HP 3000 Computer Systems

Systems Programming Language
textbook

LANGUAGES & SUBSYSTEMS

HP 3000 Computer Systems

Systems Programming Language

Textbook

HEWLETT hp *PACKARD*

LIST OF EFFECTIVE PAGES

The List of Effective Pages gives the most recent date on which the technical material on any given page was altered. If a page is simply re-arranged due to a technical change on a previous page, it is not listed as a changed page. Within the manual, changes are marked with a vertical bar in the margin.

Pages	Effective Date	Pages	Effective Date
Title	Jun 1976	3-3 to 3-4	Sep 1976
ii	Jun 1976	3-5 to 3-18	Jun 1976
iii to iv	Jan 1977	4-1 to 4-22	Jun 1976
v	Jun 1976	5-1 to 5-25	Jun 1976
vi	Jan 1977	6-1 to 6-32	Jun 1976
vii to xiii	Jun 1976	7-1 to 7-2	Jun 1976
xv to xvi	Jun 1976	7-3 to 7-5	Jan 1977
1-1	Jun 1976	7-6 to 7-12	Jun 1976
1-2	Sep 1976	8-1 to 8-4	Jun 1976
1-3 to 1-6	Jun 1976	8-5 to 8-8	Oct 1976
1-7	Jan 1977	8-9 to 8-17	Jun 1976
1-8	Jun 1976	A-1 to A-3	Jun 1976
1-9	Sep 1976	B-1	Jun 1976
1-10 to 1-11	Jun 1976	C-1 to C-2	Jun 1976
2-1 to 2-19	Jun 1976	D-1 to D-21	Jun 1976
3-1 to 3-2	Jun 1976		

PRINTING HISTORY

New editions incorporate all update material since the previous edition. Update packages, which are issued between editions, contain additional and replacement pages to be merged into the manual by the customer. The date on the title page and back cover changes only when a new edition is published. If minor corrections and updates are incorporated, the manual is reprinted but neither the date on the title page and back cover nor the edition change.

First Edition. . . . . Jun 1976
Update Package #1 . . . . Sep 1976
Update Package #2 . . . . Oct 1976
Update Package #3 . . . . Jan 1977
Update Incorporated . . . . Sep 1977

PREFACE

SPL/3000, Hewlett-Packard's systems programming language for the HP 3000 Computer System, is a high-level, machine-dependent programming language. SPL/3000 is designed for writing compilers, operating systems, and other systems software.

The *Systems Programming Language Textbook* is an introduction to the structure and uses of SPL. The textbook is written for the reader who has at least one year of programming experience with high-level and assembly-level programming languages.

Other publications which the reader should have available include:

Systems Programming Language Reference Manual (HP 30000-90024)

The authoritative reference source on questions of syntax and semantics about SPL.

General Information Manual (HP 30000-90008)

An integrated presentation of the features and capabilities of the software developed for the HP 3000 Series II Computer System.

MPE Commands Reference Manual (HP 30000-90009)

The authoritative reference source for commands to the MPE Operating System.

MPE Intrinsics Reference Manual (HP 30000-90010)

The authoritative reference for internal procedures of the MPE Operating System.

MPE Segmenter Reference Manual (HP 30000-90011)

The authoritative reference source for using the Segmenter.

Machine Instruction Set Reference Manual (HP 30000-90022)

The authoritative reference source for the machine instructions.

System Reference Manual (HP 30000-90020)

The authoritative reference on the organization of the HP 3000 Series II Computer System hardware.

This manual applies in its entirety to the HP 3000 Series II Computer System. Information in the manual may also be used with pre-Series II systems if the following changes are noted.

● Pre-Series II systems use three word representations of LONG data.

● Delete the following machine instruction mnemonics:

DISP	PSDB
IXIT	PSEB
LDEA	RCLK
LOCK	QASL
LSEA	QASR
LST	SCLK
MABS	SDEA
MDS	SSEA
MFDS	SST
MTDS	UNLK
PCN	

● Add the following machine instruction mnemonics:

Mnemonic	Function	Format
SIRF	Set external interrupt reference flag.	6
TSBM	Test and set bit in memory.	3b

For your convenience, these differences are noted in the manual.

