSI and non-ANSI. TBJ is a static table formed when the compiler is assembled. The macro INTRIN makes entries in TBJ and its companion table, TBK. TBJ is divided into two parts, ANSI function names and non-ANSI function names. Each part has generic function names arranged alphabetically. Specific function names are grouped with the appropriate generic name.

Format:

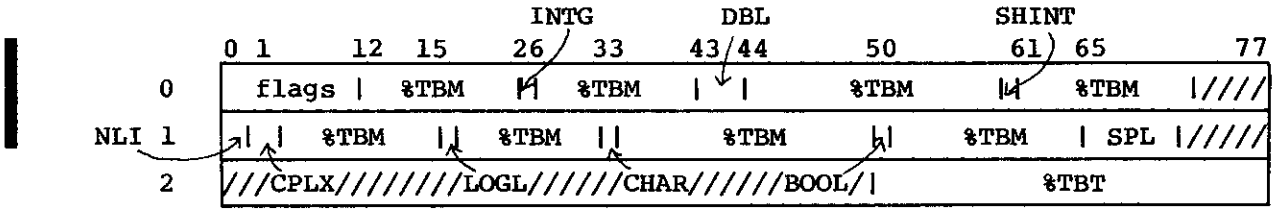


Field	Bits	Description
FNM	1-77	Function name; in 8-bit ASCII characters, right-justified, zero-filled

4.TBK TBK - INTRINSIC FUNCTION ATTRIBUTE TABLE

TBK contains attribute information for the intrinsic function names appearing in TBJ. This information includes TBM indices for the proper skeleton, the number of arguments, and flag bits indicating if the function is ANSI or if the function can be passed. Each TBK entry contains three words. Entry *n* of TBK corresponds to entry *n* of TBJ. The macro INTRIN makes entries in TBK and TBJ.

Format:



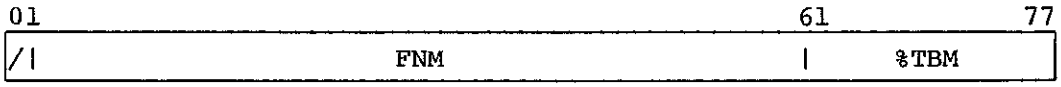
Field	Word	Bits	Description
Flags:	0	0-11	
EXT	0	0	Set if function name is declared in an EXTERNAL statement
INTRIN	0	1	Set if function name is declared in an INTRINSIC statement
ANSI	0	2	Set if function is part of the ANSI standard set of functions
NOPAS	0	3	Set if the function name cannot be passed as an actual argument
GENER	0	4	Set if the function name is a generic name but not a specific name
MXMN	0	5	Set if the function has a variable number of arguments, applies only to functions finding the maximum or minimum of a series of arguments
ARGS	0	6-10	Number of arguments
REAL	0	11	Set if the real argument is valid for the function
	0	12-25	%TBM if REAL is set

Field	Word	Bits	Description
INTG	0	26	Set if long integer arguments are valid for the function
	0	26-42	%TBM if INTG is set
DBL	0	43	Set if double-precision arguments are valid for the function
	0	44-57	%TBM if DBL is set
SHINT	0	60	Set if short integer arguments are valid for the function
	0	61-74	%TBM if SHINT is set
NLI	1	0	Set if function name no longer retains its intrinsic property, such as when declared as an array
CPLX	1	1	Set if complex arguments are valid for the function
	1	2-15	%TBM if CPLX is set
LOGL	1	16	Set if logical arguments are valid for the function
	1	17-32	%TBM if LOGL is set
CHAR	1	33	Set if character arguments are valid for the function
	1	34-47	%TBM if CHAR is set
BOOL	1	50	Set if Boolean arguments are valid for the function
	1	51-65	%TBM if BOOL is set
SPL	1	65-73	Index into computed $GOTO near OP90; jumps to appropriate special processor.
%TBT	2	50-77	%TBT secondary entry if intrinsic function is passed as an actual argument

4.TBL TBL - EXTERNAL LIBRARY TAG TABLE

This table contains the names of all external library function tags, mainly the I/O routines. TBL, along with TBM, resembles the Symbol and Tag Tables (TBS and TBT) in format. TBL is a static table formed when the compiler is assembled and is kept in sorted order.

Format:

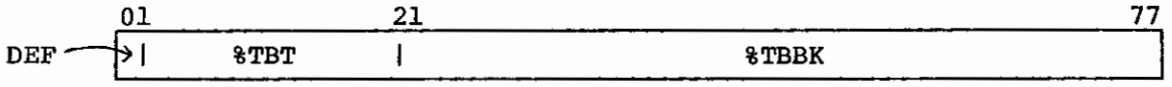


Field	Bits	Description
FNM	1-60	Function name; in 6-bit (compressed) ASCII characters, left-justified, zero-filled
%TBM	61-77	Index into the corresponding function definition in TBM

4.TBLB TBLB - LABEL USAGE TABLE

TBLB describes the flow of control in the table by describing the block label usage and definitions.

Format:



Field	Bits	Description
DEF	0	Label entry type: 0 Label definition entry 1 Label reference entry
%TBT	1-20	%TBTE of label
%TBBK	21-77	%TBBKE of statement using this statement number

4.TBM TBM - LIBRARY MACRO TABLE

TBM contains the function tags for external functions and skeletons for the internal statement functions and the macro functions. TBM, along with TBL, resembles the Symbol and Tag Tables (TBT and TBS) in format. TBM is a static table that is formed when the compiler is assembled. The name of the function in left-justified 8-bit ASCII appears before the function tags for the macro functions.

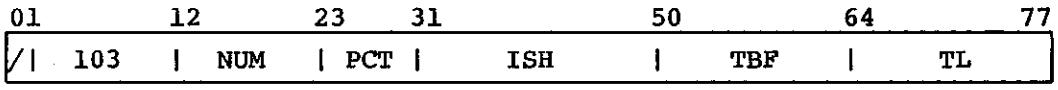
The convention for an inline function with a scalar and a vector definition is:

1. A header for the definition (of the form I@name)
2. A header for the scalar definition (of the form I$name)
3. One word per instruction for the scalar definition for that instruction
4. A zero word flagging the end of the scalar definition
5. A header for the vector definition (of the form I%name)
6. One word per instruction for the vector definition for that instruction
7. A zero word flagging the end of the vector definition

If no vector definition for the function exists, item 1 and items 5 through 7 are omitted.

Function header word

Format:



Field	Bits	Description
TAG	1-11	Tag; 103 (intrinsic function tag).
NUM	12-22	Number of words in scalar definition (including the header and trailer)
PCT	23-30	Parameter count; number of arguments to the function.
ISH	31-47	Index into TBM of scalar header
TBF	50-63	Tag Buffer flags (refer to 4.TGB)
TL	64-77	Type and length

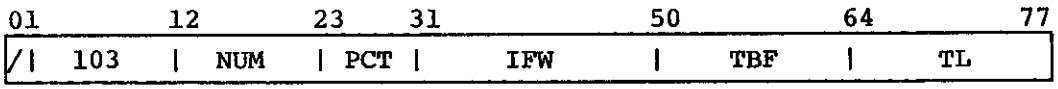
Scalar definition header

Format:

Field	Bits	Description
TAG	1-11	Tag; 103 (intrinsic function tag).
NUM	12-22	Number of pseudo registers used
PCT	23-30	Parameter count; number of arguments to the function.
IFW	31-47	Index into TBM of first word of scalar definition
TBF	50-63	Tag Buffer flags (refer to 4.TGB)
TL	64-77	Type and length

Vector definition header

Format:

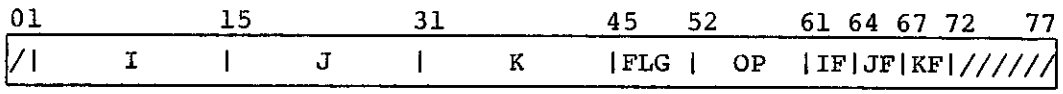


Field	Bits	Description
TAG	1-11	Tag; 103 (intrinsic function tag).
NUM	12-22	Number of pseudo registers used
PCT	23-30	Parameter count; number of arguments to the function.
IFW	31-47	Index into TBM of first word of vector definition
TBF	50-63	Tag Buffer flags (refer to 4.TGB)
TL	64-77	Type and length

Instruction entry for scalar or vector definition

Each definition consists of a number of instruction entries, one instruction per word.

Format:



Field	Bits	Description
I	1-14	I field of instruction, pseudo register number, or parameter number
J	15-30	J field of instruction, pseudo register number, or parameter number
K	31-44	K field of instruction, pseudo register number, or parameter number
FLG	45-51	Flags, as follows: 45 If set, I field is result or is unused; if clear, I field is operand pseudo register or constant. 46 If set, J field is result or is unused; if clear, J field is operand pseudo register or constant. 47 If set, K field is result or is unused; if clear, J field is operand pseudo register or constant. 50 If set, I is both an operand and a result (as in shift operations). If set and operand is floating-point multiply, perform a rounded multiply even if rounding is disabled. 51 If set and operand is a floating-point instruction, inhibit the truncation option for this instruction.
OP	52-60	Opcode for instruction to be generated
IF	61-63	Field flag for I field
JF	64-66	Field flag for J field

Field	Bits	Description
KF	67-71	Field flag for K field

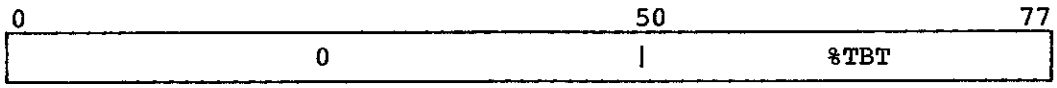
Field flags are as follows.

Value	Explanation
4	Field is the number of a pseudo register to use (beginning with 1).
7	Field is the number of the argument to the function (beginning with 0 from left to right).
0	Field is a constant or unused; substitute directly.

4.TBNOBT TBNOBT - TBT INDEX OF VARIABLES NOT ASSIGNABLE TO B/T REGISTER TABLE

TBNOBT is used in Pass 1 to identify variables which cannot be assigned to T registers. If OPT=NOBTREG is specified on the CFT control statement, no entries are made. NOBTVAR makes the entries. Entries are made for variables named in DATA, NAMELIST, and SAVE statements, and for variables used in I/O control lists, such as IOSTAT. The loop at EN13 scans the table, and variables found in the table are assigned to memory instead of a T register.

Format:



Field	Bits	Description
%TBT	50-77	Index of TBT entry

4.TBO TBO - ARRAY BOUNDS CHECKING TABLE

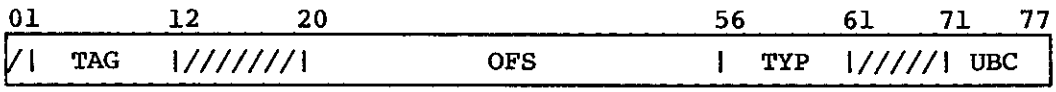
TBO contains array bounds checking information. If word 0 of TBO is negative, all arrays are checked for subscript bounds errors.

If word 0 is positive, the remaining words in TBO contain the 8-bit ASCII names of the arrays to be bounds checked. These names are left-justified and zero-filled.

4.TBP TBP - COMMON AND EQUIVALENCE / DUMMY ARGUMENT ADDRESS TABLE

The following TBP entry exists for each entity in a common block.

Format:



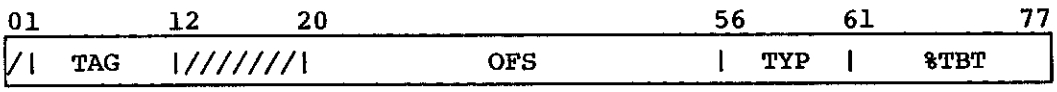
Field	Bits	Description
TAG	1-11	Tag of the common block to which this entity belongs
OFS	20-55	Bit offset from the base of the common block for the beginning location of this entity: 20-47 Word offset 50-55 Bit offset into the first word.
TYP	56-60	Type of the common block if all entities in the block must have the same type (that is, character)
UBC	71-77	Unused bit count in the last word of this entity. This field can be nonzero if the common block is of type character.

At the end of declarative processing during Pass 1, the EQUIVALENCE statements packed in TBR are processed and an entry for each equivalenced variable is made TBP. Variables used in EQUIVALENCE statements are assigned storage and TBP is cleared at the end of Pass 1.

Following the common block entries in TBP is a zero word, separating common block entries from equivalence entries. Equivalence entries are also terminated by a zero word.

The following TBP entry exists for each entity in an EQUIVALENCE statement.

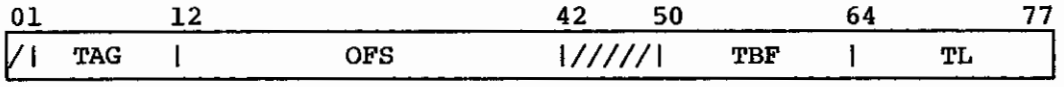
Format:



Field	Bits	Description
TAG	1-11	After EQ and before EN, TAG is the common block tag for the equivalenced entity or the index into TBT of the equivalence group's base tag. After EN, TAG is the Common or Program Block tag for the equivalenced entity
OFS	20-55	Bit offset for the beginning location of this entity. After EQ and before EN, the offset is from the base of the common block or the equivalence group's base tag. After EN, the offset is from the base of the common, static, or stack block.
TYP	56-60	Type of the equivalenced entity if the entity must be equivalenced to another entity of the same type
%TBT	61-77	Index into TBT for the variable or array being equivalenced

During Pass 2, the addresses of dummy arguments are entered into TBP. For example, the first TBP entry gives the address of the 110-tagged dummy argument.

Format:

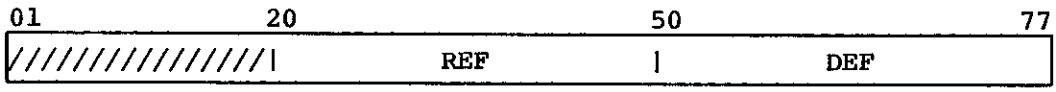


Field	Bits	Description
TAG	1-11	In the old sequence, 602 (program block, positive relocation tag) In the new sequence, either 601 (offset from (B01)) for a single-entry routine or 604 (positive relocation into #CL) for a multiple-entry routine
OFS	12-41	Word offset from block base
TBF	50-63	Tag Buffer flags (refer to 4.TGB)
TL	64-77	Type and length

4.TBPD TBPD - PLUS DEPENDENCY TABLE

TBPD is used by code generation to resolve subsequent plus dependencies.

Format:



Field	Bits	Description
REF	20-47	Index of reference in TBG
DEF	50-77	Index of definition in TBG

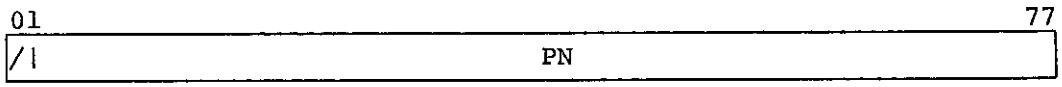
4.TBPG TBPG - PAGE NUMBER TABLE

TBPG, along with the Program Name Table (TBPN), keeps track of the program units compiled by CFT.

The position of any program unit name can be found in TBPN. Then, using the TBPN pointer, the page number for that program unit can be referenced in TBPG.

Entries are made to TBPG as encountered, using routine ELWD.

Format:



Field	Bits	Description
PN	1-77	Page number in 8-bit ASCII, left-justified and blank-filled

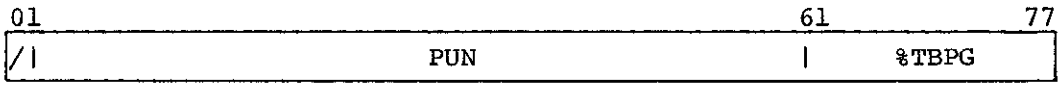
4.TBPN TBPN - PROGRAM UNIT NAME TABLE

TBPN, along with the Page Number Table (TBPG), keeps track of the program units compiled by CFT.

TBPN has one entry for each program unit encountered. This entry includes an index into TBPG, giving the page number location of that program unit.

TBPN, like TBS, is maintained in sorted order.

Format:



Field	Bits	Description
PUN	1-60	Program unit name in 6-bit ASCII
%TBPG	61-77	Index into corresponding entry in TBPG

4.TBQ TBQ - VARIABLE DECLARATOR TABLE

TBQ is used for variable dimension declarators. It contains statements in Pass 1 tag-and-operator format that generate code to save the run-time value of a variable declarator on entry to the program unit.

The format for these entries is the same as that for replacement statements in TGB. The statements in TBQ are moved into TGB during END statement processing before Pass 2 begins.

4.TBR TBR - PACKED EQUIVALENCE / BLOCK RELOCATION TABLE

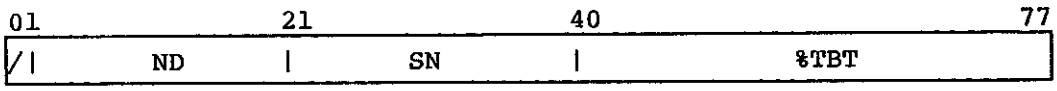
During Pass 1, TBR resembles one large EQUIVALENCE statement. All EQUIVALENCE statements within a program are packed up, eight characters per word, in TBR. The remnant part of an EQUIVALENCE statement is packed, left-justified, and zero-filled into TBR. When a new EQUIVALENCE statement appears, the current sequence number in T.SH is entered into TBR separating each EQUIVALENCE statement. The sign bit set in this word distinguishes it from words with packed characters. These sequence numbers are used during subsequent EQUIVALENCE processing (at EQ29) ensuring correct sequence numbers in error messages and cross-references. Packed EQUIVALENCEs are retained in TBR throughout all declarative processing. TBR is processed at the end of declarative processing in Pass 1. During executable statement processing, the DO parameter stack is retained in TBR. In addition, the conditional block stack is maintained in TBR. A conditional block entry has the sign bit set; the sign bit is clear for a DO entry.

During Pass 2, the loader BRT, Block Relocation Table (type 15) is generated in TBR. See the COS Table Descriptions Internal Reference Manual, CRI publication SM-0045, for more information.

The DO entry consists of the following words.

- Terminal Statement Number flag
- DO variable tag, DOUT
- DO increment tag, INCT
- Iteration count tag, ITRT
- Beginning statement number tag, BSNT
- Ending statement number tag, ESNT

Format:

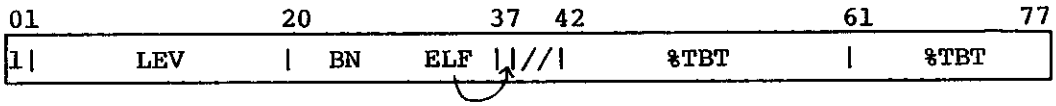


Field	Bits	Description
ND	2-20	Nesting depth
SN	21-37	Statement number
%TBT	40-77	%TBT of generated statement number

Conditional block entries

TBR contains one entry per conditional block statement.

Format:



Field	Bits	Description
1	0	Indicates a conditional block statement
LEV	1-17	IF-level (≥1)
BN	20-36	Block number. Internal statement number for the conditional statement
ELF	37	ELSE flag. Set if this entry corresponds with an ELSE statement
%TBT	42-60	Index into TBT of the generated statement number assigned to the ENDIF statement
%TBT	61-77	Index into TBT of the generated statement number for the next conditional block within this IF-level

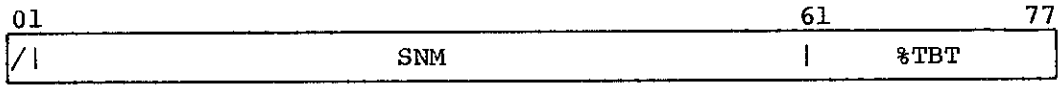
4.TBS TBS - SYMBOL TABLE

TBS is the Symbol Table, sorted alphabetically. One entry appears in TBS for each symbol referenced. The corresponding Tag Table is TBT.

TBS is a list of all symbol names encountered by the compiler. The attributes of each symbol name are kept in TBT, the Tag Table. TBS maintains an index for each symbol name's corresponding TBT entry.

Refer to the first part of the tables section in this manual for a description of how TBS is maintained.

Format:



Field	Bits	Description
SNM	1-60	Symbol name in 6-bit ASCII (left-justified and zero-filled) or 5-digit statement number with leading zeros
%TBT	61-77	Index into corresponding TBT entry

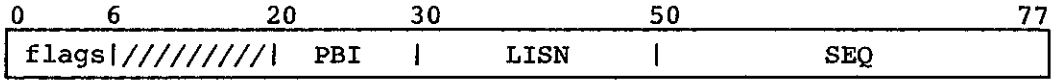
4.TBSB TBSB - SUBSTRING DEFINITION TABLE

TBSB is the Substring Definition Table used during Pass 2. TBSB contains the first character and length definitions of active substrings in a block. The entry is two words, the first word is the first character tag in TGB format and the second word is the length of the substring tag in TGB format.

4.TBSN TBSN - SEQUENCE NUMBER TABLE

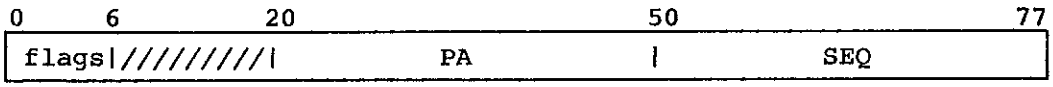
TBSN is the table of sequence numbers. RNXT makes an entry for each statement in a program unit. ABRA uses information added to the table in Pass 1 to perform branch statement analysis. DETB uses information added during Pass 2 to write parcel addresses to the Debug Symbol Table when the DEBUG parameter is specified on the CFT control statement.

Format for Pass 1:



Field	Bits	Description
Flags:	0-5	
SNSMT	0	Symbol Table flag; set in ABLK if entry is written to the SMT by routine DETB.
SNMF	1	Mode flag; set if DEBUG is enabled for this statement.
SNEL	2	ELSE or ELSEIF flag; indicates the statement is an ELSE or ELSEIF statement.
SNEI	3	ENDIF flag; indicates the statement is an ENDIF statement.
SNIF	4	IF flag; set if the statement is the beginning of a block IF statement.
SNNT	5	ENTRY flag; set for ENTRY statements.
PBI	20-30	Parent block index; nesting level of current IF, ELSE, or ELSEIF block.
LISN	31-47	Last internal sequence number; set for IF, ELSE, or ELSEIF statements, this is the internal sequence number of the last statement number in the block.
SEQ	50-77	Sequence number of the statement

Format for Pass 2:



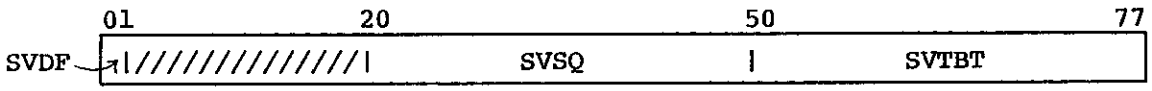
Field	Bits	Description
Flags:	0-5	
SNSMT	0	Symbol Table flag; set in ABLK if entry is written to the SMT by routine DETB.
SNMF	1	Mode flag; set if DEBUG is enabled for this statement.
SNEL	2	ELSE or ELSEIF flag; indicates the statement is an ELSE or ELSEIF statement.
SNEI	3	ENDIF flag; indicates the statement is an ENDIF statement.
SNIF	4	IF flag; set if the statement is the beginning of a block IF statement.
SNNT	5	ENTRY flag; set for ENTRY statements.
PA	20-47	Parcel address of the first executable instruction generated for the FORTRAN statement. This field is defined only for executable statements.
SEQ	50-77	Sequence number of the FORTRAN statement

4.TBSV TBSV - SAVED VARIABLE TABLE

TBSV identifies the local variables assigned to permanent static storage. Entries are made by the SAVE statement processor (routine SAST) and the DATA statement processor (routine DAST). Routine SA50 uses TBSV for SAVE statement error checking and assigning the saved variables to static storage.

TBSV exists only during Pass 1. At the end of Pass 1, routine LTST collapses TBSV. TBSV is reused during Pass 2 under the name TBEE (the Entry/Exit Address Table).

Format:



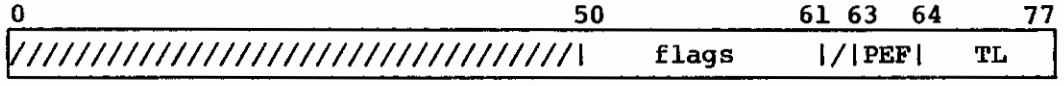
Field	Bits	Description
SVDF	0	Data flag; set only if the entry was made because the variable appeared in a DATA statement.
	1-17	Reserved for future use; zero-filled.
SVSQ	20-47	Sequence number of the source text line containing the SAVE or DATA statement for which this entry was made
SVTBT	50-77	Index into TBT of the variable's tag

4.TBT TBT - TAG TABLE

TBT is the Tag Table for TBS. Basically, the two tables are arranged in one-to-one correspondence. Given a symbol name, its position can be found in TBS. Then, using the pointer in the TBS entry, the attributes of the symbol can be referenced in TBT.

Entries are made to TBT as they are encountered, using routine ELWD. When a new symbol is encountered, tag type is determined by syntax analysis of the statement.

Tag Buffer flags are as follows:

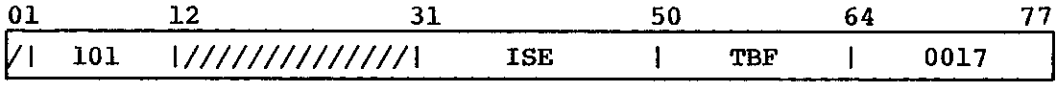


Field	Bits	Description
Flags:	50-61	
DAF	50	Dummy Argument flag
EQF	51	Equivalence flag
FNF	52, 53-55	Function flag; if flag is set, the symbol name is a function and bits 53-55 are as follows: 53 Call-by-value flag (CBV) 54 Function/subroutine call flag (FSC) 55 Single/multiple result function flag (SMR) If flag is clear and bits 53-55 are zeros, then the symbol name is a simple variable. Otherwise, the symbol name is an array; bit 52 is clear and bits 53-55 contain the number of dimensions in the array (up to 7). If the TBT entry is the tag for the program unit being compiled, FNF is set and bits 53-55 determine the type of program unit: bit 53 for the main program, bit 54 for the subroutine, and bit 55 for the function. Bits 53-55 are clear for a BLOCK DATA subprogram.
DFF	56	Defined Entry flag; set if the symbol name was defined (assigned a value) either 1) in a DATA statement, 2) on the left-hand side of a statement, 3) in a read, or 4) as an argument in a call.

Field	Bits	Description
RFF	57	Referenced Entry flag; set if the symbol name was referenced either on the right-hand side of a statement or in a call.
TDF	60	During nonexecutable statement processing for variables, this bit is the Explicit Type flag. If set, flag indicates that the variable was mentioned in a typing statement. This bit is also the DO Terminator flag; it is set if the variable is in use as a DO-loop terminator. This flag is set at the start of each DO-loop and cleared at the end. This bit is set for DO-loop terminating statement numbers during Pass 1.
ASF	61	Assigned Statement Number flag / TBF Skeleton flag
PEF	63	Previous Entry flag. If tag is a dummy argument tag, this flag indicates that the dummy argument appeared in a previous entry.
TL	64-77	Type and length

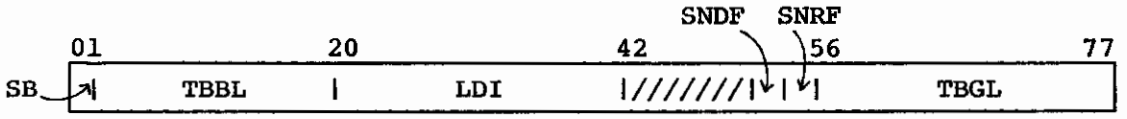
Statement number entry

Format of TBT primary entry for a statement number during Pass 1:



Field	Bits	Description
TAG	1-11	Tag; 101 (statement number tag).
ISE	31-47	Index into TBT secondary entry, if one exists
TBF	50-63	Tag Buffer flags
TL	64-77	Type and length; 0017.

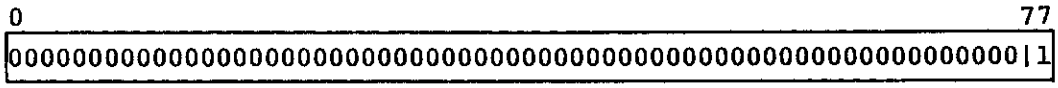
Format of a TBT primary entry for a statement label during Pass 2:



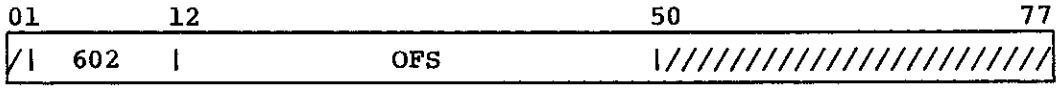
Field	Bits	Description
SB	0	Sign bit; set for statement numbers with secondary entries. This bit is set between Pass 1 and Pass 2 when the initial jump instruction is generated to the label and stored at the offset specified by the secondary entry. This bit is set for other labels when the statement number definition is compiled and entered in TBB.
TBBL	1-17	TBB last reference index. TBBL is the index of the last reference to this label in TBB relative to T.PBS. TBBL is updated for each reference compiled and is the head of the chain of references in TBB.
LDI	20-41	Label definition index; initially an index relative to the LWA+1 of TBG pointing to the statement header of the statement where the label is defined. When the statement number definition is compiled, it becomes the index of the definition in TBB.
SNDF	54	Statement number defined flag; set until the label definition is compiled and entered in TBB.
SNRF	55	Statement number referenced flag; set for referenced statement numbers.
TBGL	56-77	TBG last reference index. TBGL is the index to the last reference of the label in TBG relative to the LWA+1 of TBG. TBLG serves as the head of the chain of references in TBG.

A TBT secondary entry exists for statement numbers with parcel addresses must be read into a register at run time. User-defined statement numbers appearing in ASSIGN statements or END= or ERR= branches of I/O statements require secondary TBT entries. Compiler generated labels also have secondary entries when associated with the first word address of a jump table generated for alternate return subroutine calls and computed GOTOs.

Format of a TBT secondary entry during Pass 1:



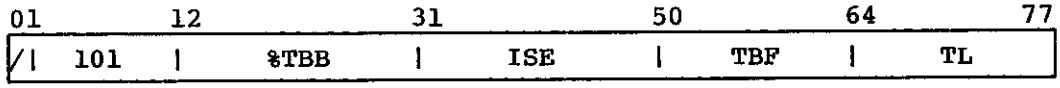
Format of a TBT secondary entry during Pass 2:



Field	Bits	Description
TAG	1-11	Tag; 602, constant (program) block.
OFS	12-47	Offset in the constant block where a statically allocated word containing a jump instruction to the parcel address of the statement label is stored. Loading an A register from this word gives the parcel address of the label (in the generated code).

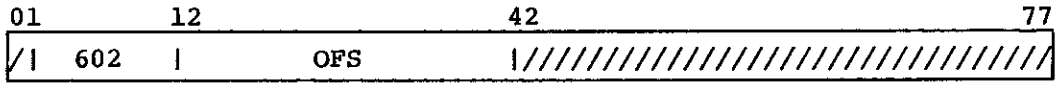
Format number entry

Format of TBT primary entry for a format number:



Field	Bits	Description
TAG	1-11	Tag; 101 (statement number tag).
%TBB	12-30	Index into TBB
ISE	31-47	Index into TBT secondary entry, if one exists
TBF	50-63	Tag Buffer flags
TL	64-77	Type and length; 2017 for a 24-bit ASCII value, 2077 for a 64-bit ASCII value.

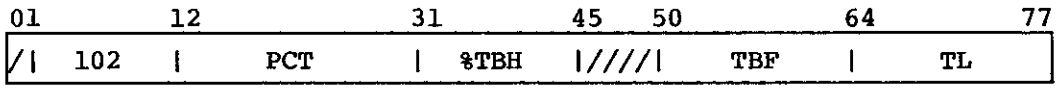
A TBT secondary entry exists when a format number appears in an ASSIGN statement. Format of a TBT secondary entry for a format number:



Field	Bits	Description
TAG	1-11	Tag; 602 (program block, positive relocation tag).
OFS	12-41	Word offset into program block

External function entry

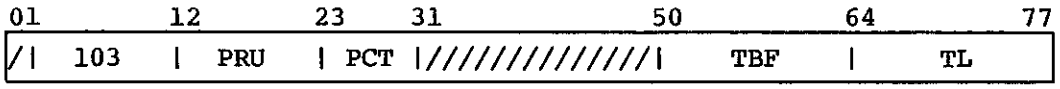
Format:



Field	Bits	Description
TAG	1-11	Tag; 102 (external function tag).
PCT	12-30	Parameter count
%TBH	31-44	%TBH of the external function
TBF	50-63	Tag Buffer flags
TL	64-77	Type and length

Intrinsic function entry header

Format:

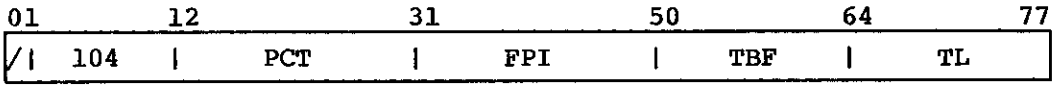


Field	Bits	Description
TAG	1-11	Tag; 103 (intrinsic function tag).
PRU	12-22	Pseudo registers used
PCT	23-30	Parameter count
TBF	50-63	Tag Buffer flags
TL	64-77	Result type and length

Statement function entry header

The statement function tag is written over immediately during Pass 1 when the expanded function is moved down in TGB at OP87.

Format:

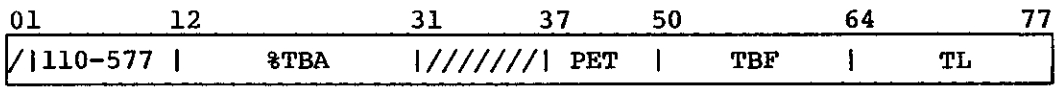


Field	Bits	Description
TAG	1-11	Tag; 104 (statement function tag).
PCT	12-30	Parameter count
FPI	31-47	Function processing information: If mode flag 61 is set, then this field contains the index to the skeleton in TBF; if it is clear, then this field contains the index to the skeleton in TBM. If mode flag 46 is set, then this field contains the word address of return for special function evaluation, for example, DO statement processing.
TBF	50-63	Tag Buffer flags
TL	64-77	Result type and length

Dummy argument entry

The Dummy Argument flag (bit 50) is set in this entry.

Format:

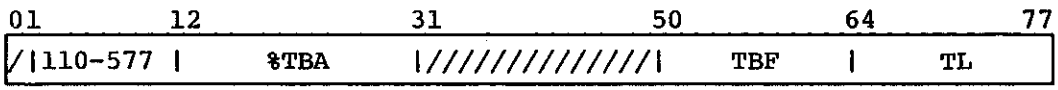


Field	Bits	Description
TAG	1-11	Tag; a number from 110 to 577 (assigned in ascending order).
%TBA	12-30	Index into TBA (for array references)
PET	37-47	Tag of dummy argument at its primary entry. This field is used only if the previous entry flag (PEF) is set, indicating that the dummy argument has appeared in the argument list of a previously encountered entry in the subprogram.
TBF	50-63	Tag Buffer flags
TL	64-77	Type and length

Pointee entry

The Dummy Argument flag (bit 50) and the EQUIVALENCE flag (bit 51) are set in this entry.

Format:



Field	Bits	Description
TAG	1-11	Tag; a number from 110 to 577 (assigned in ascending order).
%TBA	12-30	Index into TBA (for array references)

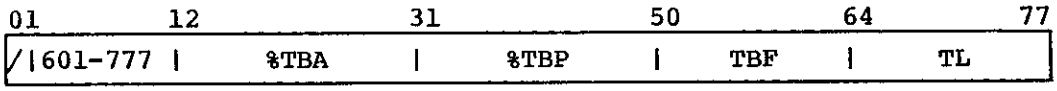
Field	Bits	Description
TBF	50-63	Tag Buffer flags
TL	64-77	Type and length

Constant entry

An entry is made in TGB for each constant encountered. Additionally, an entry is made in TBT for constants within PARAMETER statements. Refer to description of constant entry in tables section on TGB.

Program and common block entry

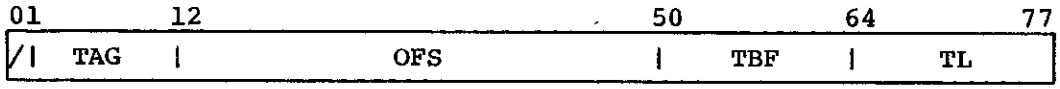
Format of TBT Pass 1 entry for a program or common block:



Field	Bits	Description
TAG	1-11	Tag; one of the following: 601 Offset from B01 (dummy argument addresses) 602 Program block, positive relocation 603 (#TB) Temporary block tag 604 (#CL) Argument list block tag 605 (#ST) Offset from B03; stack tag. 606 (#RG) B/T register tag 607 (#DA) Data block tag 610-777 Common block tag (assigned in ascending order)
%TBA	12-30	Index into TBA (for array references)
%TBP	31-47	For an equivalenced or common block tag
TBF	50-63	Tag Buffer flags
TL	64-77	Type and length

At the beginning of Pass 2, the TBT tag for an entry contains the tag (110-577) of the last dummy argument associated with the entry in bits 12-22.

Format of TBT Pass 2 entry for a program or common block:

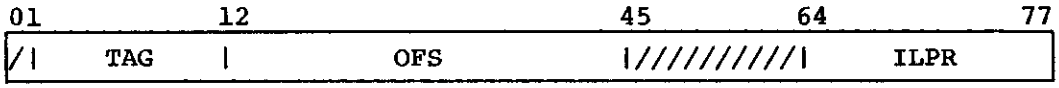


Field	Bits	Description
TAG	1-11	Tag; same as corresponding primary entry.
OFS	12-47	Offset within block specified by tag 12-41 Word offset 42-47 Bit offset within first word
TBF	50-63	Tag Buffer flags
TL	64-77	Type and length

4.TBTR TBTR - T-REGISTER ASSOCIATES TABLE

TBTR is used in Pass 2 by CSRD and CSWR for loop mode variant scalars that are referenced ahead of definition. CSRD forms the following entry.

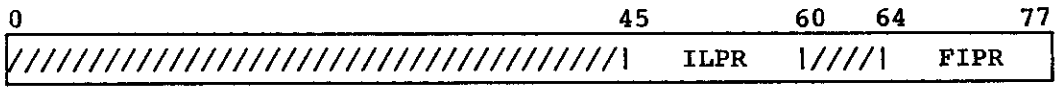
Format:



Field	Bits	Description
TAG	1-11	Tag; one of the following: 601 Offset from B01 (dummy argument addresses) 602 Program block, positive relocation 603 (#TB) Temporary block tag 604 (#CL) Argument list block tag 605 (#ST) Offset from B03; stack tag. 606 (#RG) B/T register tag 607 (#DA) Data block tag 610-777 Common block tag (assigned in ascending order)
OFS	12-44	Bit offset from the base of the block
ILPR	64-77	Initial loop pseudo register

This entry is updated by CSWR to the following entry.

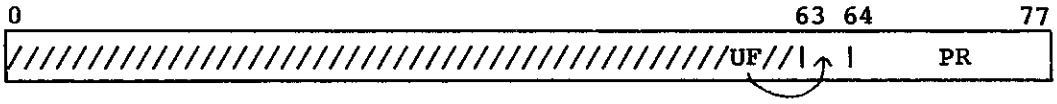
Format:



Field	Bits	Description
ILPR	45-60	Initial loop pseudo register
FIPR	64-77	Final loop pseudo register

RASN uses TBTR passed on by CSWR. It also generates the following temporary entries as required.

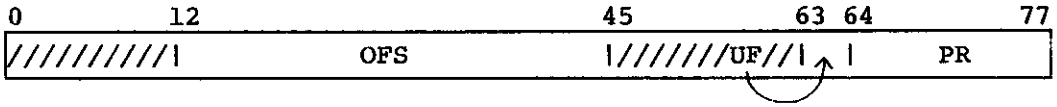
Format:



Field	Bits	Description
UF	63	Previously used flag
PR	64-77	Pseudo registers

In addition, if the temporary variables overflow the available secondary registers, the entry is as follows.

Format:



Field	Bits	Description
OFS	12-44	Bit offset from the base of the block
UF	63	Previously used flag
PR	64-77	Pseudo registers

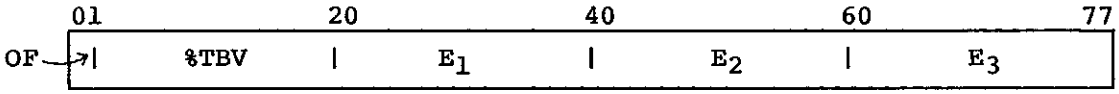
This entry gives the temporary block location assigned to the temporary variable.

4.TBU TBU - SYMBOL CROSS REFERENCE TABLE

TBU contains a cross reference to all symbolic names and statement labels used within the source program. TBV is an overflow table for TBU.

Entries in TBU are one word long. Entries are shifted into TBU from the right. If a TBU entry overflows the word, it causes an overflow flag in the TBU word to be set and a pointer to the corresponding TBV entry is inserted. The overflow information is then stored in TBV at the location pointed to by TBU.

Format:

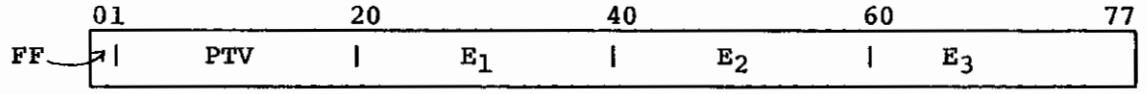


Field	Bits	Description
OF	0	Overflow flag; set if overflow entries have been made in TBV for this symbol.
%TBV	1-17	TBV pointer; contains an index to a TBV entry containing a pointer to the latest information for this symbol. If a symbol has less than four entries, this field is empty and there is no corresponding entry in TBV.
E_1-E_3	20-37, 40-57, 60-77	Symbol entries; each entry contains a usage code in its first 3 bits and the next 13 bits contain a source program line number.

4.TBV TBV - CROSS REFERENCE OVERFLOW TABLE

TBV is the overflow table for TBU. When the TBU entry has overflowed, a pointer to TBV is inserted in TBU. This pointer to TBV points to the latest TBV word for the symbol. Entries to TBV are shifted in from the right.

Format:



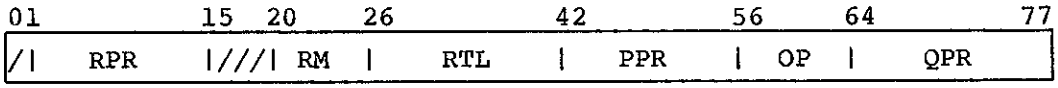
Field	Bits	Description
FF	0	First TBV entry flag; set if this word is the first TBV entry for this symbol.
PTV	1-17	Index to previous TBV word used for this symbol. If this is the first TBV word for this symbol, PTV is 0.
E_1-E_3	20-37, 40-57, 60-77	Symbol entries; each entry contains a usage code in its first 3 bits and the source program line number in the remaining 13 bits.

4.TBW TBW - TRIAD TABLE

During Pass 2, each triad (P operand - operator - Q operand) compiled is entered in TBW, together with the result type and result pseudo register. Thus, if a triad is encountered more than once during compilation, it is only compiled once.

TBW is cleared at the beginning of each new block that does not have a drop-through entry.

Format:

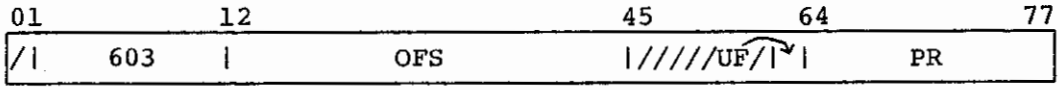


Field	Bits	Description
RPR	1-14	Result pseudo register
RM	20-25	Result mode
RTL	26-41	Result type and length
PPR	42-55	P operand pseudo register
OP	56-63	Operator (see section 4.TGB for a description of operator representations). A null entry indicates type conversion or register transfer.
QPR	64-77	Q operand pseudo register, null if type conversion or register transfer

4.TBWR TBWR - W-REGISTER ASSOCIATES TABLE

TBWR acts as an overflow area for the vector register backups, the W registers. It is a fixed length table.

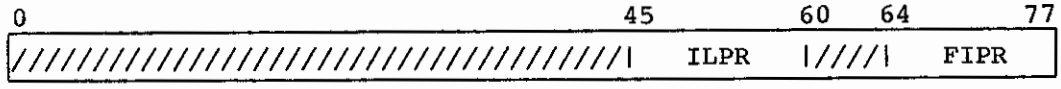
Format:



Field	Bits	Description
TAG	1-11	Tag; 603 (temporary block tag
OFS	12-44	Bit offset from the base of the block
UF	63	Previously used flag
PR	64-77	Pseudo registers

This entry is updated by CSWR to the following entry.

Format:



Field	Bits	Description
ILPR	45-60	Initial loop pseudo register
FIPR	64-77	Final loop pseudo register

4.TBX TBX - VARIABLE REFERENCE TABLE

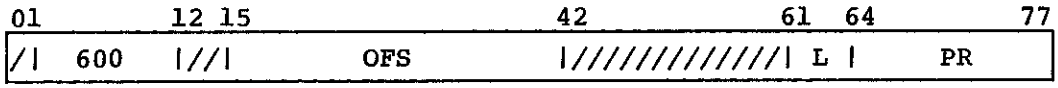
During Pass 2, TBX contains entries for any variable whether scalar or vector. For an array reference, TBX contains the index pseudo register.

Compiled instructions contain the index to TBX. At the primary instruction level, the index is converted using the tag offset field. Information regarding the common block or the program block is contained in the tag entry.