CONTENTS

FIGURES

TABLES

INTRODUCTION

SPL/3000 has many features normally found only in applications languages such as ALGOL or PL/1: free-form structure, arithmetic and logical expressions, high-level statements with unlimited nesting (e.g., IF, FOR, SWITCH, CASE, DO-UNTIL, WHILE-DO, MOVE, SCAN, Assignment, and Compound statements), recursive procedures and subroutines, and variables and arrays of many different data types (byte, integer, logical, real, double integer, and long real).

Yet, SPL/3000, unlike any application language, allows the programmer to operate directly on hardware registers, perform branches based on hardware status, extract/deposit/shift bit fields, and generate any sequence of hardware machine instructions (all in the midst of high-level constructs).

This textbook introduces the reader to most SPL/3000 features. It is organized according to programming functions — from simple constructs to difficult — therefore, it should be read from beginning to end. Throughout the textbook complete example programs are listed and explained; these programs illustrate the topics discussed in the sections and should be studied carefully. Small examples of one or two lines are also provided; in many cases, the analysis is left to the reader. In addition, each section ends with a set of exercises to test comprehension. (Answers are provided in Appendix D.)

The topics covered in this textbook are:

Section I:	Environment of a HP 3000 Process
Section II:	Basic Elements of SPL/3000
Section III:	Transfer of Control
Section IV:	Looping Constructs
Section V:	Bytes, Pointers, Move and Scan
Section VI:	Procedures and Subroutines
Section VII:	Data Access Concepts
Section VIII:	Assemble Statement

Appendices contain the following information:

    A. ASCII Character Set

    B. Reserved Words

    C. Brief Summary of Commands

    D. Answers to Exercises

When you have mastered this textbook you should use the *System Programming Language* manual as your main source of information, since it contains a complete description of SPL/3000 and describes many special cases not covered in the textbook.