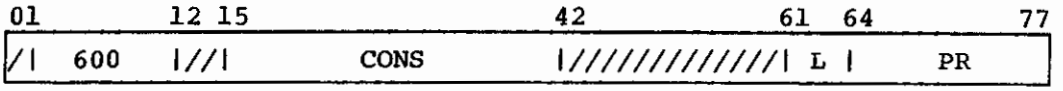
TBX is used by CSRD and CSWR, the compile scalar read and write routines, to keep track of what has been loaded and stored from the pseudo registers. A section of code within CBLK also uses the information in TBX when doing vector array loads and stores.

TBX is cleared at the beginning of each new block that does not have a drop-through entry.

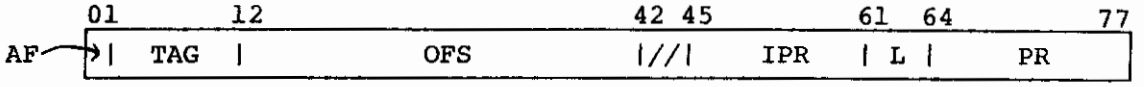
Format:



Field	Bits	Description
TAG	1-11	Tag; 600 (constant tag).
OFS	15-41	Word offset into block for constant
L	61-63	Length of operand (bits) / L value: 30 / 0; 100 / 2; 200 / 6
PR	64-77	Pseudo register assigned to the constant



Field	Bits	Description
TAG	1-11	Tag; 600 (constant tag).
CONS	15-41	22-bit immediate constant
L	61-63	Length of operand (bits) / L value: 30 / 0; 100 / 2; 200 / 6
PR	64-77	Pseudo register assigned to the constant

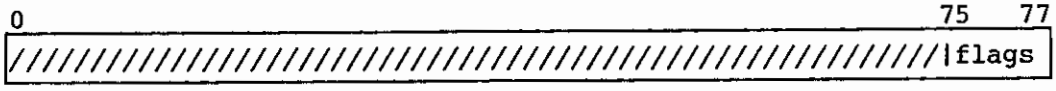


Field	Bits	Description
AF	0	Address flag; if set, PR is the pseudo register containing the address of the tag.
TAG	1-11	Tag
OFS	12-41	Word offset within the block
IPR	45-60	Index pseudo register if entry is an array reference: 7774 EMA common block base indicator 7775 Task common block base indicator 7776 SKED indicator 7777 Obsolete definition indicator
L	61-63	Length of operand (bits) / L value: 30 / 0; 100 / 2; 200 / 6
	63	Subsequent Ambiguous Reference flag
PR	64-77	Pseudo register assigned to the constant

4.TBXX TBXX - TBX EXTENSION TABLE

TBXX is an extension of TBX (Variable Reference Table). See subsection 4.TBX for a detailed description of TBX.

Format:



Field	Bits	Description
Flags:	75-77	
XCTG	75	Conditional definition of tag
XRMV	76	Conditional definition of sub-block
XSVE	77	Save definition at sub-block boundary

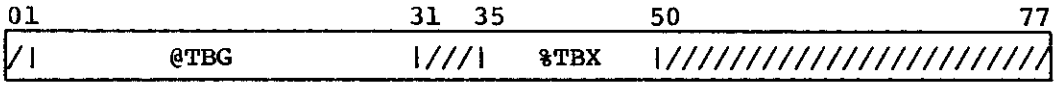
4.TBY TBY - DEPENDENT REFERENCE TABLE

During Pass 2, TBY contains pointers to definitions contained within the block and all references relative to each definition. Each definition in the block has an entry, followed by an entry for each reference to that entry.

Routine ADEP builds TBY for use by SKED, the instruction scheduling routine. TBY tells SKED how far it can move instructions, particularly loads and stores. TBY is also used by CSRD and CSWR (the compile scalar read and write routines) to resolve loads and stores.

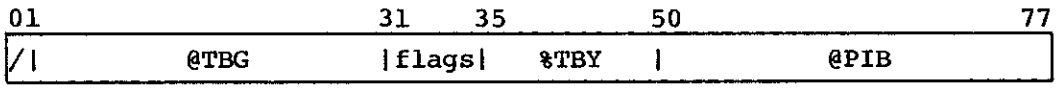
TBY is cleared at the beginning of each new block that does not have a drop-through entry.

Format of a definition entry:



Field	Bits	Description
@TBG	1-30	Address of TBG entry
%TBX	35-47	Index into TBX entry

Format of a reference entry:



Field	Bits	Description
@TBG	1-30	Address of TBG entry
Flags:	31-34	
DEPDEF	31	Reference/definition (0/1)
DEPSUB	32	Previous/subsequent (0/1) relative to the definition entry
DEPMIN	33	Minus dependency. Entry referenced in earlier loop iteration instead of definition entry. If entry is not in loop iteration, it is in storage before definition entry.

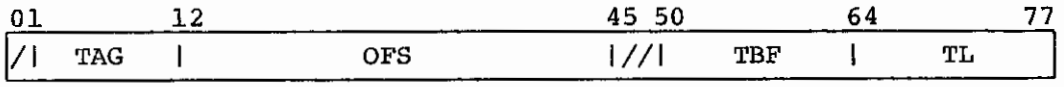
Field	Bits	Description
DEPPLS	34	Plus dependency. Entry referenced in later loop iteration instead of definition entry. If the entry is not in the loop iteration, it is in storage after definition entry.
%TBY	35-47	Offset from TBY definition entry to TBY reference entry
@PIB	50-77	Address of PIB entry

4.TBZ TBZ - DEFINED VARIABLE TABLE

During Pass 2, each variable that is defined in a block is entered in TBZ. Each entry consists of two words. The first word, word 0, contains the variable tag. The second word, word 1, contains the tag location index, tag definition index, and a number of flags pertaining to constant integer analysis.

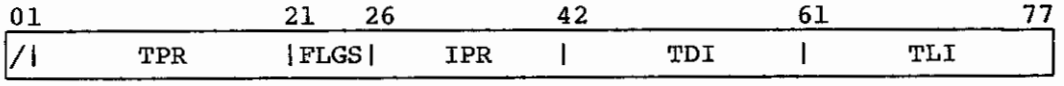
TBZ is cleared at the beginning of each new block that does not have a drop-through entry.

Format:



Field	Bits	Description
TAG	1-11	Tag
OFS	12-44	Offset
TBF	50-63	Tag Buffer flags (refer to 4.TGB)
TL	64-77	Type and length

Format:



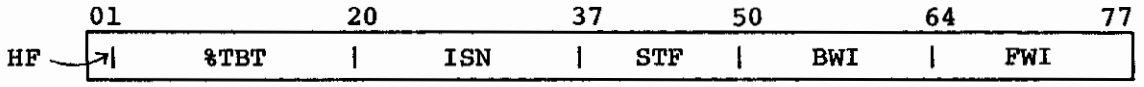
Field	Bits	Description
TPR	1-20	Tag pseudo register
FLGS	21-25	Flags, as follows:

Bit	Description
21	Increment type; clear for 24-bit integer, set for 64-bit integer
22	Trip Count flag
23	Self-reference flag

Field	Bits	Description
		Bit Description 24-25 Increment sign, as follows: Value Description 00,01 Ambiguous 10 Plus 11 Minus
IPR	26-41	Increment pseudo register
TDI	42-60	Tag definition index. For a replacement definition, TDI equals the block index of beginning of next statement in TBG; otherwise, TDI=TLI.
TLI	61-77	Tag location index; equal to block index of tag location in TBG

4.TGB TGB - TAG BUFFER

During Pass 1, a TGB statement entry begins with a statement header word. This header word has its sign bit set, whereas none of the entry words do; the header word's sign bit is set so that during Pass 2, where things are processed on a statement-by-statement basis, a quick search will locate the head of each statement unit. The statement header word has the following format:



Field	Bits	Description
HF	0	Flag set to indicate header word
%TBT	1-17	Index into statement number entry in TBT, if one exists. If header is for an entry (ENF set), the field is the index into TBH of the entry name.
ISN	20-36	Internal sequence number in binary, as it appears on FORTRAN source listing on left margin
STF	37-47	Statement type flags, as follows:

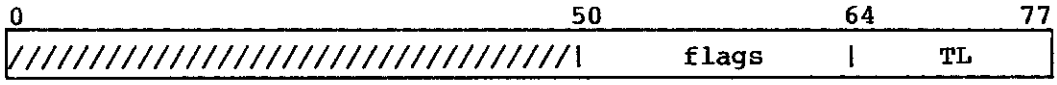
Bit	Flag	Description
37	ENF	ENTRY statement
40	DBF	Beginning of a DO
41	CSF	Conditional statement
42	RPF	Replacement statement
43	CAF	CALL statement
44	ISF	IF statement
45	GTF	GO TO statement
46	CNF	CONTINUE statement
47	IDF	Ignore Vector Dependency flag

If the header is for an entry (ENF set), bits 40-47 give the number of arguments associated with this entry.

BWI	50-63	Backward offset relative to the location of this header to previous statement header in TGB
FWI	64-77	Forward offset relative to the location of this header to next statement header in TGB

Following the header word for a statement is a 1-word entry for each of the elements in the statement. This entry can be a tag, an operator, or a separator. A tag is derived from the TBT entry for the corresponding symbol. It contains an index to the TBT entry. Operators and separators are translated to 6-bit codes that reflect processing precedence.

Tag Buffer flags are as follows:



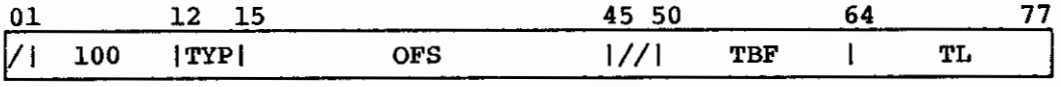
Field	Bit	Description
Flags:	50-63	
DAF	50	Dummy Argument flag
EQF	51	Equivalence flag If DAF and EQF are set, the tag is a pointer reference
FNF	52, 53-55	Function flag; if flag is set, the symbol name is a function and bits 53-55 are as follows. 53 Call-by-value flag (CBV) 54 Function/subroutine Call flag (FSC) 55 Single/multiple Result Function flag (SMR) If flag is clear and bits 53-55 are zeros, then the symbol name is a simple variable. Otherwise, the symbol name is an array; bit 52 is clear and bits 53-55 contain the number of dimensions in the array (up to 7). If bits 51 and 52 are set, the function has side effects and cannot be optimized.
INF	56	Internal Statement Function flag or
RDF	56	Defined flag; set if variable is defined (assigned a value). This bit is set on a block-by-block basis.
IVF	57	Variant/invariant flag; set if variable is a variant within the block. This bit is set on a block-by-block basis.
CIF	60	Constant Increment Integer (CII) flag
VAF	61	Vector Array flag or Function flag; set if this array reference or function call can be vectorized. Vectorization is possible if the

Field	Bit	Description
VAF (continued)		subscripting is acceptable, if the vector array or function is in a vectorizable form, or if it is a known vector library routine with a vector argument.
SAF	62	Intrinsic function special processing bit; bit is checked only in function headers with VAF set. If SAF is also set, a call is made to SPFH from CBLK and to SPFR to handle special processing for intrinsic functions such as SHIFT and CSMG.
		or
KSF	62	Known Sign flag; set if the sign of a number is known. Used for constant tags (600) only.
MAF	63	Sign bit, if known (if bit 62 is set)
		or
SCF	63	Subsequent ambiguous reference
TL	64-77	Type and length

Pseudo entry

A pseudo tag is used to represent a temporary value which is not to be allocated a memory location. The pseudo tag (100) is used only in TGB and TBG. Fields conform to function and variable. Pseudo entry for a function is only used for RETURN statements, while that for a variable is used throughout.

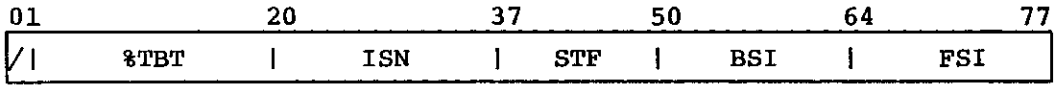
Format:



Field	Bits	Description
TAG	1-11	Tag; 100 (constant tag).
TYP	12-14	Constant tag subtype
OFS	15-44	Offset
TBF	50-63	Tag Buffer flags
TL	64-77	Type and length

Statement number definition entry

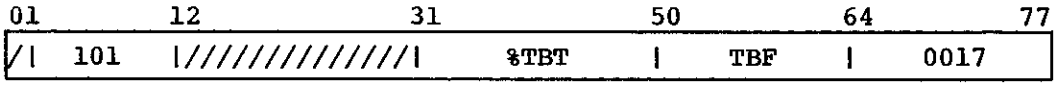
Format:



Field	Bits	Description
%TBT	1-17	Index into TBT
ISN	20-36	Internal sequence number
STF	37-47	Statement Type flags
BSI	50-63	Backward statement index
FSI	64-77	Forward statement index

Statement number reference entry

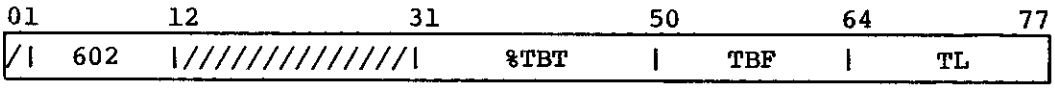
Format:



Field	Bits	Description
TAG	1-11	Tag; 101 (statement number tag).
%TBT	31-47	Index into TBT
TBF	50-63	Tag Buffer flags
TL	64-77	Type and length; 0017.

Format number reference entry

Format:

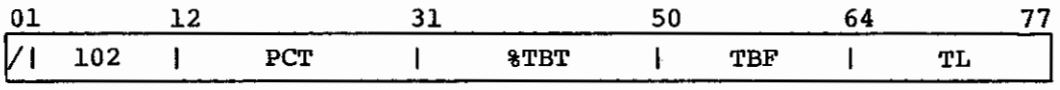


Field	Bits	Description
TAG	1-11	Tag; 602 (program block, positive relocation tag).
%TBT	31-47	Index into TBT
TBF	50-63	Tag Buffer flags
TL	64-77	Type and length; 2017 for 24-bit ASCII value, 2077 for 64-bit ASCII value.

External function entry

(An external intrinsic function has an entry in TBL and TBM.)

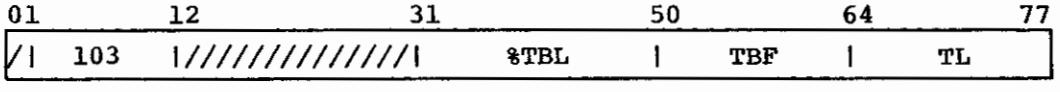
Format:



Field	Bits	Description
TAG	1-11	Tag; 102 (external function tag).
PCT	12-30	Parameter count; number of arguments to the function (taken from TBM).
%TBT	31-47	Index into TBT
TBF	50-63	Tag Buffer flags
TL	64-77	Type and length

Intrinsic function entry

Format:

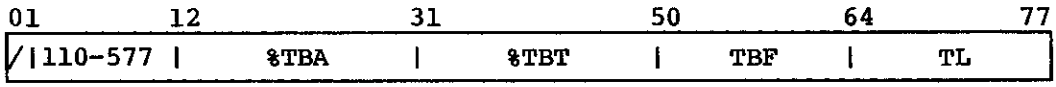


Field	Bits	Description
TAG	1-11	Tag; 103 (intrinsic function tag).

Field	Bits	Description
%TBL	31-47	Index into TBL entry
TBF	50-63	Tag Buffer flags
TL	64-77	Type and length

Dummy argument entry (Pass 1)

Format:



Field	Bits	Description
TAG	1-11	Tag; a number from 110 to 577 (dummy argument tag).
%TBA	12-30	Index into TBA (for array references)
%TBT	31-47	Index into TBT
TBF	50-63	Tag Buffer flags
TL	64-77	Type and length

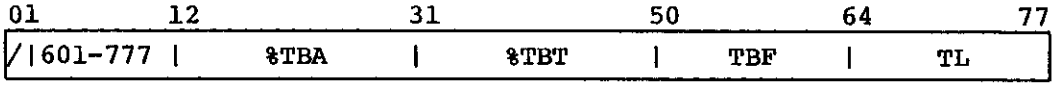
Dummy argument entry (Pass 2)

Format:

Field	Bits	Description
TAG	1-11	Tag; a number from 110 to 577 (dummy argument tag).
OFS	12-47	Offset in block
TBF	50-63	Tag Buffer flags
TL	64-77	Type and length

Variable in program and common block (Pass 1)

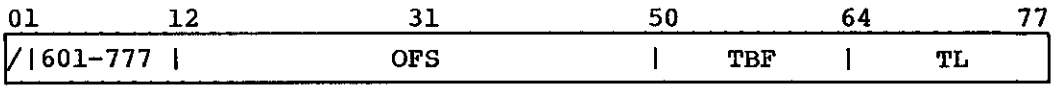
Format:



Field	Bits	Description
TAG	1-11	Tag; one of the following: 601 Offset from B01 (dummy argument addresses) 602 Program block, positive relocation 603 (#TB) Temporary block tag 604 (#CL) Argument list block tag 605 (#ST) Offset from B03; stack tag. 606 (#RG) B/T register tag 607 (#DA) Data block tag 610-777 Common block tag (assigned in ascending order)
%TBA	12-30	Index into TBA (for array references)
%TBT	31-47	Index into TBT
TBF	50-63	Tag Buffer flags
TL	64-77	Type and length

Variable in program and common block (Pass 2)

Format:



Field	Bits	Description
TAG	1-11	Tag; one of the following: 601 Offset from B01 (dummy argument addresses) 602 Program block, positive relocation 603 (#TB) Temporary block tag 604 (#CL) Argument list block tag 605 (#ST) Offset from B03; stack tag. 606 (#RG) B/T register tag

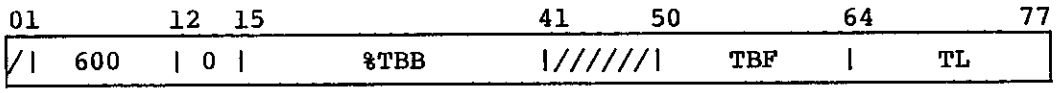
Field	Bits	Description
TAG (continued)		607 (#DA) Data block tag 610-777 Common block tag (assigned in ascending order)
OFS	12-47	Offset in block
TBF	50-63	Tag Buffer flags
TL	64-77	Type and length

Constant entry

An entry is made in TGB for each constant encountered. Four constant tag subtypes are available. Their formats follow.

A subtype 0 constant tag gives TBB entry information.

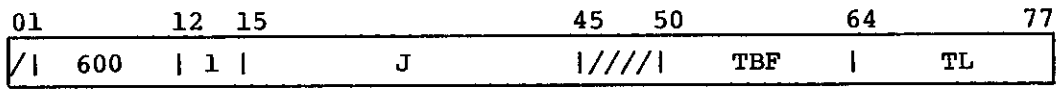
Format:



Field	Bits	Description
TAG	1-11	Tag; 600 (constant tag).
SUB	12-14	Constant tag subtype; 0 (constant in TBB).
%TBB	15-41	Index into TBB
TBF	50-63	Tag Buffer flags
TL	64-77	Type and length

A subtype 1 constant tag signals a machine-generated constant. (The 071 machine instruction allows generation of a number of different constants.)

Format:



Field	Bits	Description
TAG	1-11	Tag; 600 (constant tag).
SUB	12-14	Constant tag subtype; 1 (machine-generated constant).
J	15-44	J holds a value that corresponds to the opcode's J value as follows:

Opcode	Constant	J value
071i4x	0.5	4
071i5x	1.0	5
071i6x	2.0	6
071i7x	4.0	7

TBF	50-63	Tag Buffer flags
TL	64-77	Type and length

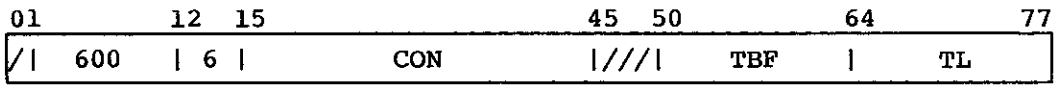
A subtype 4 constant tag is used for a 22-bit immediate constant.

Format:

Field	Bits	Description
TAG	1-11	Tag; 600 (constant tag).
SUB	12-14	Constant tag subtype; 4 (immediate).
CON	15-44	22-bit immediate constant (preceded by 2 sign bits)
TBF	50-63	Tag Buffer flags
TL	64-77	Type and length

A subtype 6 constant tag contains a 22-bit intermediate shifted constant. The 22-bit constant is shifted left 51 places. This subtype is used to generate floating-point constants.

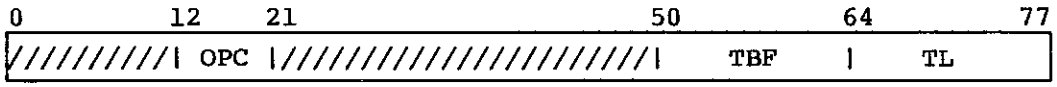
Format:



Field	Bits	Description
TAG	1-11	Tag; 600 (constant tag).
SUB	12-14	Constant tag subtype; 6 (shifted constant).
CON	15-44	22-bit intermediate shifted constant
TBF	50-63	Tag Buffer flags
TL	64-77	Type and length

Operator entry

Format:



Field	Bits	Description
OPC	12-20	TGB representation of operator or separator in 6-bit code form. OPC codes are given in table 4.TGB-1.
TBF	50-63	Tag Buffer flags
TL	64-77	Type and length

The following bits are used when the OPC field is 53, 54, or 55.

Field	Bits	Description
	20-37	Size of the parenthesis group if OPC is 53 or 54
	34-47	%TBW of subscript increment if OPC is 53 or 54 Number of arguments if OPC is 53 or 54 Number of subscripts if OPC is 53, 54, or 55 Argument number if OPC is 53, 54, or 55

Field	Bits	Description
PMF	50-63	Parenthesis Mode flag if OPC is 53, 54, or 55. The description of the bits follows:

Bit	Flag	Description
53	STF	Substring
	CBV	If PLF is set, bit 53 is the call-by-value parameter list
54	RTF	Required type
55	DSF	Data statement/variable subscript
56	IOF	I/O list
57	SNF	Statement number
60	IFF	IF statement
61	PLR	Parameter list
62	SSF	Subscript flag
63	EXF	Expression flag

Table 4.TGB-1. TGB representation of operators and separators

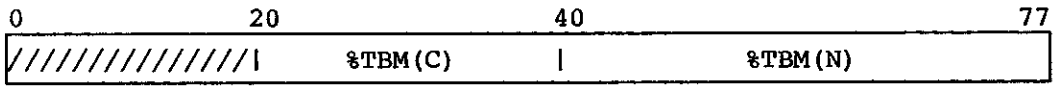
Code	Description	Precedence Level
77	Arithmetic exponentiation	16
76	Arithmetic quotient	15
75	Arithmetic product	15
74	Arithmetic difference	14
73	Arithmetic sum	14
72	Character concatenation	13
71	.EQ.	12
70	.NE.	12
67	.LE.	12
66	.GT.	12
65	.GE.	12
64	.LT.	12
63	.NOT. Logical complement	11
62	.AND. Product	10
61	.OR. Sum	7
60	.XOR. Difference (.NEQV.)	6
57	.EQV. Equivalence	6
56	Colon	5
55	Comma	4
54	Close parenthesis	3
53	Open parenthesis	2
52	Equal sign	1

4.TSIF TSIF - SPECIAL INTRINSIC FUNCTION TABLE

TSIF appears in CB near CB35B to cause special intrinsic functions to be preprocessed by CBLK before the call to INFN. For example, the external value functions are processed as repeated calls to two argument versions including INFN as follows.

```
MAX0(I,J,K) = MAX0(I,MAX0(J,K))
```

Format:



Field	Bits	Description
%TBM(C)	20-37	Index into TBM of 2-argument version of function with type conversion from arguments to result (integer to floating-point or vice versa)
%TBM(N)	40-77	Index into TBM of 2-argument version of function with result type equal to argument type

SUBROUTINES 5

This section includes brief descriptions of the major subroutines that comprise the CFT compiler. These routines are listed in alphabetical order according to routine name.

ROUTINE: ABLK - Analyze block

PASS: 2

DESCRIPTION: ABLK is the main driver for Pass 2. It divides TBG into segments, called blocks, which are then compiled into code, one at a time.

During Pass 1, all extraneous statement numbers are deactivated. Thus, at the beginning of Pass 2, each statement number in TBG is a target.

Each time ABLK is called, it searches through TBG to find the next block boundary. A block boundary is an ENTRY statement, a loop begin, or a statement referenced from outside the block.

ABLK does the following:

- Builds TBZ, the list of variables defined within the block
- Sets the variant bit in all references to variant elements, including variables used in EQUIVALENCE statements and common or dummy arguments (if there is an external reference to them)
- Locates all constant increment integer (CII) variables and creates a TBZ entry for each CII

The ABLK sequence is:

1. ABLK
2. Analyze dependency conditions (ADEP)
3. Promote constants from subscripts (PCON)
4. Set Vector Array flag in all probable scalar temporary vectors (SVEC)
5. Examine array or function references (EAFR)
6. Vector loop control (VCTL)
7. Compile block (CBLK)

ROUTINE: ABRA - Analyze branch statements

PASS: 1 (during END statement processing)

DESCRIPTION: ABRA is called from EN30 at the end of Pass 1 before Pass 2 begins. The intermediate text is built and statement labels and references to the text are linked together before ABRA is called. ABRA examines all statement numbers to see if they are defined in the range of a DO-loop. If they are in the range of a DO-loop, all references to the statement number are examined. Each reference outside the loop where the statement label is defined causes ABRA to issue a warning message if the reference is a conditional, unconditional, or computed GO TO; two or three branch arithmetic IF; or an alternate return subroutine call. A caution message is issued if a statement number defined in a loop is referenced in an ASSIGN statement or in the END= or ERR= branch of an I/O statement.

ROUTINES: ACAL, ACAN, ACGR, ACNU - Assemble character string

PASS: 1

DESCRIPTION: These routines assemble a character string in register S1, left-adjusted and packed 8 characters per word. The string is gathered from a character string, generally the Character Buffer (CHB).

The routines and the string types they handle are:

- ACAL - Alphabetic characters
- ACAN - Alphanumeric characters
- ACGR - General strings; user specifies list of characters allowed.
- ACNU - Numeric characters

A routine continues assembling a string until it encounters a character not in the appropriate group. Control transfers to the appropriate routine.

These routines use a special technique possible on the CRAY-1 computer for determining whether or not an ASCII character is a member of a specific group.

DESCRIPTION: (continued)

ASCII characters are 8-bit characters; however, only the low-order 7 bits are used. Thus, there are 128 possible ASCII characters. Two S registers are treated as if they are concatenated to form a 128-bit string. Two masks are entered into these registers with ones in the bit position corresponding to an ASCII character being searched for. The position of the character to be examined is then loaded into an A register and a double length concatenated shift is performed on the two S registers, putting the result into S0. If S0 is negative, then the character being examined is a member of the set being looked for; if S0 is positive, the character is not a member.

These routines can assemble a symbol of up to eight characters. Compatibility and transportability can be enforced by inserting code into these routines to limit symbol lengths to six or seven characters.

ROUTINE: ADEP - Analyze dependency conditions

PASS: 2

DESCRIPTION:

ADEP examines array references within a block, looking for vector potential. ADEP's function is to analyze the dependency of array references and to identify characteristics inhibiting vectorization. It marks both actual and ambiguous dependencies. If a dependency is found, ADEP clears the Vector Loop flag.

ADEP also sets up TBY, the Dependent Reference Table, which is used in optimization.

Clearing the Vector Loop flag occurs in S7 under one of the following conditions.

- When the Reference, Previous, and Minus flags are set
- When the Define, Previous, and Minus flags are set

Constant stride analysis determines if a detected dependency can be ignored because array references never access the same memory locations. A(I) and A(I-1) never access the same memory locations if the stride through the DO-loop is 2.

DESCRIPTION: (continued)

Analysis continues even if the Vector Loop flag has been cleared in ADEP because dependency cases are important for other optimizations. The Vector Array flag may be set for other array tags, but a vector read does not occur for those arrays unless the Vector Loop flag is also set.

ROUTINE: AIBF - Analyze internal block flow

PASS: 2

DESCRIPTION: AIBF analyzes the flow of control within a block selected by ABLK. BOF should be marked if the flow transfers out of the block. AIBF also determines which statements are executed each time and the extent of the loop, if any. If the block is a loop, set LPF (and VLF and DLF, if appropriate).

ROUTINE: ARUS - Analyze register usage

PASS: 2

DESCRIPTION: APWD counts usage of each register type.

ROUTINE: ASST - ASSIGN statement processor

PASS: 1

DESCRIPTION: This routine processes ASSIGN *xx* TO *yy* statements.

ROUTINE: ASVL - Assign short-loop registers

PASS: 2

DESCRIPTION: ASVL attempts to assign recursion registers to real registers for the life of the loop. ASVL also extracts instructions to be bottom-loaded and sets up the bottom load.

ROUTINE: ASVM - Assign loop boundaries

PASS: 2

DESCRIPTION: ASVM determines the boundaries of the IF-loop and sets up the preamble for bottom-loading the IF-loop.

ROUTINE: AT*xx* - Assemble Tag Buffer

PASS: 1

DESCRIPTION: This routine is the main driver in assembling the Tag Buffer (TGB). AT*xx* works along with OP*xx* for Pass 1 syntax checking.

AT02 performs the syntax checking for all expressions. It compares the previous tag or operator with the current tag or operator and a precedence mask for the current tag or operator to determine whether the sequence is legal.

AT05 begins processing of the next element. Three element types are possible: Hollerith text, operator or separator, and alphabetic or numeric. If the code at AT05 detects Hollerith notation, it branches to routine HOLD, the Hollerith text processor; if it encounters an operator, it branches to OP02 to process that operator. However, alphabetic and numeric fields are processed by AT*xx* itself. An alphabetic field is converted to a tag. Then, the next character is examined. If it is an open parenthesis for a subscript or an argument, list processing is set up. A numeric item can be either a number or a statement label.

AT*xx* processes both expressions and lists (for example, I/O lists or argument lists). If the array bounds feature is enabled, AT*xx* transforms array references into function calls.

AT*xx* also sets the referenced entry flag and/or the defined entry flag for tags.

After AT38, there are several entries to initialize for special statement processing.

ROUTINE: BDST - BLOCK DATA statement processor

PASS: 1

DESCRIPTION: Routine BDST processes BLOCK DATA statements and enters SRST at SR04.

ROUTINE: BFST - BUFFER IN and BUFFER OUT statement processor

PASS: 1

DESCRIPTION: This routine processes BUFFER IN and BUFFER OUT statements. It also does some syntax checking and branches to routine IOST (at I066).

ROUTINE: BGIN - Begin compilation

PASS: 1

DESCRIPTION: Routine BGIN initializes CFT and calls the control statement cracking routine, CARD.

The code beginning at BG10 performs partial initialization at the start of Pass 1 for each block compiled.

ROUTINE: BKST - BACKSPACE statement processor

PASS: 1

DESCRIPTION: Routine BKST is the BACKSPACE statement processor. Upon recognition of 'BACKSPACE', control is passed to RW01 in RWST for further processing.

ROUTINE:	BLCN - Blank count
PASS:	1,2
DESCRIPTION:	Routine BLCN returns the number of bits occupied by the leading blanks in the right-justified, blank-filled, 8-bit ASCII symbol in S1. The caller can use this bit count to left-justify the symbol.

ROUTINE:	BLFL - Blank fill a word
PASS:	1,2
DESCRIPTION:	This routine blank fills an ASCII word in register S1.

ROUTINE:	BTD - Convert binary value to ASCII decimal value
PASS:	1,2
DESCRIPTION:	BTD converts a binary value (S1) to the ASCII representation of its decimal value (returned in S1). The contents of A0-A7 and S7 are saved, the external library routine $BTD does the conversion, and A0-A7 and S7 are restored.

ROUTINE:	CADR - Compile address
PASS:	2
DESCRIPTION:	CADR compiles the address of a tag. It compiles the code needed to load an address into an A register.

ROUTINE: CADW - Compile dummy argument address

PASS: 2

DESCRIPTION: Routine CADW compiles code to copy a dummy argument address from a secondary entry to the primary entry.

ROUTINE: CARD - Crack CFT control statement

PASS: 0

DESCRIPTION: CARD is called as part of the initialization process performed by routine BGIN. It cracks the CFT control statement, determines the datasets needed, and opens them if they are not already open. CARD also collects the list and error processing options specified on the CFT control statement, setting indicator bits in register T.OCW showing which options are ON and OFF. Default values are used for options not specified on the CFT control statement.

ROUTINE: CBLK - Compile block

PASS: 2

DESCRIPTION: Compile block. This routine sequences through a block line-by-line and generates code. On each line, CBLK locates the innermost parentheses pair and drives routine OLEV through the operator precedence list to compile code (PTRI). CBLK also processes function calls, transmits argument addresses, broadcasts scalar arguments, expands intrinsic functions (calling routine INFN), and forces expression type conversion, if necessary.

The code beginning at CB70 processes vector array loads and stores.

ROUTINE: CCAT - Compile concatenation

PASS: 2

DESCRIPTION: CCAT compiles concatenation by taking a sequence of character operands and generating temporary storage and calls for the run time routines doing the concatenation.

ROUTINE: CCLA - Construct character operand address

PASS: 2

DESCRIPTION: CCLA generates code constructing a character operand address and transmits the address to another routine.

ROUTINE: CCLO - Convert character constant

PASS: 1

DESCRIPTION: CCLO converts a character constant to a Boolean operand when a character operand is illegal.

ROUTINE: CCRS - Convert conditional replacement statement

PASS: 1

DESCRIPTION: CCRS optimizes conditional replacement statements of the form IF(*logical expression*)*var=expression* by producing code similar to the replacement statement *var*=CVMGT(*expression,var,logical expression*). A new tag buffer entry is created from the conditional replacement statement entry and is copied over that entry in the tag buffer.

DESCRIPTION: (continued)	Certain conditions must be met before optimization occurs. Optimization must first be enabled by specifying OPT=FULLIFCON or OPT=PARTIALIFCON on the CFT control card. (Optimization is disabled by specifying OPT=NOIFCON on the CFT control card (default) or using the CDIR$ NOIFCON compiler directive.) If the optimization level is partial-IF conversion (OPT=PARTIALIFCON), the replacement expression cannot involve division or an external function reference. The last requirement for optimization is that the type of the replacement variable must be integer, logical, or real.

ROUTINE:	CCTB - Convert character constant operand
PASS:	2
DESCRIPTION:	CCTB converts a character constant operand to a Boolean operand when the character constant operand is used as an arithmetic operator operand.

ROUTINE:	CDIR - Compiler directive processor
PASS:	1
DESCRIPTION:	CDIR processes the CDIR$ directives during Pass 1.

ROUTINE:	CDPR - Compiler directive processor
PASS:	2
DESCRIPTION:	CDPR processes the ALIGN, BL, BLOCK, CODE, CVL, FASTMD, NOBL, NOCODE, NOCVL, NODOREP, NOIFCON, NORECURRENCE, NOVECTOR, RESUMEDOREP, RESUMEIFCON, ROLL, SAFEIF, SLOWMD, and UNSAFEIF compiler directives during Pass 2.

ROUTINE: CE*xx* - Check EQUIVALENCE overlap

PASS: 2

DESCRIPTION: These routines check for EQUIVALENCE overlap, looking for EQUIVALENCE dependencies between two tags.

CE*xx* checks all references against the definitions in TBZ. CEOV selects exact matches for the references in TBZ and then begins an analysis of the subscripts. Because all constants are packed together at the end of the subscript expression during Pass 1, all scans proceed from right to left. If the constant add-ins do not match, a flag is set.

ROUTINE: CEXP - Constant expression evaluation

PASS: 1,2

DESCRIPTION: CEXP examines the tag buffer for a simple constant expression inside parentheses. If the tag buffer contains ($constant_1$ *operator* $constant_2$) where the constant tags are of the form 06004... and the operator must be KAP, KAS, or KAD, this expression is evaluated to a single constant and stored back into the tag buffer.

ROUTINE: CFBI - Correct forward and backward indices

PASS: 1,2

DESCRIPTION: CFBI examines a section of the tag buffer to ensure that all statement header forward and backward indices are correct.

ROUTINE: CHST - CHARACTER statement processor

PASS: 1

DESCRIPTION: This routine sets the appropriate type and length fields in S7, verifies the spelling of CHARACTER, and jumps to routine DCLR. Control returns to CH10 from DC02 if the character length must be evaluated. After the length evaluation, control reverts to DCLR.

ROUTINE: CIDN - Check identifier names

PASS: End of Pass 1

DESCRIPTION: CIDN is called at the end of Pass 1 only if non-ANSI messages are enabled. A pass is made through TBS to examine identifiers with lengths greater than 6 characters and to examine symbolic constant names used as common block names. Both cases are flagged with non-ANSI messages.

ROUTINE: CKRF - Examine IF statements

PASS: 1

DESCRIPTION: CKRF examines IF statements for replacement by a MAX or MIN function.

ROUTINES: CLAT, CLCF, CLGA, CLOF, CLOG, CLRS, CLSZ, CLTG - Process external function and subroutine calls

PASS: 2

DESCRIPTION: These routines are primarily used by CBLK to process external function and subroutine calls.

DESCRIPTION:
(continued)

Argument lists are built in block #CL in static mode. Argument lists are built on the stack in stack mode (see Appendix D for the stack frame format). Static and stack modes have argument list headers built as compile-time constants in #CL. The argument list header for a call by address in stack mode is read from #CL and transferred to the stack at run time.

The CLSZ function is used between Passes 1 and 2 to compute the space needed in #CL. CLRS reserves space in #CL.

Processing a call begins with CLOF or CLOG. CLOF is invoked for calls in TBG and CLOG is invoked for by-value function calls created in Pass 2. Both routines reserve #CL and stack space needed for the call, construct the arglist header in #CL, and save call information used by CLAT, CLGA, and CLTG. Saved call information is stored on a compile-time stack and includes: the amount of #CL space reserved for the call, a pointer to the arglist header in #CL, a pointer to the call list in TBG, the amount of space reserved for the argument list, a pointer to the arglist header slot in the run time stack (in stack mode), and a pseudo register with the base address of the argument list (for calls by address).

All the information for a call may not be known when CLOF or CLOG is invoked. For example, CBLK creates an address-of-result argument for character functions after CLOF is complete. CLGA makes adjustments to process these generated arguments.

When a call is being processed, CLAT and CLTG are invoked to obtain tags for argument list entities. CLTG returns a tag for an entity in #CL and is usually used to obtain a tag for the arglist header. CLAT returns a tag for an arglist entity in the form of an offset from a base-address pseudo register and is usually used to generate code which stores an argument address in the argument list. The type of tag returned by CLAT can be used in stack and static modes and therefore, most CBLK call-handling code ignores the mode in use.

Invoke CLCF to complete call processing. CLCF pops the compile-time stack which reactivates the processing of an enclosing call, if any.

ROUTINE: CLOS - CLOSE statement processor

PASS: 1

DESCRIPTION: This routine processes CLOSE statements. Most processing takes place in OPEN.

ROUTINE: CLST - CALL statement processor

PASS: 1

DESCRIPTION: CLST processes CALL statements. If alternate returns are present, CLST branches to GT10 to process as a computed GO TO; otherwise, it branches to AT44 to process the argument list.

ROUTINE: CMST - COMMON statement processor

PASS: 1

DESCRIPTION: CMST processes COMMON statements and then prepares for DCLR. CMST also processes TASK COMMON statements. Control transfers to CMST when the TASK keyword is used. If the syntax passes and a task common block was declared, CMST places the necessary information in a static table (COMTAG). COMTAG is defined at the end of CMST and contains additional common block attribute information.

ROUTINE: CNST - CONTINUE statement processor

PASS: 1

DESCRIPTION: CNST processes CONTINUE statements. Processing of a statement number (typically a DO terminator) is done at STTR.

ROUTINES: CNTG, CNTB, CNTD - Convert constant to tag

PASS: 1

DESCRIPTION: These routines convert a constant stored in registers S1 and S2 into a tag. They then enter the value into a table or generate an immediate constant tag.

CNTD handles double and complex entries.

CNTB (used by DATA statement processsing) bypasses the check for immediate constants.

ROUTINE: COPR - Returns and enters calculations into TBW

PASS: 2

DESCRIPTION: COPR loops operation in TBW and returns previous calculations if present. If not in TBW, the result is calculated and then entered into TBW.

ROUTINE: CPOP - Compare operands for equality

PASS: 1,2

DESCRIPTION: CPOP compares operands for equality and uses the pointers set up by SOPT.

ROUTINE: CPST - COMPLEX statement processor

PASS: 1

DESCRIPTION: Routine CPST processes COMPLEX statements and then sets up for DCLR.

ROUTINE:	CQYL - Copy loop
PASS:	2
DESCRIPTION:	CQYL makes a second copy of the body of a loop in TBG and adds the appropriate entries to TBY. CQYL completes the IF statement in TBCT.

ROUTINE:	CRAR - Check register assignment
PASS:	2
DESCRIPTION:	CRAR analyzes the register assignment for the release of loop-assigned registers.

ROUTINE:	CRMV - Issue a register transfer
PASS:	2
DESCRIPTION:	CRMV builds and issues a secondary/primary register transfer.

ROUTINE:	CRNK - Close up ranks
PASS:	2
DESCRIPTION:	CRNK copies a block of code in TBG format, removing nulls and adjusting parentheses groups to account for removed nulls.

ROUTINE:	CRRG - Clear assigned registers at end of loop
PASS:	2
DESCRIPTION:	CRRG finds registers assigned for the life of the loop and clears registers not needed after the end of the loop.

ROUTINE:	CRVR - Process vector recursion
PASS:	2
DESCRIPTION:	CRVR does the final processing on vector reduction assigned to the real register for the life of the loop.

ROUTINE:	CSNR - Compile statement number
PASS:	2
DESCRIPTION:	This routine compiles a jump to a statement number tag.

ROUTINE:	CSRD - Compile scalar read
PASS:	2
DESCRIPTION:	CSRD compiles a scalar read instruction to read from memory into a pseudo register. CSRD, along with CSWR, maintains TBX, a table of the variables placed in pseudo registers.

DESCRIPTION: (continued)

CSRD searches TBX to see if a value is already in a pseudo register. If it is, CSRD returns the TBX entry rather than generating another read command. CSRD searches TBX backward so that CSRD always obtains the most recent version of the variable. If the value is not already in a pseudo register, CSRD generates the code to do a load.

CSRD checks for stores following reads and checks for redefinition. Additionally, CSRD handles pseudo register generation for double-word operands (complex and double-precision) and does two reads into two adjacently numbered pseudo registers.

CSRD processes subscripted references, immediate and shifted constants as well as normal scalars. A secondary entry for CSRD exists at CS04.

ROUTINE: CSWR - Compile scalar write

PASS: 2

DESCRIPTION: This routine compiles the write instruction. It compiles a move from the A to the S registers, if necessary. CSWR, along with CSRD, maintains TBX, which holds variable tag and new pseudo register information to suppress future reads. A secondary entry for CSWR exists at CS38.

ROUTINE: CTRI - Compile triad

PASS: 2

DESCRIPTION: CTRI is called to compile the code for the triad selected by routine PTRI. A triad consists of $operand_1$, *operator*, $operand_2$. CTRI examines the two operands to see if either is a variable tag rather than a pseudo register. If one is a variable tag, it must be a scalar variable because vector array tags are handled earlier and thus do not reach CTRI as simple operands.

DESCRIPTION: (continued)

CTRI calls CSRD if needed to load operands into registers and does any needed type conversion (calling routine TRAN). It then generates the code.

CTRI may call routine INFN to expand macros for some operations (for example, complex arithmetic). It calls routine LGCL to process logical operators and DBLE to process double-precision operations or function calls.

ROUTINE: CTTY - Check triad type

PASS: 2

DESCRIPTION: CTTY checks the operand types of each triad passed to CTRI and detects nonstandard and illegal logical, relational, and arithmetic expressions.

ROUTINE: CU*xx* - Constant integer operations

PASS: 2

DESCRIPTION: These subroutines of CTRI treat as special cases constant integer operations.

ROUTINE: CVAL - Convert constant tag to value

PASS: 1,2

DESCRIPTION: CVAL converts a constant tag into the corresponding values. CVAL is the inverse of CNTG.

ROUTINE: CX00 - Compile constant expression

PASS: 1

DESCRIPTION: This routine compiles constant expressions in declaratives, PARAMETER, and DATA statements. CX00 is similar to CBLK. Value and pseudo register associations are maintained in TBX and TBY. Several operations (for example, exponentiation) are special cases and the compiled code is not expanded as it normally would be.

CX70 does type conversion on constants, similar to that done by TRAN.

ROUTINE: DAST - DATA or NAMELIST statement processor

PASS: 1

DESCRIPTION: DAST processes DATA and NAMELIST statements. It has two sections. The first section converts a statement to tag-operator format and copies it to TBG. If necessary, CX00 is called to process implied DOs and subscript expressions. Constants are entered into TBE.

The second section of DAST begins at DA40. It processes all of the DATA statements accumulated by the end of Pass 1. DA40 is a driver for routine DP00, which cycles through TBG and builds text tables for the loader.

ROUTINE: DBLE - Double-precision operator processor

PASS: 2

DESCRIPTION: Routine DBLE processes double-precision operators by generating a function call (through CB42A). DBLE is called by routine CTRI.

ROUTINE: DBST - DOUBLE or DOUBLE PRECISION statement processor

PASS: 1

DESCRIPTION: This routine processes DOUBLE and DOUBLE PRECISION statements and then sets up for DCLR.

ROUTINE: DCLR - Declarative processor

PASS: 1

DESCRIPTION: Routine DCLR processes lists for type, COMMON, and DIMENSION declaratives. An array declared via a POINTER statement is also processed in DCLR. It does the following:

- Adds names to TBS and TBT for each simple variable encountered
- Forces types
- Processes DIMENSION descriptors
- Makes an entry in TBA for each array reference
- Makes entries in TBQ for dummy declarators
- Makes an entry in TBI for each array if the Debug Symbol Table is being written

---

ROUTINE: DCST - DECODE statement processor

PASS: 1

DESCRIPTION: DCST processes DECODE statements and then sets up for routine IOST.

ROUTINE: DD*xx* - Implied DO processor

PASS: 1

DESCRIPTION: These routines process implied DOs during the second section of DATA statement processing. They calculate and maintain trip counts and increment the DO variable pseudo register in TBY.

ROUTINE: DETB - Build Debug Symbol Table

PASS: 2

DESCRIPTION: DETB builds the Debug Symbol Table for the loader.

ROUTINE: DLTB - Print DO-loop table

PASS: 2

DESCRIPTION: Routine DLTB prints the DO-loop table on the cross reference map, if requested.

ROUTINE: DMST - DIMENSION statement processor

PASS: 1

DESCRIPTION: This routine processes DIMENSION statements and then sets up for routine DCLR.

ROUTINE: DORP - DO-loop replacement

PASS: 1

DESCRIPTION: DORP checks for the replacement of a 1-line DO-loop with a call to a $SCILIB routine which performs the same operation more efficiently. When DORP is called from STTR, DOST expanded the DO-loop preamble and body in TBG. DORP parses the TGB for an equation match and evaluates the equation to determine if a $SCILIB routine can do the same operation. If a $SCILIB routine is chosen, the DO-loop preamble is rewritten and the $SCILIB call macro is expanded by OP81. Control returns to STTR where the DO terminator label is checked. If the 1-line DO-loop cannot be replaced, control returns to STTR before TGB is altered and DO-loop termination continues normally.

ROUTINE: DOST - DO statement processor

PASS: 1

DESCRIPTION: DOST processes DO statements and does the following:

- Checks syntax
- Calls AT44 to process the DO list
- Builds the termination package in TBR
- Calls OP81 to expand the Pass 1 macro for a DO statement
- Processes implied DOs in I/O statements

DO22 is called from STTR to expand TBR into the DO termination macro.

ROUTINE: DOUN - DO-loop unrolling

PASS: 1

DESCRIPTION: DOUN attempts to unroll an inner DO-loop with known iteration counts. The following conditions must be met before a DO-loop is unrolled.

DESCRIPTION: (continued)

- The number of iterations is less than or equal to the value specified on the UNROLL parameter in the CFT control statement.
- The number of tags associated with the DO-loop body is less than or equal to MAXTAGS (a CFT internal constant).
- No labels, references to labels, external calls, or DO control variable modifications appear inside the DO-loop body.
- A CDIR$ ROLL is not in effect.

Unrolling makes *n* copies of the DO-loop body, where *n* is the trip count, and replaces all occurrances of the DO control variable with constant tags. The DO control variable is set to the value it would have had if the DO-loop remained rolled. The table R parameter word is updated to pop the DO-loop entry.

ROUTINE: DP00 - DATA statement processor

PASS: 1

DESCRIPTION: DP00 processes the lists in TBG during DATA statement processing. DP00 works through the variable and constant lists to build TXT (the loader Text Table) and the DUP table, if possible.

ROUTINE: EAFR - Examine array or function reference

PASS: 2

DESCRIPTION: EAFR examines array or function references for vectorizability. It makes a sweep through TGB, picking out array references (array references are recognized by a nonzero value in the dimension field of the TGB entry).

DESCRIPTION: (continued)

For a function call, EAFR checks TBL to see if a vector version is available. The function must be known to TBL; this eliminates vectorization of any programmer-defined function. If the function is known to TBL, EAFR checks the function's parameters to make sure that they qualify for vectorization.

ROUTINE: EBSN - Process statement number definition within block

PASS: 2

DESCRIPTION: EBSN examines a statement number inside a block to determine if it is a forward branch within the loop. EBSN saves the DO-loop flag and enters the number in the block statement number table.

ROUTINE: EBXR - Examine block for external references

PASS: 2

DESCRIPTION: EBXR examines each statement number defined within a block for references from outside the block.

ROUTINE: EBXS - Adjust block for external entries

PASS: 2

DESCRIPTION: If the first statement of a loop has an external entry, EBXS replaces the statement number with a generated statement number and then compiles the original statement number. This allows invariant code to be removed from the loop.

ROUTINE:	ECNT - Enter conjunctive term
PASS:	2
DESCRIPTION:	ECNT copies the term from TBDT to TBCT if it is not already in TBCT.

ROUTINE:	ECNU - Enter simple term
PASS:	2
DESCRIPTION:	ECNT forms and enters a simple relational term into TBCT.

ROUTINE:	ECST - ENCODE statement processor
PASS:	1
DESCRIPTION:	This routine processes ENCODE statements and then sets up for routine IOST.

ROUTINE:	EDJT - Enter disjunctive term
PASS:	2
DESCRIPTION:	EDJT forms and adds one condition for safe vectorization of a dependency to the conditions already found. Terms are simplified and duplicates are removed before entry.

ROUTINE: EDJU - Enter test for differing CIIs

PASS: 2

DESCRIPTION: EDJT forms and adds one condition for safe vectorization of subscripts with CIIs which do not match.

ROUTINE: EFST - ENDFILE statement processor

PASS: 1

DESCRIPTION: EFST processes ENDFILE statements. Upon recognition of 'ENDFILE', control is passed to RW01 in RWST for further processing.

ROUTINE: EHOL - Enter Hollerith string

PASS: 1

DESCRIPTION: EHOL enters a Hollerith/character string in TBB (or TBE for a DATA statement). The string can be longer than one word. It is packed, eight characters per word, and any trailing H, L, or R is processed. Character or Hollerith constants with an H suffix are padded from the right with blanks to a word boundary. A zero word in TBB terminates a character or Hollerith string that appears in an argument list.

ROUTINE: EIDL - Examine implied DO-loop list

PASS: 1

DESCRIPTION: Routine EIDL examines the implied DO-loop list. It examines subscript expressions in DATA statements to see if subscripting is linear; if it is, the implied DO-loop is compressed into a loader DUP Table.

ROUTINE: ELWD - Enter last word

PASS: 1,2

DESCRIPTION: ELWD is the main table manager. It is used in creating table entries for all sequential tables. ELWD enters a value from register S4 into the last word of a table (KTx in A1).

When making an entry, ELWD detects whether the entry would cause the table to overflow. If this is the case, ELWD calls routine MTAB to expand the table. If there is a memory move, all the table pointer words (in V7) of the moved tables are adjusted.

ROUTINE: EMPR - Error message processor

PASS: 1,2

DESCRIPTION: EMPR is the error message processor. It prints out any error messages, with the message number as its argument.

Normally, for a fatal error during Pass 1, EMPR exits to STTR; for a fatal Pass 2 error, it exits to ABLK. It can optionally return to the caller on fatal errors.

EMPR always returns for nonfatal errors. Before returning, EMPR restores all registers.

ROUTINE: ENST - END statement processor

PASS: 1

DESCRIPTION: ENST terminates Pass 1 and processes FLOWEXIT, $END, or implied RETURN.

DESCRIPTION: (continued)

EN01 begins termination of Pass 1 and sets up for Pass 2. It performs the following functions:

- Allocates actual addresses for blocks
- Processes DATA statements
- Initializes the loader tables
- Links statement number references in TGB
- Counts the number of declared task common blocks and prepares the required number of reserved B registers
- Incorporates TBQ into the entries in TGB
- Moves TGB to TBG; as TGB is moved, the tags are transformed to contain actual offset addresses rather than pointers into TBT.

If fatal errors have occurred, the last three steps are skipped. After ABLK has compiled the last block, the LDR tables are closed and written to the binary file. The symbol table is printed if specified and control then transfers to BG10.

When RNXT detects end of file, control transfers to EN78. CFT issues logfile messages and exits.

ROUTINE: EQST - EQUIVALENCE statement processor

PASS: 1

DESCRIPTION: Routine EQST processes EQUIVALENCE statements. EQUIVALENCE processing is done in three steps. First, as EQUIVALENCE statements are encountered they are packed up, 8 characters per word, and moved from CHB to TBR. Control then returns to RNXT. Further processing is postponed until all other declaratives are processed, when the first executable statement or statement function is found. Finally, storage is assigned.

When the first nondeclarative is encountered, control transfers to EQ10. At this point, TBP contains an entry for each common block entity consisting of the %TBT for that entity. EQ replaces each common block entry in TBP with an entry of the form described in 4.TBP. TBH is updated reflecting each common block length. The corresponding TBT entry for each TBP entry is updated with %TBP for that entity.

DESCRIPTION: (continued)

Next, the packed equivalenced entities are extracted one by one from TBR. A corresponding TBP entry is created for each equivalenced entity. This ends the actions occurring at EQ10.

At the end of Pass 1 (at EN10), a pass is made through TBP to perform two functions for each equivalenced entity: (1) Detect a common block lengthened through an equivalence, in which case, TBH is updated; (2) Assign storage to equivalenced entities that are part of the static or stack blocks. The information in TBP is saved until the end of Pass 1 when all storage allocations are done.

At the end of Pass 1, all variable tags within a block are represented by a base tag and an offset field. This means that during Pass 2, CFT can accurately determine whether there is a conflict between EQUIVALENCE variables.

ROUTINE: ERTX - Invalidate old TBX entries

PASS: 2

DESCRIPTION: ERTX marks TBX entries which are no longer valid due to a transfer of control.

ROUTINE: ESBK - Enter new sub-block

PASS: 2

DESCRIPTION: ESBK enters the statement description into TBBK.

ROUTINE: ESNL - Enter statement number reference

PASS: 2

DESCRIPTION: ESNL checks the the statement number reference for a previous definition in the block. If a previous definition is found, the block is terminated and loop is compiled. If a previous definition is not found, the reference is entered into TBBK.

ROUTINE: ESTB - Enter Symbol Table

PASS: 1

DESCRIPTION: Routine ESTB makes an entry in the Symbol Table (TBS). ESTB is called only after SSTB has determined that the symbol name in question is not yet in TBS. SSTB makes an alphabetical search of TBS and locates the slot that the new symbol name should occupy. ESTB then makes the entry at that point in TBS.

ESTB checks to make sure that there is space in TBS for each new entry. If ESTB finds that more space is required, it calls for a memory move upward or downward, depending on whether the new symbol belongs in the upper or lower half of TBS. (Refer to the section on Table Management in this publication for a detailed description.) Then ESTB completes making the new entry.

ROUTINE: ETBX - Make TBX entry

PASS: 2

DESCRIPTION: ETBX generates the TBX entry and a parallel TBXX entry with a Valid or Invalid flag.

ROUTINE: EVOP - Evaluate operand

PASS: 1

DESCRIPTION: EVOP is used by the DORP routine. EVOP uses a pointer word in S1 as input describing an operand in the TGB. A pointer word contains the dominant type, the length in words, and the starting TGB address for the operand. EVOP returns a value in S1 indicating if the operand is variant, invariant, or illegal.

If the operand's dominant type is single real, not equivalenced, and a variable in a program or common block, the dimensions are used to evaluate the subscripts. Zero dimensions imply the operand is a scalar variable. If the scalar variable is not the DO variable for the current loop, the operand is invariant. If the scalar variable is the DO variable for the current loop, the operand is illegal. If the variable is dimensioned, the subscripts are evaluated by obtaining variant multipliers from the subscript with the DO variable. Only one subscript can contain the DO variable; otherwise, the operand is illegal. If all subscripts are evaluated as legal, the operand is flagged as variant or invariant, depending on the presence of the DO variable in the subscripts.

ROUTINE: EXST - EXTERNAL statement processor

PASS: 1

DESCRIPTION: This routine is the EXTERNAL statement processor. It adds names to TBS and TBT, if they are not already there. It also checks for possible conflicts with existing definitions.

ROUTINE:	FLVS - Find last vector store
PASS:	2
DESCRIPTION:	FLVS finds the last vector store in the loop to check for memory conflicts on the CRAY X-MP computer.

ROUTINE:	FMST - FORMAT statement processor
PASS:	1
DESCRIPTION:	This routine processes FORMAT statements. It enters the FORMAT label in TBS and TBT and acts as a driver for routine FAEV.

ROUTINE:	FNST - FUNCTION statement processor
PASS:	1
DESCRIPTION:	This routine processes FUNCTION statements and then sets up for entry SR04 in subroutine SRST.

ROUTINE:	FPAR - FORMAT statement parser
PASS:	1
DESCRIPTION:	FPAR is the FORMAT statement parser. It processes FORMAT statements, packs them into TBB, and removes blanks and unnecessary commas.

ROUTINE: FRTG - Locate argument tag

PASS: 1

DESCRIPTION: FRTG locates the argument tag if it is a single tag and returns the argument tag to the caller.

ROUTINE: FSHD - Find statement header

PASS: 2

DESCRIPTION: FSHD is given the address of a TBG entry and finds the statement header of the entry.

ROUTINE: FSTK - Force compiler-generated variables onto stack

PASS: 1

DESCRIPTION: FSTK is called in stack mode to force compiler-generated variables onto the run time stack (see Appendix D for the stack frame format). This procedure is done by inserting the stack block tag into each static-tagged TBT entry without a corresponding TBS entry (that is, it moves anonymous variables to the stack).

Two classes of compiler-generated variables are not forced by FSTK. The first class occurs when the space into which argument lists are copied for multiple-entry routines is assigned to the stack by EN10. The second class is function result tags made into special cases by EN84.

ROUTINE: FSUB - Find substring

PASS: 2

DESCRIPTION: FSUB locates the substring for the character operand.

ROUTINE: GBAT - Generate B to A register transfer instruction

PASS: 2

DESCRIPTION: GBAT adds a 024*ijk* (A*i* B*jk*) instruction to PIB. TBX is searched for a matching entry to the B pseudo register in S1. If a matching entry is found, the pseudo register is returned. If a matching entry is not found, a register tag entry is made and entered into TBX. An 024*ijk* (A*i* B*jk*) instruction is generated using a new pseudo register. The resulting A pseudo register is then returned in S1.

ROUTINE: GCBS - Get common block base pseudo register

PASS: 2

DESCRIPTION: GCBC adds instructions to PIB to address the base (0 word) of a common block. GCBS is used by Pass 2 addressing routines to address extended memory common blocks.

S4 contains the tag buffer entry. The offset in S4 is zeroed to create a new table entry. TBX is searched for a matching entry. If a match is found, the matching pseudo register is returned. If a matching entry is not found, a new pseudo register is assigned to S4 and the new entry is added to TBX. Using the new TBX entry, a 020 load instruction is generated and the new pseudo register is returned. If extended memory addressing is used, all 020 loads with common block relocation will be changed to an extended memory 01*h* 1*ijkm* load. The *ijkm* field is 24 bits long to allow for very large addresses.

DESCRIPTION: (continued)

If the offset in S4 is positive and longer than 22 bits, the offset is left in the new TBX entry and the original TBX entry is zeroed. This allows the 01*h* 1*ijkm* load to be generated with the very large offset.

ROUTINE: GCRF - Generate code-and-result tag

PASS: 2

DESCRIPTION: GCRF generates the code-and-result tag for a 22-bit constant deferred in Pass 2.

ROUTINE: GIXA - Generate index address

PASS: 2

DESCRIPTION: GIXA generates pseudo registers for an address on a stack and computes the address.

ROUTINE: GLBD - Get label definition

PASS: 2

DESCRIPTION: GLBD generates an internal label for use in Pass 2 code generation.

ROUTINE: GMEM - Get memory

PASS: 1,2

DESCRIPTION: Routine GMEM gets more memory from the operating system. COS inserts the memory between the tables and the I/O buffers. GMEM moves the tables and adjusts the table pointer words in V7.

ROUTINE: GSBS - Get stack base tag

PASS: 2

DESCRIPTION: GSBS finds the pseudo register holding the address of the stack base.

ROUTINE: GTCB - Get task common block base pseudo register

PASS: 2

DESCRIPTION: GTCB adds instructions to PIB to address the base (word 0) of a task common block. GTCB is used by Pass 2 addressing routines to address task common blocks.

The corresponding common block attribute entry (COMTAG) is obtained based on the S4 tag. If the B pseudo register field is nonzero, a call is made to GBAT to generate a B to A register transfer. If the B register field is 0, the base address of the common block must be calculated. A new TBX entry is created by zeroing the offset. TBX is searched for a matching entry. If a match is found, the matching pseudo register is returned. If a matching entry is not found, a new pseudo register is assigned and the new TBX entry is added to TBX. A 100-load instruction is generated and the base pseudo register is returned.

If the S4 offset is positive and longer than 22 bits, a check is made for extended memory addressing. If extended memory addressing is not requested, an error message is issued. If EMA is being used, the offset for the original TBX entry is zeroed and a 020 load instruction is generated (it will become a 01*h ijkm* extended memory load). An A register add is then generated to add the original task common block base PR to the very large offset PR. The pseudo register used as the result of the add is then returned.

ROUTINE:	GTST - GO TO statement processor
PASS:	1,2
DESCRIPTION:	This routine processes GO TO statements. The code beginning at GT30 does Pass 2 processing.

ROUTINE:	HOLD - Hollerith data assembler
PASS:	1
DESCRIPTION:	HOLD assembles Hollerith/character data (using routine EHOL) and returns a tag for AT02. HOLD assigns a type of Boolean to any Hollerith constant not used in an argument list.

ROUTINE:	IATY - Insert address in TBY
PASS:	2
DESCRIPTION:	Routine IATY inserts TBG address in TBY reference entries.

ROUTINE:	IELS - ELSE and ELSE IF statement processor
PASS:	1
DESCRIPTION:	IELS processes ELSE and ELSE IF statements. An entry is made in TBR for each ELSE/ELSE IF statement.

ROUTINE: IEND - ENDIF statement processor

PASS: 1

DESCRIPTION: This routine processes ENDIF statements. It uses the TBR entries for the current IF level. The TBR entries at the current IF level are then deleted.

ROUTINE: IFST - IF statement processor

PASS: 1,2

DESCRIPTION: IFST is the IF statement processor. It assembles test expressions (via routine AT*xx*) and builds the statement number list. It creates a TBR entry for a block IF statement.

IF40 generates the jump instructions during Pass 2.

ROUTINE: IGXF - Intergroup transfer

PASS: 2

DESCRIPTION: IGXF resets segment pointers and definitions when starting new segments of code in a block of code.

ROUTINE: IMST - IMPLICIT statement processor

PASS: 1

DESCRIPTION: This routine processes IMPLICIT statements. It changes the implicit type codes in register V7 and changes the types of a function entry and dummy arguments, if necessary.

ROUTINE:	INFN - Intrinsic function expander
PASS:	2
DESCRIPTION:	INFN expands intrinsic function macros during Pass 2. It compiles the functions and substitutes argument pseudo registers. The macros for this are in TBM. INFN requires that the parameters to the intrinsic function are stored in V5, beginning with element 0. Up to 64 parameters can be included, depending on how many the intrinsic function calls for.

ROUTINE:	IOST - I/O statement processor
PASS:	1
DESCRIPTION:	IOST is the main processor for I/O statements. It converts I/O statements into a series of calls to library routines.

ROUTINE:	IPRN - Insert parentheses
PASS:	1
DESCRIPTION:	IPRN encloses an expression in parentheses if the expression contains a masking operator (.EQV., .XOR., .NEQV., .OR., and .AND.).

ROUTINE: IRST - Process INTRINSIC statement

PASS: 1

DESCRIPTION: IRST processes the INTRINSIC statement and ensures that the names declared INTRINSIC are intrinsic function names and were correctly used as intrinsic functions. IRST creates an entry in TBS for the intrinsic function name. An entry consisting of an external function tag is made in TBT and is updated at the first function reference.

ROUTINE: ISRF - Insert subscripted reference

PASS: 2

DESCRIPTION: ISRF sets up code generation for a subscripted reference.

ROUTINE: ITYP - Implicit type determiner

PASS: 1

DESCRIPTION: ITYP determines the implicit type of a variable name. It uses the first character of the variable name as an index into register V7.

ROUTINE: IVTX - Invalidate TBX entries

PASS: 2

DESCRIPTION: IVTX marks TBX entries generated after a given point as invalid so they will not be used incorrectly.

ROUTINE: LBLK - Locate sub-block definition

PASS: 2

DESCRIPTION: LBLK locates the TBBK entry and the TBG address.

ROUTINE: LDIV - Integer divide processor

PASS: 2

DESCRIPTION: LDIV generates a function call to process the 64-bit integer divide. LDIV is called by CTRI.

ROUTINE: LGCL - Logical and relational operator processor

PASS: 2

DESCRIPTION: Routine LGCL generates code to process relational and logical operators. It is called by CTRI and it may call CTRI or INFN to generate code.

ROUTINE: LGST - LOGICAL statement processor

PASS: 1

DESCRIPTION: This routine processes LOGICAL statements and then sets up a call to routine DCLR.

ROUTINE: LSOM - Load store overlap move

PASS: 2

DESCRIPTION: LSOM inserts DBM,EBM around a vector instruction which could cause memory overlap on the CRAY X-MP Computer System.

ROUTINE: LSOV - Load/store overlap check

PASS: 2

DESCRIPTION: LSOV checks for vector load/vector store overlaps on the CRAY X-MP Computer System.

ROUTINE: LT*xx* - Loader Table generator

PASS: 2

DESCRIPTION: The LT*xx* routines build the loader tables.

LTST initializes the loader tables called by END at the start of Pass 2. It takes all the instructions from RASN and packs them into the format required by the loader. It adds indicator bits used by the loader to indicate what to relocate, whether an instruction references an external, common block information, and so on.

LTGN builds the loader tables for each block. It is called after routine RASN has assigned registers. LTGN exits to ABLK to fetch the next block. It packs each instruction into TBB. Since the actual target address for jumps may not be known, all jump instructions are linked in TBB. LTGN also builds the External Reference Table (TBE) and the Block Relocation Table (TBR) for variable references and jumps. If the generated code listing is requested, LTGN calls the appropriate OUT*xx* routine to format the output.

LTFU can be called to force pass instructions until a word boundary is reached. It is normally used at the end of a routine or before an entry.

LTND is called at the end of Pass 2 to terminate the loader tables. The actual number of B and T registers needed to be saved/restored is inserted in all of the EXIT/ENTRY sequences. Actual statement label addresses are inserted in jump instructions.

ROUTINE: MAP - Map block names and lengths

PASS: 2

DESCRIPTION: MAP prints the block names and lengths list in the symbol table, if requested.

ROUTINE: MCEX - Special case handling for scheduler

PASS: 2

DESCRIPTION: MCEX expands the instruction sequence after the sequence is scheduled.

ROUTINE: MMEM - Manage memory

PASS: 1,2

DESCRIPTION: MMEM is a general memory management routine that does memory moves through the vector registers. It can move a block of memory either up or down. The source and destination blocks specified may overlap.

ELWD determines that a move is necessary, MTAB determines which way and where to move, and MMEM does the actual move.

ROUTINE: MSAR - Move S to A register

PASS: 2

DESCRIPTION: MSAR generates the TBW entry to move the S to the A register. MSAR returns the PR found in TBW if one exists; otherwise, the next entry in TBW is returned and generates code moving the S to the A register.

ROUTINE: MTAB - Move table

PASS: 1,2

DESCRIPTION: MTAB moves tables in memory. It is called by ELWD or ESTB. Each time an addition must be made to a table (a table has overflowed), MTAB looks at adjacent table pointer words in V7 to see what kind of table move is necessary.

First, MTAB looks above the current table for any gaps; if there are, tables are moved upward to fill the gaps (rather than moving down and taking more memory). If no gaps exist above this table, MTAB moves the table and all tables below it down 100_8 words of memory and allocates 100_8 words to the table that overflowed. If the downward move gets close to the top of TGB or PIB, GMEM is called to get more memory. MTAB calls MMEM to do the actual move. Table pointers in V7 are updated by MTAB.

ROUTINE: MVOP - Move operands

PASS: 1,2

DESCRIPTION: MVOP moves operands and uses the pointers set up by SOPT.

ROUTINE: NARG - Return number of arguments

PASS: 2

DESCRIPTION: NARG is a function returning the number of arguments in a tag-and-operator-format subroutine or function call.

ROUTINE: NICV - Numeric input conversion

PASS: 1,2

DESCRIPTION: NICV converts numeric input by calling the external routine $NICV. Input parameters are moved from the registers used by the old entry sequence to $NICV to the registers used by the new entry sequence to $NICV. Output parameters are moved from the registers used by the new exit sequence to the registers used by the old exit sequence. Registers A7 and S7 are preserved across the call as in the old sequence calls to $NICV.

ROUTINE: NLST - NAMELIST statement processor

PASS: 1,2

DESCRIPTION: Routine NLST is the NAMELIST statement processor. Pass 1 sets up for DA08 (of DAST) to convert the list to tags and place them in TBG. In Pass 2, the code beginning at NL20 processes each NAMELIST group. It builds the table of names and attributes and calls LTGN to pack the list into the loader tables.

ROUTINE: NMTB - Write statement number table

PASS: 2

DESCRIPTION: NMTB writes the statement number table in the symbol table, if requested.

ROUTINE: NOBTVAR - Enter TBT index into table KTNOBT

PASS: 1

DESCRIPTION: If OPT=BTREG, NOBTVAR enters the TBT index into table KTNOBT of any variable used in a DATA, NAMELIST, or SAVE statement or in an IOSTAT=, or EXIST=, etc. in an I/O statement. These variables will not be assigned to B or T registers.

ROUTINE: NOCV - Numeric output conversion

PASS: 1,2

DESCRIPTION: NOCV converts numeric output by calling the external routine $NOCV. Input parameters are moved from the registers used by the old entry sequence to $NOCV to registers used by the new entry sequence to $NOCV. Output parameters are moved from the registers used by the new exit sequence to the registers used by the old exit sequence.

ROUTINE: NTRY - ENTRY statement processor

PASS: 1,2

DESCRIPTION: NT20 is called once for each routine entry point during Pass 2. For each entry point NT20 must:

- Place the entry name before entry sequence code,
- Place the entry parcel value in PDT (TBH),
- Compile the entry sequence code,
- Compute address tags for the entry's arguments and save in TBP for later reference, and
- If there are multiple entry points, compile code to move passed-in arguments to standard locations and reset B01 (the argument list base pointer) to point to the start of the standard location space.

DESCRIPTION:
(continued)

Alternate returns and character functions can cause processing complications. Each alternate return is assigned a dummy argument tag and contributes to its entry point's argument count. Alternate returns are unique because their dummy argument tag does not have a TBT entry and the caller does not pass anything although space in the argument list is reserved by the caller.

When character functions are used, the compiler creates an address-of-result argument for the caller and prefixes it to the user's argument list. A dummy argument tag is not assigned to the created argument and the argument is not included in the TBG number-of-arguments field.

The following example shows entries to a routine and the associated dummy argument tags.

Entries	Associated dummy argument tags
E1(A,*,B)	110, 111, 112
E2()	none
E3(*,*)	113, 114
E4(B,C)	115, 116
E5(C,B,A,X)	117, 120, 121, 122

The alternate return tags are 111, 113, and 114 and they have no TBT entry.

The primary dummy argument tags (the smallest or only dummy argument tag associated with an argument) are 110(A), 112(B), 116(C), and 122(X).

The secondary dummy argument tags (nonprimary nonalternate return dummy argument tags) are 115 and 120 associated with 112(B), 117 associated with 116(C), and 121 associated with 110(A).

If the code following E4 references B and the calling code entered the routine at E4, B's descriptor is the first in the argument list. If the routine entered at E1, B's descriptor is the third in the argument list. This ambiguity cannot be resolved at compile time. CFT solves the problem by placing code at each entry point of a multiple-entry routine moving the entry's passed-in argument descriptors to standard locations. The standard location space is contiguous and contains one slot for the argument list header, one slot for

DESCRIPTION: (continued)

each dummy argument (for example, four slots for A, B, C, and X), and if the routine is a character function, one slot for the invented address-of-result argument.

In the non-stack calling sequence, the standard location space for multiple-entry routines is the beginning of #CL.

For a single-entry routine, argument-moving code is not needed because the standard location space is the passed-in argument list. If there are multiple entries, NT20 also generates code resetting B01 to the base of the standard location space.

NT20 communicates these offsets to the rest of CFT by table P (TBP is built by NT20). TBP contains one address tag for each dummy argument tag in increasing order: TBP's first entry is an address tag giving standard location of the 110-tagged dummy argument, the second TBP entry is for the 111-tagged dummy argument, and so on.

Continuing from the previous example, this example shows the dummy argument tags and how TBP appears with the standard location base of B01 (the 601 tag).

Dummy argument tag	TBP (standard location tags)
110 (primary A)	601000000010000024027
111 (alternate return)	0
112 (primary B)	601000000020000024027
113 (alternate return)	0
114 (alternate return)	0
115 (second B)	copy of 112's entry
116 (primary C)	601000000030000024027
117 (second C)	copy of 116's entry
120 (third B)	copy of 112's entry
121 (second A)	copy of 110's entry
122 (primary X)	601000000040000024027

For a multiple-entry non-stack routine, the standard location is in the first frame of #CL (#CL+0 and so on) and CFT generates offset-into-#CL code for argument references (the TBP tag fields are 604).

DESCRIPTION: (continued)

The first offset field is 1 because the 0-offset address is reserved for a copy of the passed-in argument list header. For a character function, the first offset field is 2 because the 1-offset address is reserved for the address-of-result argument (this argument does not have a 110-577 tag, and must be special-cased).

ROUTINE: OLEV - Operator level

PASS: 2

DESCRIPTION:

OLEV finds the current operator level. It is called by CBLK to begin a forward scan through an expression, looking for operators at certain precedence levels. CBLK makes successive calls to OLEV once for each operator level, beginning from the top of the hierarchy and working downward. (All operations of the same precedence are done at the same time, taking advantage of the CRAY's independent functional units.) OLEV works from the innermost parenthesis pair outwards. The hierarchy from highest to lowest is:

Exponentiation
Multiplies and divides
Adds and subtracts
Logical
Relational

OLEV works together with CBLK and PTRI to generate code for each operation and then to replace each triad processed in the text with the result register used.

Whenever OLEV finds an operator, it returns a pointer to that operator to CBLK, and CBLK calls PTRI to process the triad. Once PTRI and CTRI have reduced the triad to a result and returned the result register to CBLK, CBLK again calls OLEV to find another operator.

A simple test in OLEV determines whether any more operators of the same precedence level remain in the expression. Once no operators remain, the scan is terminated.

DESCRIPTION: (continued)

Because Pass 1 puts a set of parentheses around the entire righthand side of an expression, the entire expression can be worked off to one result register. It is also necessary to look at the left-hand side of the expression because a subscript expression may have to be worked off as well.

ROUTINE: OP*xx* - Operator processor

PASS: 1

DESCRIPTION: OP*xx* processes operators during Pass 1 conversion to tag-operator format. It works in conjunction with AT*xx*, converting operators into precedence numbers for TBG. It also processes parentheses in expressions, subscripts, and calls and maintains a parenthesis stack in V6. OP*xx* processes commas in lists, and in subscripts it expands the subscript reference.

OP81 expands Pass 1 macros such as the DO macros and also expands arithmetic statement functions.

ROUTINE: OPEN - OPEN, CLOSE, and INQUIRE statement processor

PASS: 1

DESCRIPTION: This routine processes OPEN, CLOSE, and INQUIRE statements and then sets up a call to routine IOST.

ROUTINE: OTBL - Output table generator

PASS: 2

DESCRIPTION: OTBL performs output table generation. It writes the IDENT, etc. on the pseudo CAL file, if it is called for.

ROUTINE: OUT*xx* - Pseudo-CAL output generator

PASS: 2

DESCRIPTION: OUT*xx* generates the pseudo CAL output. It converts instructions to ASCII and writes them on the output or pseudo CAL file, if requested.

ROUTINE: OUTBB - Output BLOCK BEGINS

PASS: 2

DESCRIPTION: This routine writes the "BLOCK BEGINS" message.

ROUTINE: PAST - PAUSE statement processor

PASS: 1

DESCRIPTION: Routine PAST processes PAUSE statements.

PA02 is the code common for processing STOP and PAUSE.

PA10 is common for STOP and PAUSE.

ROUTINE: PBLK - Select and prepare compilation of next statement

PASS: 2

DESCRIPTION: PBLK invalidates the TBX entries, if necessary, and selects the next statement to be compiled. At the end of the block, the block is transferred to SKED. Prepare the compressed index sub-blocks and compile any statement number definitions.

ROUTINE: PCIV - Process conditional CII variables

PASS: 2

DESCRIPTION: PCIV examines the conditional block for CII references and moves the load of CII from the conditional block.

ROUTINE: PCON - Promote constants

PASS: 2

DESCRIPTION: PCON promotes the constants that have been retained thus far intact in subscript expressions. It scans through the subscript references in TBG and collects all possible constants into a single offset term.

ROUTINE: PCST - Process conditional store

PASS: 2

DESCRIPTION: PCST invalidates the TBX entries made unusable by a conditional store.

ROUTINE: PEXP - Process exponent

PASS: 2

DESCRIPTION: PEXP is the exponentiation processor. It is called from CBLK if OLEV finds an exponentiation operation. It searches for the last ** in a sequence (for example, in the expression A**B**C, the B**C would be processed first).

This routine handles special casing of **2, **3, and **4 by calling routine CTRI to do the multiplies. Otherwise, PEXP builds the name of the external routine that does the operation and calls CTRI; CTRI then calls PE90 to generate the actual CALL (through CB32 and CB42A).

ROUTINE: PGST - PROGRAM statement processor

PASS: 1

DESCRIPTION: Routine PGST processes the PROGRAM statement. It then sets up and transfers to SR04 (of SRST).

ROUTINE: PHDL - Print header line

PASS: 1,2

DESCRIPTION: PHDL prints the header line on the top of each page of the program listing.

ROUTINE: PIST - PRINT statement processor

PASS: 1

DESCRIPTION: This routine processes the PRINT statement, sets unit to 101, and goes to routine IOST.

ROUTINE: PLDP - Process loop dependencies

PASS: 2

DESCRIPTION: PLDP marks the dependencies in TBBK.

ROUTINE: PMRT - Process memory reference time

PASS: 2

DESCRIPTION: PMRT determines when memory allows the issue of a given instruction.

ROUTINE: PMST - Process memory set time

PASS: 2

DESCRIPTION: PMST sets the time that memory is free after an instruction issue.

ROUTINE: PNST - POINTER statement processor

PASS: 1

DESCRIPTION: PNST processes the POINTER statement. It adds the pointer and pointee names to TBS and TBT, checks syntax, and then calls DC17 (of DCLR) to process pointee dimensions, if necessary.

ROUTINE: PPDP - Process plus dependency

PASS: 2

DESCRIPTION: PPDP checks for a plus dependency. If a plus dependency is found, PPDP inserts a load of the referenced operands.

ROUTINE: PPGN - Print page number list

PASS: End of compilation

DESCRIPTION: Routine PPGN prints out a list of routine names and starting page numbers at the end of the source listing.

ROUTINE: PRST - PARAMETER statement processor

PASS: 1

DESCRIPTION: This routine processes the PARAMETER statement. It enters the parameter name in TBS, evaluates the value (via CX*xx* and XC*xxx*), creates a constant tag and enters it in TBT. PRST will also do type conversion (via CX80) if necessary.

ROUTINE: PSRC - Process secondary register clear

PASS: 2

DESCRIPTION: PSRC clears out the secondary registers that have contents no longer needed.

ROUTINE: PTRI - Process triad

PASS: 2

DESCRIPTION: This routine processes triads. It is the main driver for routine CTRI. PTRI is called after OLEV has found an operator. It finds the corresponding operands and calls CTRI.

If there is a sequence of operators at the same precedence level, PTRI attempts to select operands that are both variant or both invariant. For example, given 1+J+2, PTRI selects 1+2 as the first triad to process. For triad A*3*B, PTRI processes A*B first.

The code at PT40 treats constant increment integer variables as special cases. The code is entered from ABLK.

ROUTINE: PUST - PUNCH statement processor

PASS: 1

DESCRIPTION: Routine PUST processes PUNCH statements. It then sets unit to 102 and goes to IOST (via PI01).

ROUTINE: RASN - Register assignment

PASS: 2

DESCRIPTION: RASN assigns pseudo registers to real registers and issues instructions to the final instruction buffer.

ROUTINE: RBIN - RASN initialization

PASS: 2

DESCRIPTION: RBIN sets up initial values for temporaries used in register assignment.

ROUTINE: RBLI - Reissue bottom-load instructions

PASS: 2

DESCRIPTION: RBLI analyzes bottom-load instructions to determine when to reissue them and sets up the reissue.

ROUTINE: RBMV - Remove the inserted save from the IF-block

PASS: 2

DESCRIPTION: RBMV is used by the instruction scheduler to move the save of the register from the conditional block of code.

ROUTINE: RBRG - Initialize register times

PASS: 2

DESCRIPTION: RBRG assigns initial times to registers.

ROUTINE: RCCK - Register chain check

PASS: 2

DESCRIPTION: RBRG determines if the result of one operation is needed for the evaluation of a specified pseudo register.

ROUTINE: RDPT - Remove duplicate terms

PASS: 2

DESCRIPTION: RDPT compares two terms and removes duplicate additive expressions from both terms. Duplicate expressions are replaced by nulls.

ROUTINE: RDST - READ statement processor

PASS: 1

DESCRIPTION: RDST processes READ statements. It sets unit to 100 if the READ is in short form and goes to IOST.

ROUTINE: REST - REAL statement processor

PASS: 1

DESCRIPTION: REST processes REAL statements and then sets up for routine DCLR.

ROUTINE: RNXT - Read next statement

PASS: 1

DESCRIPTION: Routine RNXT reads the next statement from the input file, checks for comment and continuation cards, and (if necessary) drives the source output listers.

RNXT places the statement in the Character Buffer (CHB). It skips comment lines and concatenates continuation lines. Columns 1 through 6 and 72 through the end of line are discarded. All blanks are removed from CHB except on FORMAT statements. Hollerith/character text characters are flagged by setting their sign bits. RNXT converts lowercase characters to uppercase.

FORMAT and END statements are special cases. RNXT also checks to ensure all parentheses are matched pairs. See section 2 for a detailed description of RNXT.

End of statement is indicated by a zero word.

ROUTINE: ROSR - Convert an invariant **CII

PASS: 1

DESCRIPTION: ROSR reduces the strength of an invariant **CII by converting it to a multiply operation.

ROUTINE: RPST - Replacement statement processor

PASS: 1

DESCRIPTION: This routine processes the replacement statement. It sets up and transfers to AT36 to convert to tag-operator format.

DESCRIPTION: (continued) If array bounds checking is in effect and the left-hand side of the replacement statement is an array, RPST sets up for OP02 to do the actual bounds checking.

ROUTINE: RSTB - Restore table pointers

PASS: 1,2

DESCRIPTION: RSTB restores a dynamic table pointer to a previous state given the current and previous pointers.

ROUTINE: RTC - Real-time clock

PASS: End of compilation

DESCRIPTION: This routine calculates compile time and formats the logfile message.

ROUTINE: RTST - RETURN statement processor

PASS: 1

DESCRIPTION: RTST processes the RETURN statement.

ROUTINE: RWST - REWIND statement processor

PASS: 1

DESCRIPTION: This routine processes the REWIND statement. RW01 is common for REWIND, ENDFILE, and BACKSPACE. Subroutines at IO10 and IO20 are called and control is passed to IO67 for final processing.

ROUTINES: SAST, SA50 - SAVE statement processors

PASS: 1

DESCRIPTION: SAST processes the SAVE statement. If the SAVE list is empty, SAVE ALL is implied. SAST sets a SAVE ALL flag, makes the default block tag a static tag, and replaces all stack tags in TBT with static tags. If the SAVE list is not empty, SAST creates an entry in TBSV for each list entry (see 4.TBSV-1).

Semantic checking of the SAVE list cannot be done until all specification statements are processed. For example, variables in common blocks cannot appear in a SAVE list, but variables in common blocks are not known until all specifications are processed and equivalences are resolved. Routine SA50 performs semantic checks after the equivalence resolution.

ROUTINES: SBLS, SBLT, SBMS, SBRS, SBRT - Search table

PASS: 1,2

DESCRIPTION: These routines are all used for table searches. See section 4 for detailed descriptions.

ROUTINE: SBOP - Scan buffer for operator

PASS: 1

DESCRIPTION: This routine scans a buffer for an operator at the current parenthesis level, that is, one that is not contained within parentheses. It searches forward through a string buffer until end of statement or the specified operator is found. The operator to be searched is given in a 2-word mask at the current parenthesis level.

If SBOP encounters a parenthesis, it scans for the matching parenthesis before resuming its scan for the target character.

ROUTINE: SBUF - Scan buffer for match or end of statement

PASS: 1

DESCRIPTION: Routine SBUF scans a string buffer to find a matching character or end of statement. SBUF scans forward through the string until it finds a character that matches the one specified by a 2-word mask. SBUF terminates if it comes to the zero word at the end of the character string.

ROUTINE: SDCO - Suppress dead code

PASS: 2

DESCRIPTION: SDCO is called after an unconditional branch eliminating following statements which cannot be executed.

ROUTINE: SDPF - Set Dependency flags

PASS: 2

DESCRIPTION: SDPF sets the Block Dependency flags in KTBK.

ROUTINE: SDPN - Search double-precision function name table

PASS: 1

DESCRIPTION: SDPN searches the double-precision function name table. This routine is used if OFF=P is selected on the CFT card. OFF=P causes the conversion of double-precision function names to single-precision names.

ROUTINES: SFLS, SFLT, SFMS, SFRS, SFRT - Search table

PASS: 1,2

DESCRIPTION: These routines are all used for table searches. See section 4 for detailed descriptions.

ROUTINE: SFMN - Search forward for nonzero field

PASS: 1,2

DESCRIPTION: SFMN searches the string for an entry with a nonzero field.

ROUTINE: SFST - Arithmetic statement function definition processor

PASS: 1

DESCRIPTION: Routine SFST processes arithmetic statement functions definitions. It collects dummy argument names in register V5, sets up and branches to AT06 to convert the right-hand side to tag-operator format. The tag-operator definition is copied to TBF by routine STTR and then expanded into TBG whenever an actual reference to that arithmetic statement function is encountered.

ROUTINE: SGES - Search group for equality in string

PASS: 2

DESCRIPTION: SGES searches for duplicates of a given group in a specified string.

ROUTINE: SIDL - Search implied DO-loop name list

PASS: 1

DESCRIPTION: Routine SIDL searches the implied DO-loop name list in register V4. SIDL is used by routine DAST to search the list of known active DO-loop indexes.

ROUTINE: SIN - Search for Intrinsic Function Name Table (TBJ)

PASS: 1

DESCRIPTION: Routine SIN searches TBJ to determine if the input parameter is an intrinsic function name. If the input parameter is an intrinsic function name, the attributes of that function (from TBK) are returned.

ROUTINE: SKED - Code scheduling

PASS: 2

DESCRIPTION: SKED is the instruction scheduler. It breaks the code block into groups and determines instruction dependencies. SKED is entered from ABLK, CBLK, or TRUNC.

ROUTINE: SNGN - Statement number generator

PASS: 1

DESCRIPTION: This routine generates a 5-digit internal statement number. Statement numbers are assigned in ascending order, beginning with 00001. SNGN is used when a statement number is needed for DO-loops or block IFs, for example.

ROUTINE: SOPT - Scan for end of operand

PASS: 1,2

DESCRIPTION: SOPT scans for the end of the operand.

ROUTINE: SPFH - Special-case intrinsic function header based on its arguments

PASS: 2

DESCRIPTION: SPFH is called from CBLK at CB35 if an intrinsic function header has the SAF (special processing) bit set. Shift and condition vector merge intrinsics are special cases if their arguments are vector or scalar.

ROUTINE: SPFR - Check function arguments for special cases

PASS: 2

DESCRIPTION: SPFR is called from EAFR at EA40 if an intrinsic function header has the SAF (special processing) bit set. Shift intrinsics are treated as special cases to ensure a scalar second argument. If the second argument is not scalar, the current loop cannot vectorize.

ROUTINE: SPRN - Suppress redundant parentheses groups

PASS: 2

DESCRIPTION: SPRN scans a string in TGB format, searching for redundant parentheses and replacing them with nulls. Redundant groups are either empty or contain only one operand.

ROUTINE: SRST - SUBROUTINE statement processor

PASS: 1

DESCRIPTION: Routine SRST processes SUBROUTINE statements.

Beginning at SR04 is code common for PROGRAM, SUBROUTINE, FUNCTION, and BLOCK DATA statements. It enters the dummy argument names in TBS and TBT. SRST also generates a call to FLOWENTR if ON=F is specified on the CFT statement.

ROUTINE: SSTB - Search sorted table

PASS: 1,2

DESCRIPTION: SSTB is the sorted table search routine. See section 4 for a detailed description.

ROUTINE: STST - STOP statement processor

PASS: 1

DESCRIPTION: STST is the STOP statement processor. If ON=F is specified on the CFT statement, STST generates a call to FLOWSTOP. It then sets up and branches to PA02.

ROUTINE: STTP - Statement type determination

PASS: 1

DESCRIPTION: Routine STTP determines statement type. It is called after RNXT has assembled a statement in the Character Buffer. STTP treats as special cases assignment statements, statement function definitions, and DO and FORMAT statements.

DESCRIPTION: (continued)

If the statement being considered is not one of the above, STTP assembles the first four alphabetic characters from CHB and searches the statement type table. The low-order 24 bits of a table entry contain the address of the corresponding statement processor. If a match is found, STTP exits to the appropriate statement processor.

STTP determines statement group and checks whether statements in the program are in legal order. See section 2 for a details on legal statement order.

If the statement is determined to be the first nondeclarative statement, STTP branches to EQ10 to process the EQUIVALENCE statements.

ROUTINE: STTR - Statement processing terminator

PASS: 1

DESCRIPTION: STTR terminates statement processing. If the statement is an arithmetic statement function, STTR copies the definition to TBF. A check is made for possible replacement by conditional vector merge intrinsic functions. STTR links statement headers by setting the forward and backward index fields. STTR detects DO-loop termination. If there is a 1-line DO-loop, a call is made to DORP for conditional replacement by a SCILIB routine. If the replacement is successful and the statement number does not terminate additional DO-loops, STTR branches to RNXT to begin the next statement. If the replacement is unsuccessful or the DO-loop is not a 1-line DO-loop, STTR branches to DO22 to process DO-loop termination. STTR returns if it is called during an internal Pass 1 macro expansion.

ROUTINE: SVEC - Set Vector Array flag

PASS: 2

DESCRIPTION: SVEC sets the Vector Array flag in all probable scalar temporary vectors. It is called by PCON.

ROUTINE: SYMADD - Add symbol to cross-reference

PASS: 1

DESCRIPTION: SYMADD updates cross-reference tables TBU and TBV.

ROUTINE: SYTB - Print Symbol Table

PASS: 2

DESCRIPTION: SYTB prints out the Symbol Table in either the short form or the full cross-reference form, if requested.

ROUTINE: TFBK - Transfer between sub-blocks in a conditional loop

PASS: 2

DESCRIPTION: TFBK controls compilations of various sub-blocks in a conditional vector loop. TFBK inserts a transfer around a block and the definitions for the generated statement numbers of the loop. TFBK replaces the statement number definitions in a second block with the generated statement numbers.

ROUTINE: TPRU - Tally PR usage

PASS: 2

DESCRIPTION: TPRU makes a list of PRs to be used after the end of the instruction group.

ROUTINE:	TRAN - Do type conversion
PASS:	2
DESCRIPTION:	This routine compiles code for type conversion. If the two operands are not the same type, TRAN generates code converting the operand of lowest type to match the code of the higher type operand.

ROUTINE:	TRUN - Truncate after each floating-point operation
PASS:	2
DESCRIPTION:	TRUN compiles code to do truncation after each floating-point operation, if TRUNC=*nn* is specified.

ROUTINE:	VCTL - Vector loop control
PASS:	2
DESCRIPTION:	This routine provides vector loop control.

ROUTINE:	VE*xx* - VECTOR/NOVECTOR directive processor
PASS:	1
DESCRIPTION:	The code at VE00 processes the VECTOR compiler directive, while that at VE01 processes the NOVECTOR directive.

ROUTINE: VLAN - Vector loop analysis

PASS: 2

DESCRIPTION: VLAN analyzes the loop for vector hazards such as vector temporaries defined in conditional code, transfers out of the loop, and dependencies.

ROUTINE: WRST - WRITE statement processor

PASS: 1

DESCRIPTION: This routine processes WRITE statements. It sets unit to 101 for the short form of WRITE and branches to IOST.

ROUTINE: XC00 - Execute code

PASS: 1

DESCRIPTION: This routine interpretively executes the code compiled by CX00 for constant expressions.

ROUTINE: XX00 - Set of interpreters for instructions compiled by CX00

PASS: 1

DESCRIPTION: This routine consists of a set of interpreters for the instructions compiled by CX00. It is driven by XC00.

ROUTINE:	ZMEM - Clear a block of memory
PASS:	1,2
DESCRIPTION:	ZMEM clears a block of memory. ZMEM is called to clear the tag buffer between compilation units and clear additional tag buffer space when the tag buffer grows. ZMEM is also called to clear scratch tag buffer space after restructuring IF statements.

CFT I/O

6

CFT converts a user program written in FORTRAN to the binary machine language of the CRAY-1. The compiler is loaded and begins processing as a result of a control statement call from a user job deck.

CFT requires two types of input: the user program to be compiled and a user job deck that gives instructions for controlling compilation. The output provided by CFT includes the user's compiled FORTRAN program in relocatable binary and a printable record of the compilation.

Figure 6-1 illustrates the I/O datasets used by CFT during compilation.

6.1 INPUT TO CFT

A user's FORTRAN program submitted to CFT must comply with certain program specifications. The syntactic and notational guidelines outlined in the CRAY-1 FORTRAN (CFT) Reference Manual must be observed.

A user may select compiler options through the CFT control statement and through use of compiler directives. (Refer to the CRAY-1 FORTRAN (CFT) Reference Manual for descriptions of both the CFT statement and CFT directives.)

6.2 OUTPUT FROM CFT

The output obtained from CFT is dependent on the options selected by the user. The principal output obtained is the compiled FORTRAN program in relocatable form. Optional output consists of a printable record of the compilation, including the FORTRAN source code, the assembly language equivalent generated by CFT, the cross-reference lists, the Debug Symbol Table, and more.

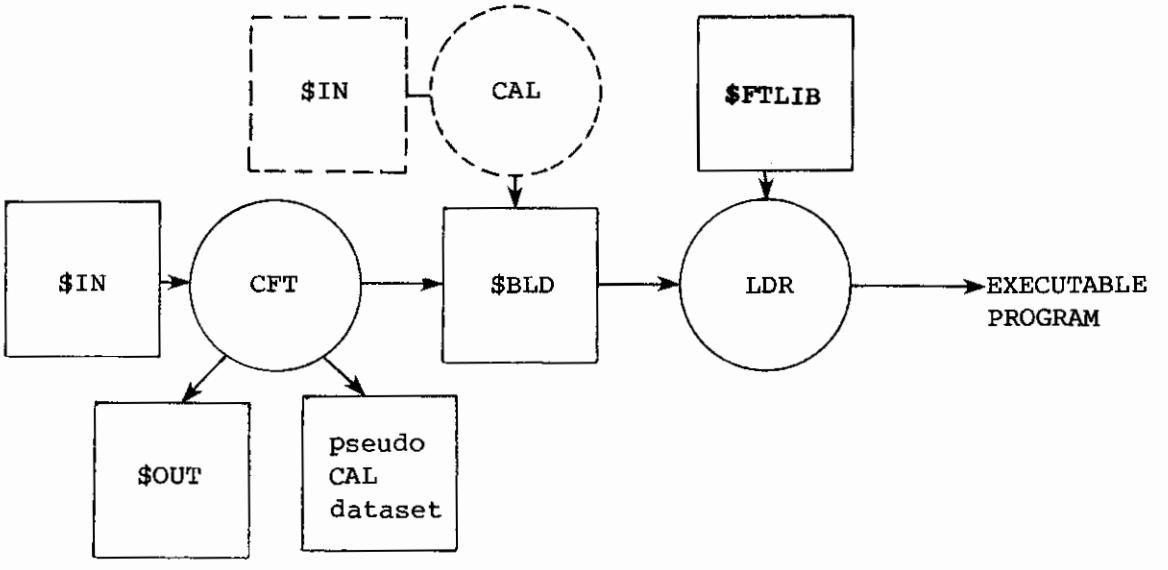


Figure 6-1. I/O datasets used during compilation

6.3 I/O DATASETS

As many as four I/O datasets are in use during compilation. These are:

- The text input dataset, $IN (by default)
- The source output dataset, $OUT (by default)
- The binary load-and-go dataset, $BLD (by default)
- A pseudo-CAL output dataset (if requested by the user)

After cracking the CFT control statement, CFT opens the datasets it will need, using system default parameters. If the datasets are already open, the system ignores the redundant OPEN request.

The compiler references a set of library routines to aid the user in manipulating dataset buffers. These logical I/O routines do all the driving of the buffers, make the system calls to fill the buffers, and keep track of the IN and OUT pointers. The I/O routines read or write strings of words or characters and make the system calls that allow physical I/O to be transparent to the user.

CFT first writes out the generated code in binary to a dataset called $BLD (by default) or its equivalent. This is used as input to the loader. Using the FORTRAN support library $FTLIB, the loader produces an executable program version of the information in $BLD.

CFT also produces a source listing on dataset $OUT containing a copy of the source input, compiler messages, and so on. Optionally, the user can also obtain a pseudo-CAL listing of the code in a form almost completely acceptable to the assembler; allowing the user to hand-optimize portions of the code if desired.

CFT uses the standard $SYSLIB routines $RCW and $WCW to perform all I/O.

APPENDIX SECTION

CHARACTER SET A

This appendix describes the 128 control and graphic characters comprising the ASCII character set. Those numbers, letters, and special characters that form the CRAY-1 FORTRAN character set are identified by the appearance of the letter C in the fourth column. All other characters are members of the auxiliary character set. The letter A in the fourth column of the table indicates those characters belonging to the ANSI FORTRAN character set. Note that all control characters are grouped on the first page.

In the description column, CC designates communication control, FE designates format effector, and IS designates information separator.

CONTROL CHARACTER	ASCII OCTAL CODE	ASCII PUNCHED-CARD CODE	FORTRAN (A=ANSI) (C=CRAY)	DESCRIPTION
NUL	000	12-0-9-8-1		Null
SOH	001	12-9-1		Start of heading (CC)
STX	002	12-9-2		Start of text (CC)
ETX	003	12-9-3		End of text (CC)
EOT	004	9-7		End of transmission (CC)
ENQ	005	0-9-8-5		Enquiry (CC)
ACK	006	0-9-8-6		Acknowledge (CC)
BEL	007	0-9-8-7		Bell (audible or attention signal)
BS	010	11-9-6		Backspace (FE)
HT	011	12-9-5		Horizontal tabulation (FE)
LF	012	0-9-5		Line feed (FE)
VT	013	12-9-8-3		Vertical tabulation (FE)
FF	014	12-9-8-4		Form feed (FE)
CR	015	12-9-8-5		Carriage return (FE)
SO	016	12-9-8-6		Shift out
SI	017	12-9-8-7		Shift in
DLE	020	12-11-9-8-1		Data link escape (CC)
DC1	021	11-9-1		Device control 1
DC2	022	11-9-2		Device control 2
DC3	023	11-9-3		Device control 3
DC4	024	9-8-4		Device control 4 (stop)
NAK	025	9-8-5		Negative acknowledge (CC)
SYN	026	9-2		Synchronous idle (CC)
ETB	027	0-9-6		End of transmission block (CC)
CAN	030	11-9-8		Cancel
EM	031	11-9-8-1		End of medium
SUB	032	9-8-7		Substitute
ESC	033	0-9-7		Escape
FS	034	11-9-8-4		File separator (IS)
GS	035	11-9-8-5		Group separator (IS)
RS	036	11-9-8-6		Record separator (IS)
US	037	11-9-8-7		Unit separator (IS)
DEL	177	12-9-7		Delete

CONTROL CHARACTER	ASCII OCTAL CODE	ASCII PUNCHED-CARD CODE	FORTRAN (A=ANSI) (C=CRAY)	DESCRIPTION
(Space)	040	(None)	A,C	Space (blank)
!	041	12-8-7		Exclamation point
"	042	8-7	C	Quotation marks (diaeresis)
#	043	8-3		Number sign
$	044	11-8-3	A,C	Dollar sign (currency symbol)
%	045	0-8-4		Percent
&	046	12		Ampersand
'	047	8-5	A,C	Apostrophe (closing single quotation mark)
(	050	12-8-5	A,C	Opening (left) parenthesis
)	051	11-8-5	A,C	Closing (right) parenthesis
*	052	11-8-4	A,C	Asterisk
+	053	12-8-6	A,C	Plus
,	054	0-8-3	A,C	Comma (cedilla)
-	055	11	A,C	Minus (hyphen)
.	056	12-8-3	A,C	Period (decimal point)
/	057	0-1	A,C	Slant (slash, virgule)
0	060	0	A,C	Zero
1	061	1	A,C	One
2	062	2	A,C	Two
3	063	3	A,C	Three
4	064	4	A,C	Four
5	065	5	A,C	Five
6	066	6	A,C	Six
7	067	7	A,C	Seven
8	068	8	A,C	Eight
9	069	9	A,C	Nine
:	072	8-2	A,C	Colon
;	073	11-8-6		Semicolon
<	074	12-8-4		Less than
=	075	8-6	A,C	Equal
|	076	0-8-6		Greater than
?	077	0-8-7		Question mark

CONTROL CHARACTER	ASCII OCTAL CODE	ASCII PUNCHED-CARD CODE	FORTRAN (A=ANSI) (C=CRAY)	DESCRIPTION
@	100	8-4		Commercial at sign
A	101	12-1	A,C	
B	102	12-2	A,C	
C	103	12-3	A,C	
D	104	12-4	A,C	
E	105	12-5	A,C	
F	106	12-6	A,C	
G	107	12-7	A,C	
H	110	12-8	A,C	
I	111	12-9	A,C	
J	112	11-1	A,C	
K	113	11-2	A,C	
L	114	11-3	A,C	
M	115	11-4	A,C	
N	116	11-5	A,C	Uppercase letters
O	117	11-6	A,C	
P	120	11-7	A,C	
Q	121	11-8	A,C	
R	122	11-9	A,C	
S	123	0-2	A,C	
T	124	0-3	A,C	
U	125	0-4	A,C	
V	126	0-5	A,C	
W	127	0-6	A,C	
X	130	0-7	A,C	
Y	131	0-8	A,C	
Z	132	0-9	A,C	
[	133	12-8-2		Opening (left) bracket
\	134	0-8-2		Reverse slant (backslash)
]	135	11-8-2		Closing (right) bracket
^	136	11-8-7		Circumflex
-	137	0-8-5		Underline

CONTROL CHARACTER	ASCII OCTAL CODE	ASCII PUNCHED-CARD CODE	FORTRAN (A=ANSI) (C=CRAY)	DESCRIPTION
`	140	8-1		Grave accent (opening single quotation mark)
a	141	12-0-1	C	Lowercase letters
b	142	12-0-2	C	
c	143	12-0-3	C	
d	144	12-0-4	C	
e	145	12-0-5	C	
f	146	12-0-6	C	
g	147	12-0-7	C	
h	150	12-0-8	C	
i	151	12-0-9	C	
j	152	12-11-1	C	
k	153	12-11-2	C	
l	154	12-11-3	C	
m	155	12-11-4	C	
n	156	12-11-5	C	
o	157	12-11-6	C	
p	160	12-11-7	C	
q	161	12-11-8	C	
r	162	12-11-9	C	
s	163	11-0-2	C	
t	164	11-0-3	C	
u	165	11-0-4	C	
v	166	11-0-5	C	
w	167	11-0-6	C	
x	170	11-0-7	C	
y	171	11-0-8	C	
z	172	11-0-9	C	
{	173	12-0		Opening (left) brace
¦	174	12-11		Vertical line
}	175	11-0		Closing (right) brace
~	176	11-0-1		Overline (tilde, general accent)

REGISTER USAGE B

The CRAY-1 computation section includes two basic types of operating registers: primary (A, S, and V) registers and secondary or intermediate (B and T) registers. CFT makes extensive use of the CRAY-1's high-speed intermediate storage registers to take full advantage of the CRAY-1's internal architecture. CFT uses vector registers to store compiler tables and working tables.

PRIMARY REGISTERS

CFT uses A registers A0 through A7, S registers S0 through S7, and V registers V0 through V7 as general purpose registers. Several of these registers have permanent uses throughout most of compilation. These include:

Register	Description
A6	A6 is usually a pointer to the source of an item.
A7	A7 is a pointer to the location where the result of a process will be stored. For example, when CFT is processing an expression such as B*C during pass 1, register A6 points to the B and A7 points to the location where the tag for B will be stored.
S6	During constant increment integer analysis in Pass 2, S6 contains the following information:

Field	Bits	Description
IS	32-33	Increment sign: 00 No increment 01 Plus 10 Minus 11 Ambiguous

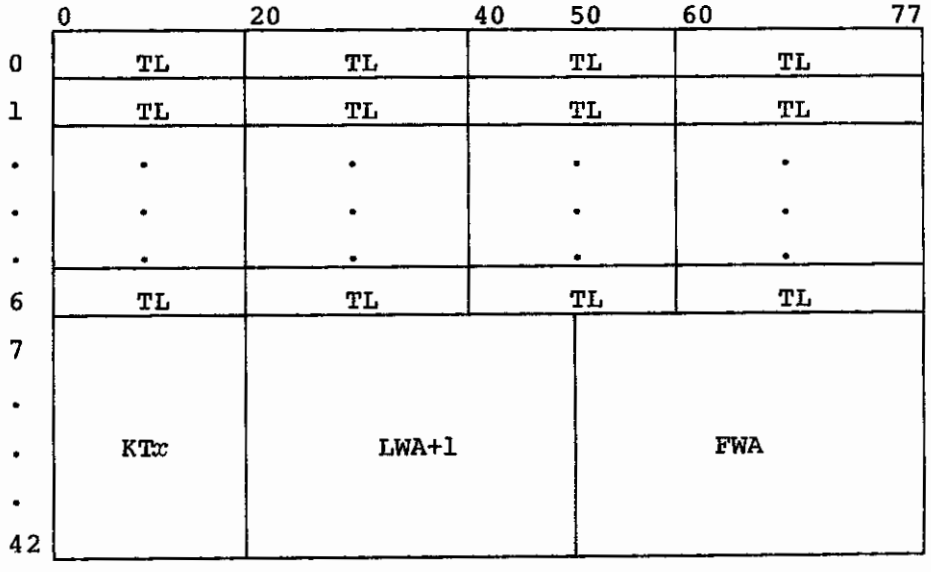
Register	Description			
		Field	Bits	Description
S6 (continued)		FLG	54-56	Flags: 54 Invariant encountered 55 Variant sign; if equal to 0, variant is positive 56 If equal to 1, address is self-referencing
		@TBY	57-77	Address of TBY table entry base

S7 In Pass 1, S7 tracks the current parenthesis level; for each new open parenthesis encountered, a new entry is made in S7 and the current entry is pushed onto a stack in V6 (refer to description of V6 in this section). In Pass 2, S7 is set block-by-block to contain generation mode flags. (Refer to section entitled Compiler Tables.)

V6 During Pass 1, a parenthesis stack is maintained in V6. Whenever a new open parenthesis is encountered, a new entry is saved on a stack that contains the current position and mode flags; then, whenever a close parenthesis is encountered, the last entry on the stack is popped. An entry is made for parenthesis limits (PL), subscript expressions ((s), subscripts ([s), and expressions (EXP). V6 is thus a normal push-down pop-up stack maintained by the compiler itself. Maximum stack depth is 77.

Field	Bits	Description
SBI	0-17	Statement buffer index
ATI	20-51	Array Table index or count. This field contains: Parameter count (PL) Constant count ((s) Array Table index ([s) Zeros (EXP)
PTF	52-63	Parenthesis group type flags
TL	64-77	Type and length

Register	Description
V7	Throughout compilation, register V7 contains an Intrinsic Type Table and table parameter words. Words 0 through 6 of V7 contain implicit type codes. Because each code takes 16 bits, each word contains the type codes for four characters. Thus, enough room exists to accommodate all 26 allowed characters (A through Z). The codes are preset during initialization but can be changed through use of an IMPLICIT statement. Starting in word 7 of V7 are the table parameter words, one for each table maintained by CFT. Each table parameter word contains FWA, LWA+1, and table index information.



Field	Word	Bits	Description
TL	0-6	0-17, 20-37, 40-57, 60-77	Implicit type and length; one entry for each letter (refer to tables section for more information).
KTx	7-42	10-17	Table pointer; index to the corresponding table in V7.

Register Description

Field	Word	Bits	Description
LWA+1	7-42	20-47	Last word address + 1 of corresponding table
FWA	7-42	50-77	First word address of corresponding table

V7 Table Parameter Words

There are three sets of register numbers, one set for the CFT 1.11 compiler version calling sequence, one set for the CFT 1.13 compiler version calling sequence, and one set for the CFT 1.14 compiler version calling sequence. The three sets are referred to as the CFT 1.11, CFT 1.13, and CFT 1.14 register numbers, respectively. Table B-1 lists the register numbers arranged alphabetically by symbol name.

Table B-1. Register numbers

Symbol	Description	CFT 1.11 register	CFT 1.13 register	CFT 1.14 register
KTA	Array Table (TBA)	44	51	57
KTB	Constant, Binary Table (TBB)	27	33	40
KTBK	Block Definition Table (TBBK)	None	None	16
KTBR	B-register Associates Table (TBBR)	7	16	23
KTC	Pointer Variable Table (TBC)	35	41	46
KTCALL	Register Variables to Restore After a CALL Table (TBCALL)	None	14	21
KTCLEN	Character Length Table (TBCLEN)	None	None	56
KTCLTXT	#CL Text Table (TBCLTXT)	41	45	52
KTCT	Conjunctive Term Table (TBCT)	None	None	15
KTD	DO-loop Table (TBD)	37	43	50
KTDT	Disjunctive Term Table (TBDT)	None	None	14
KTE	External Reference Table (TBE)	23	26	33
KTEE	Entry/Exit Address Table (TBEE)	None	31	36

Table B-1. Register numbers (continued)

Symbol	Description	CFT 1.11 register	CFT 1.13 register	CFT 1.14 register
KTF	Statement Function Skeleton Table (TBF)	40	44	51
KTFR	Call-by-value Reference Table (TBFR)	None	21	26
KTG	Tag Buffer Table (TBG)	30	34	41
KTH	Program Description Table (TBH)	26	32	37
KTI	TBA Index Table (TBI)	36	42	47
KTJ	Intrinsic Function Name Table (TBJ)	12	7	7
KTK	Intrinsic Function Attribute Table (TBK)	13	10	10
KTL	Library Name Table (TBL)	14	11	11
KTLB	Label Usage Table (TBLB)	None	None	17
KTM	Library Macro Table (TBM)	15	12	12
KTNOBT	TBT Index of Variables Not Assignable to B/T Register Table (TBNOBT)	None	None	21
KTO	Array Bounds Checking Table (TBO)	43	47	54
KTP	Common and Equivalence / Dummy Argument Address Table (TBP)	25	30	35
KTPD	Plus Dependency Table (TBPD)	None	22	27
KTPG	Program Unit Page Table (TBPG)	45	46	61
KTPN	Program Unit Name Table (TBPN)	45	52	60
KTQ	Declarator Expression Table (TBQ)	42	46	53
KTR	Packed EQUIVALENCE / Block Relocation Table (TBR)	24	27	34
KTS	Symbol Table (TBS)	31	35	42
KTSB	Substring Definition Table (TBSB)	17	15	22
KTSN	Sequence Number Table (TBSN)	None	45	50
KTSV	Saved Variable Table (TBSV)	26	31	36
KTT	Tag Table (TBT)	32	36	43
KTTR	T-register Associates Table (TBTR)	10	17	24
KTU	Symbol Cross Reference Table (TBU)	33	37	44

Table B-1. Register numbers (continued)

Symbol	Description	CFT 1.11 register	CFT 1.13 register	CFT 1.14 register
KTV	Cross Reference Overflow Table (TBV)	34	40	45
KTW	Triad Table (TBW)	16	13	13
KTWR	W-register Associates Table (TBWR)	11	20	25
KTX	Variable Reference Table (TBX)	20	23	30
KTXX	TBX Extension Table (TBXX)	None	None	20
KTY	Dependent Block Reference Table (TBY)	21	24	31
KTZ	Defined Block Variable Table (TBZ)	22	25	32

SECONDARY REGISTERS

The B and T registers are divided into two classes: temporary (variable) and permanent (fixed-global). The temporary registers begin at 67 and go downward, while the permanent registers begin at register 0 and go upward. Registers 70 through 77 are reserved for routines called by CFT and short term scratch usage within CFT.

TEMPORARY SECONDARY REGISTERS

Registers B50 through B67 and T50 through T67 are generally used as multipurpose scratch registers. The contents of these registers are not preserved over a long period of time. The B and T temporary registers are referenced by symbolic names as follows:

Register	Symbol	Register	Symbol
B67	B.ZA	T67	T.ZA
B66	B.ZB	T66	T.ZB
B65	B.ZC	T65	T.ZC
.	.	.	.
.	.	.	.
.	.	.	.
B50	B.ZP	T50	T.ZP

VOLATILE SECONDARY REGISTERS

Registers B70 through B77 and T70 through T77 are used as short term scratch registers. By convention, these registers are redefined after a long jump and by most subroutine calls. The registers are referenced symbolically by the symbols VA, VB, ..., VH.

PERMANENT SECONDARY REGISTERS

The permanent B and T registers begin at the 0 registers and go upward. The contents of many of these registers are preserved over long periods of time. Each permanent register is assigned a mnemonic label that indicates the value held in that register. By convention, these mnemonics are usually 2-letter names. For example, the B register containing field length is named B.FL.

Definition of the permanent secondary registers occurs at the beginning of the compiler through a set of equates. Although none of the B registers are preset, some of the T registers are. The values placed into the T registers can be found at the end of the compiler in the constant area preceding the intrinsic functions. The base address for these constants is ITRG. Keeping the ITRG area in a one-to-one correspondence with the definition of the registers at the beginning of the compiler is important.

Symbolic 24-bit B Register Definitions

Register	Symbol	Description
1	FL	Field length
2	SF	Special function return; tag.
3	TB	Tag Buffer base, Pass 2
4	SB	Statement begin (Tag Buffer end)
5	BI	Backward statement index
6	BB	Block begin
7	BE	Block end (address in TBG)
10	LI	Parenthesis level index
11	FP	Formal parameter count
12	DP	Dimension product
13	ND	Number of dimensions
14	SG	Statement group; @ compile block SN
15	DB	DO begin (Remnant length is EAFR)
16	EQ	Address of = in replacement statement
17	RL	Remnant length
20	RW	Read/write address tag
21	AP	Address of P operand
22	AQ	Address of Q operand
23	OC	Octal output flag
24	TXT	Pointer to start of loader Text Table in TBB
25	PD	Parameter or data expression flag
26	DC	Processing declarator flag
30	MT	Machine type from the JCB or control card
27-47	-	Work registers
50-67	-	Temporary registers

Symbolic T Register Definitions

Register	Symbol	Description
0	NU	Numeric character mask
1	AL	Alpha character mask

Common B and T Register Definitions

Register	Symbol	Description
3	TM	Tag Buffer entry mask
4	OF	Offset field mask
5	CT	Constant tag
6	CS	200,200, --
7	IZ	Integer zero constant
10	IU	Integer unity constant
11	ST	Statement number tag
12	FT	External function tag
13	PB	Program relocation block tag
14	CB	Common block tag
15	DA	Dummy argument tag
16	TT	Temporary tag
17	PC	Pointer count tag
20	FLG	Miscellaneous flags
21	PN	Program name
22	NT	Program name tag
23	PT	Previous tag
24	RN	Reference name
25	SH	Current statement header ISN
26	SN	Current statement number
27	TZ	Table Z PW base value in Pass 2 (keyword in Pass 1)
30	PR	Last used pseudo register
31	PA	code block address
32	BC	Blank common tag
33	SL	Segment length tag
34	LV	Current level of parentheses
35	SR	Secondary register
36	LA	Last word address for octal list output
37	AW	Partial assembly word, instructions
40	BR	Partial block relocation table assembly
41	PBS	Program base address
42	OCW	Output control word flags
43	CD	Compiler directive flags
44	DBT	Default block tag (static or stack block)
45-47	-	Work registers
50-67	-	Temporary registers

DEBUGGING AIDS C

The organization of CFT generally allows errors to be isolated. This appendix gives some information and methods that may be useful in debugging code.

Because CFT is almost completely reinitialized at the start of each program unit compilation, it is highly unlikely that a compilation bug in one routine was caused by compiler failure in a previous routine.

During compilation, CFT makes two passes through the source code.

PASS 1

Pass 1 reads the input deck, translating it into an internal notation, and then produces the source listing. Any syntax errors detected are listed immediately after the line in error. Generally, Pass 1 bugs can be reproduced by combining the line causing the error with any declaratives that refer to variables used in that line. The statements preceding or following usually have no effect on syntax bugs; an exception to this includes bugs related to blocking sequences (DO-CONTINUE or IF-ENDIF).

PASS 2

Pass 2 is the code generation pass. The optimizer breaks the program unit into code blocks and processes these blocks one at a time. Generally, if CFT makes an error in one block, all of the other blocks can be deleted from a test program and CFT will still fail.

An easy way to determine the block structure of a routine is to use the ON=B option on the CFT statement. This procedure is especially useful if the error is an operand range error occurring during compilation.

The method CFT uses to divide a program unit into code blocks is relatively straightforward. A code block begins either:

- With a statement that has a label referenced elsewhere in the program unit, or
- At an entry statement, or
- After a loop ends.

A code block ends just before the next one begins or on a statement that causes a backward branch to the start of the current block.

Some I/O statements that have unusual side effects also cause a new block to start. Examples are NAMELIST and BUFFER IN. Also, IF statements often force the start of a block because they implicitly jump around code.

The block that immediately precedes an innermost DO-loop is usually considered to be part of the DO-loop preamble for optimization purposes.

Pass 2 bugs generally consist of:

- Variables or intermediate results being saved in B or T registers across subroutine calls by mistake, or
- Calculations being mistakenly removed from a DO-loop, or
- Loop quantities not being properly saved for relooping.

RELEASE-RELATED BUGS

Sometimes, a program compiles and executes correctly with one version of the compiler but executes incorrectly with the next version. In such cases, it is best to talk to the programmer. If this does not isolate the problem, the binary search technique detailed below may be used to find the bad routine.

Suppose a job has 100 program units. Compile the job as in the following example:

```
JOB.
OLDCFT.
REWIND,DN=$IN.
REWIND,DN=$BLD.
SKIPR,DN=$BLD,NR=-50.
NEWCFT.
LDR.
```

This example compiles the first 50 program units of the job with the old version of CFT and the last 50 with the new CFT. If the job fails, then new CFT is compiling one of the last 50 routines incorrectly. By changing the NR parameter from -50 to -25 (and then halving this number again and again in suceeding compilations), the offending routine can be pinpointed. This binary search technique converges quickly and works even if there are several program units in error (although it will find only one of them).

STACK FRAME FORMAT

D

In stack mode, each routine uses a contiguous area of the stack at run time to hold its local variables. This stack area is a stack frame. A routine's frame format is described in figure D-1. Memory addresses increase downwards.

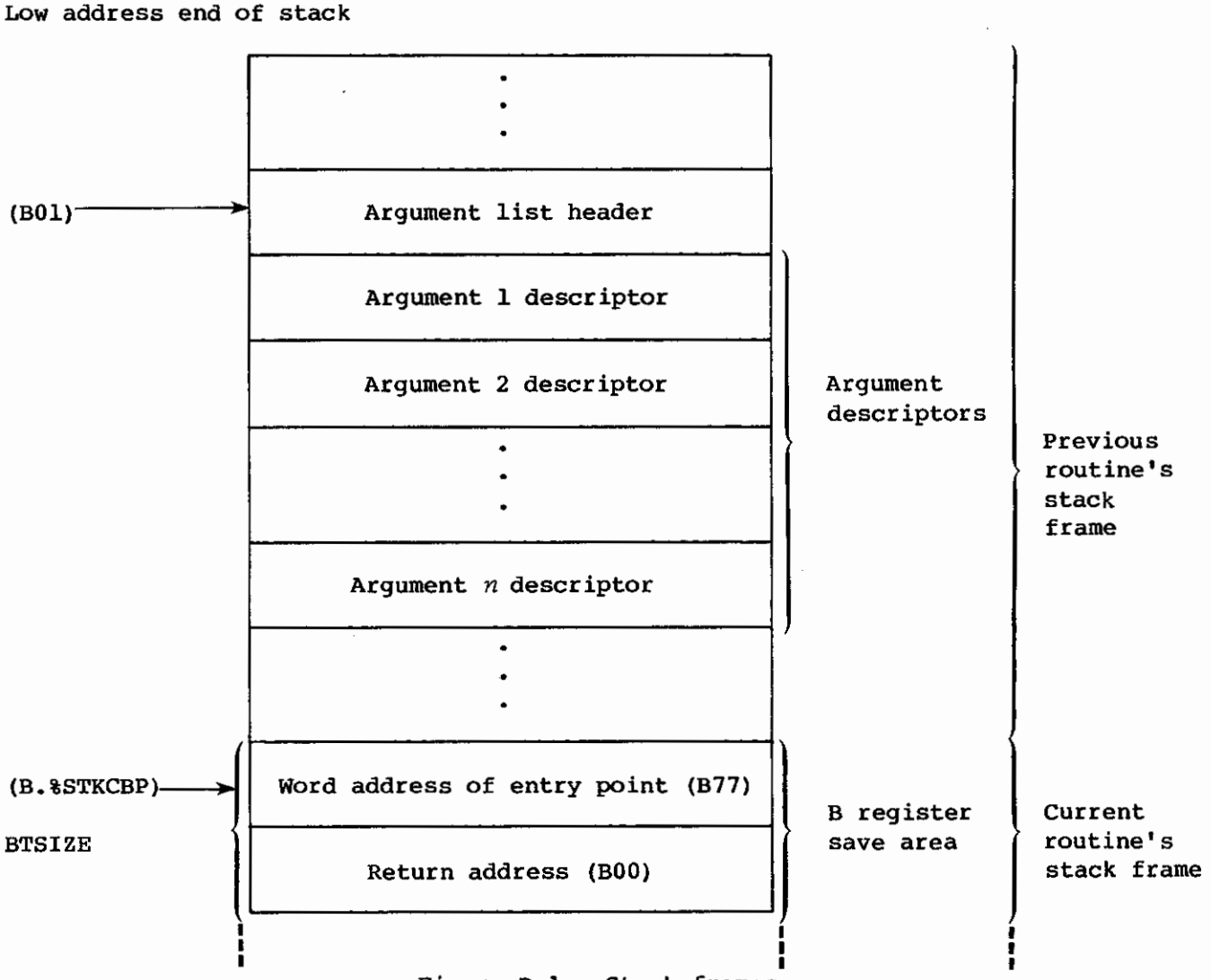


Figure D-1. Stack frames

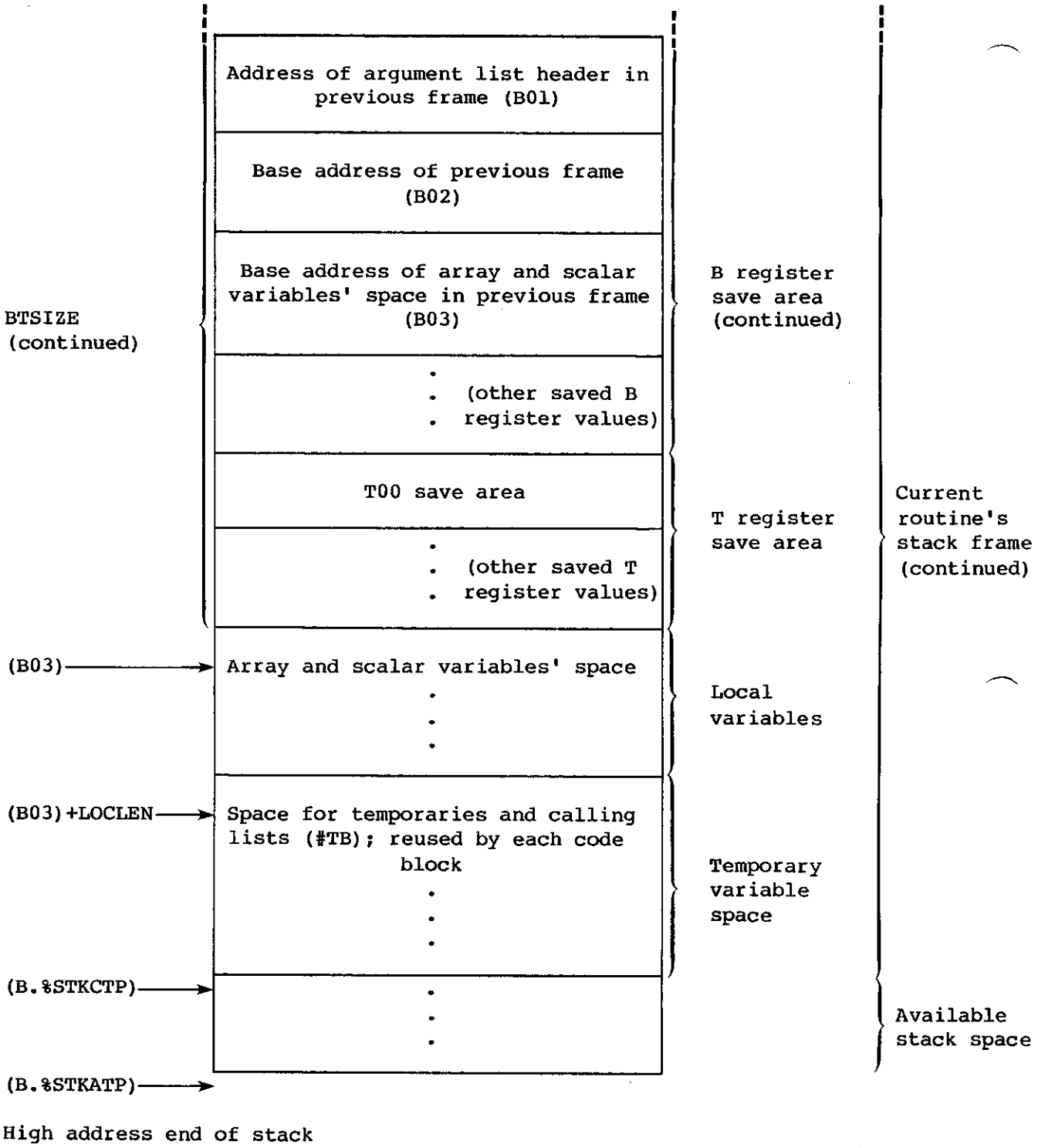


Figure D-1. Stack frames (continued)

LOCLEN is the amount of space reserved for local nontemporary variables and arrays. This includes all nonstatic user-declared variables and arrays, variables invented by the compiler which have meaning across code blocks, and the space into which arglists are copied for multiple-entry routines.

CFT must generate addressing code during Pass 2. However, the size of the B/T save area (appearing first in the stack frame) is unknown until the end of Pass 2. CFT solves the problem by setting B03 in the entry sequence to point at the first word following the B/T save area, and by treating all stack references as non-negative offsets from the run time value in B03. Stack address tags are identified by the tag field value 605; the terms stack tag and offset from B03 have the same meaning. B03 is comprised of B.%STKCBP and BTSIZE. BTSIZE contains the B and T register save areas.

CFT INSTRUCTION BUFFERS

E

The two instruction buffers used by CFT are the Pseudo Instruction Buffer (PIB) and the Final Instruction Buffer (FIB). They are described below with their formats.

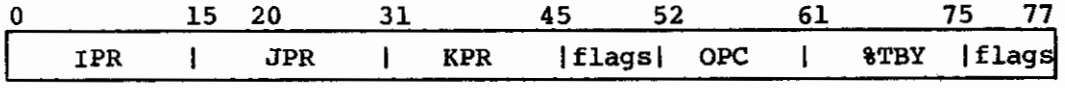
PIB holds code compiled in internal format until the code is scheduled and actual hardware registers assigned. All compiled instructions and statement numbers are entered in PIB by various code generation routines. The first part of SKED sorts the instructions into preamble, body, and postamble groups. The remainder of SKED and RASN assign hardware registers and reformat the entries into the FIB format.

FIB holds the generated code with assigned hardware registers. The instructions and statement numbers are unpacked and stored one to a word. FIB is produced by RASN and packed into the relocatable binary table by LTGN.

PIB formats

PIB has four formats. The first format described is used by most operations and the remaining three are used by the return jump/call, statement number, and pseudo statement number, respectively.

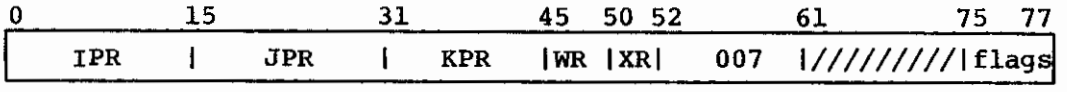
Format used by most operations:



Field	Bits	Description
IPR	0-14	I pseudo register or result pseudo register
JPR	15-30	J pseudo register
KPR	31-44	K pseudo register
Flags:	45-51	
IOP	45	I pseudo register used as an operand
JOR	46	J pseudo register used as an operand

Field	Bits	Description
KOP	47	K pseudo register used as an operand
IUR	50	Inhibits unrounding of floating-point operation
LBF	50	Loop Branch flag
DPR	50	Dependent memory reference
IRL	50	Inhibits release; used only with operation using IPR as operand. This is the Scheduling flag.
ITR	51	Inhibits truncation of arithmetic function
OPC	52-60	Operation code; if IPR≠0, the previous PIB entry is processed by MCEX.
%TBY	61-74	Index to TBY for store operations
Flags:	75-77	
PSF	75	Postamble flag
CNF	76	Constant pseudo register definition
PRF	77	Preamble flag

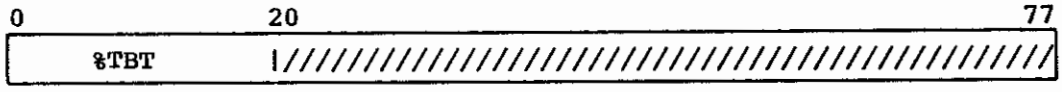
Return jump/call format:



Field	Bits	Description
IPR	0-14	I pseudo register or result pseudo register
JPR	15-30	J pseudo register
KPR	31-44	%TBX of destination if WR=6
WR	45-47	Call type: 4 Indirect through JPR 6 %TBX in KPR 7 Second word of CPX or DBL; complex and double-precision calls use a 2-word branch format. The second word which is not scheduled uses WR=7 to define the result pseudo register.

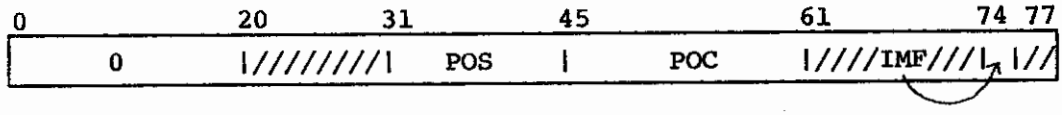
Field	Bits	Description
XR	50-51	Result of function: 0 None 1 A1 2 S1 3 V1
OPC	52-60	Operation code; 007.
Flags:	75,77	
PSF	75	Postamble flag
PRF	77	Preamble flag

Statement number format:



Field	Bits	Description
%TBT	0-17	Index into TBT

Pseudo statement number format:

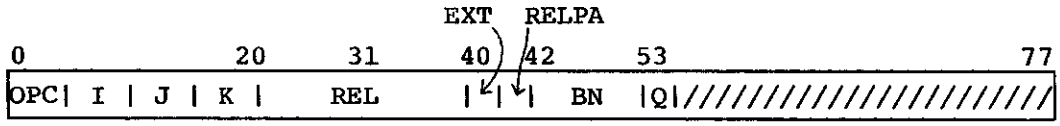


Field	Bits	Description
OPC	0-17	Operation code; 0.
POS	31-44	Pseudo operation subtype; applies only if POS≠0. 1 EFI 2 DFI 3 ERI 4 DRI 5 DBM 6 EBM 7 CMR
POC	45-60	Pseudo operation code; if POS≠0, POC>002.
IMF	74	Inhibits moving code over the pseudo statement number

FIB formats

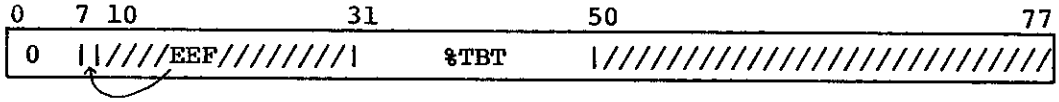
The FIB has two formats. The first format described is used for all operations except the statement number definition. The second format describes the format for statement number definition.

Format:



Field	Bits	Description
OPC	0-6	Operation code; if OPC=0, this is a statement number definition.
I	7-11	I register
J	12-14	J register
K	15-17	K register
REL	20-37	Address field
EXT	40	Set if reference is to an external name
RELPA	41	Relative to program counter relocation if RELPA=1; otherwise, normal relocation occurs.
BN	42-52	Block number (common)
Q	53	Quarter Word flag (for branch addresses)

Statement number definition format:



Field	Bits	Description
0	0-6	Statement number definition; 0.
EEF	7	Exit/Entry flag
%TBT	31-47	Index into TBT

INDEX

INDEX

READER COMMENT FORM

FORTRAN (CFT) Internal Reference Manual SM-0017 B-02

Your comments help us to improve the quality and usefulness of our publications. Please use the space provided below to share with us your comments. When possible, please give specific page and paragraph references.

NAME _____

JOB TITLE _____

FIRM _____

ADDRESS _____

CITY _____ STATE _____ ZIP _____

CRAY RESEARCH, INC.

CUT ALONG THIS LINE

FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY CARD

FIRST CLASS PERMIT NO 6184 ST PAUL, MN

POSTAGE WILL BE PAID BY ADDRESSEE

CRAY RESEARCH, INC.

Attention:
PUBLICATIONS

2520 Pilot Knob Road
Suite 350
Mendota Heights, MN 55120
U.S.A.

FOLD

READER COMMENT FORM

FORTRAN (CFT) Internal Reference Manual SM-0017 B-02

Your comments help us to improve the quality and usefulness of our publications. Please use the space provided below to share with us your comments. When possible, please give specific page and paragraph references.

NAME _____

JOB TITLE _____

FIRM _____

ADDRESS _____

CITY _____ STATE _____ ZIP _____

CRAY RESEARCH, INC.

CUT ALONG THIS LINE

FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY CARD

FIRST CLASS PERMIT NO 6184 ST PAUL, MN

POSTAGE WILL BE PAID BY ADDRESSEE

CRAY RESEARCH, INC.

Attention:
PUBLICATIONS

2520 Pilot Knob Road
Suite 350
Mendota Heights, MN 55120
U.S.A.

FOLD