SECTION I
Environment of an HP 3000 Process

An SPL/3000 process is the unique execution of a program. If the same program is run by several users, it becomes several processes. If the same user runs the same program several times, each execution is a distinct process. All of the users on a system, therefore, can be executing the same program, but each execution has its own process. All the processes use the same copy of the program code, but each has its own data area. For example, if all users were running the BASIC/3000 Interpreter, their processes would overlap as follows:
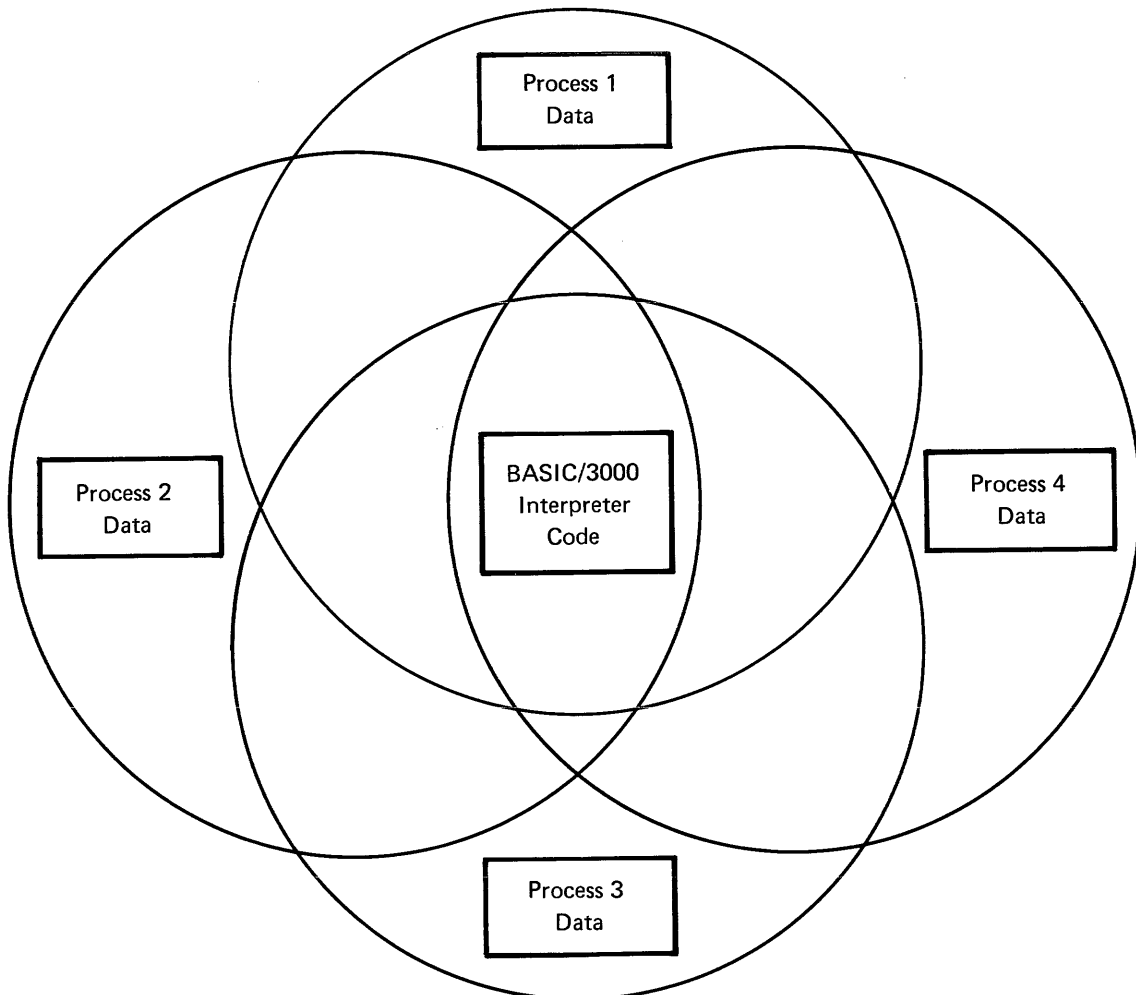
An SPL/3000 process consists of a code domain (some machine instructions that it can execute) and a data area called a "stack," and is run under control of the operating system. The code and data in HP 3000 are always separated logically. The code domain consists of segments which may be shared among several processes (for example, the code segments of the SPL/3000 compiler can be shared). The data stack is unique to the process and cannot be accessed by any other user's process.

The operating system (MPE/3000) schedules and dispatches a process for execution in competition with all other processes according to its priority.

THE CODE DOMAIN OF A PROCESS

All machine instructions within HP 3000 are organized into variable-length segments which are accessed through a hardware-known table. Since the hardware detects references to segments which are not in main memory, the code domain of a process is not limited to the size of main memory. At any one time there can be up to 256 different code segments defined in the system. Segments are brought from disc into main memory as needed. At any particular moment a process can be executing one and only one code segment. The process "escapes" from its current code segment by using a Procedure Call (PCAL) instruction. A PCAL can reference procedures in different code segments from the current one and cause control to transfer to a different code segment.

The current code segment of a process is defined by three hardware address registers:

PB—Program base register	Contains the absolute address of the starting location of the segment in main memory.
PL—Program limit register	Contains the absolute address of the last location of the code segment.
P—Program counter	Contains the absolute address of the instruction currently being executed.

The relationship of the three current code segment registers is shown in Figure 1-1. The central processor checks all instructions to insure that they stay within the bounds of the current code segment. All addresses within a current code segment are relative to these registers. The operating system can relocate the segment anywhere in main memory; only the three registers have to be changed to define the segment's locations.

Low memory

Program base register (PB)

Instructions
and constants

Program counter (P)

Program limit register (PL)

High memory

PB is the addressing base register; its absolute address is set by the operating system.

P changes as each instruction is executed

PL is the addressing limit register; its address is set by the operating system.
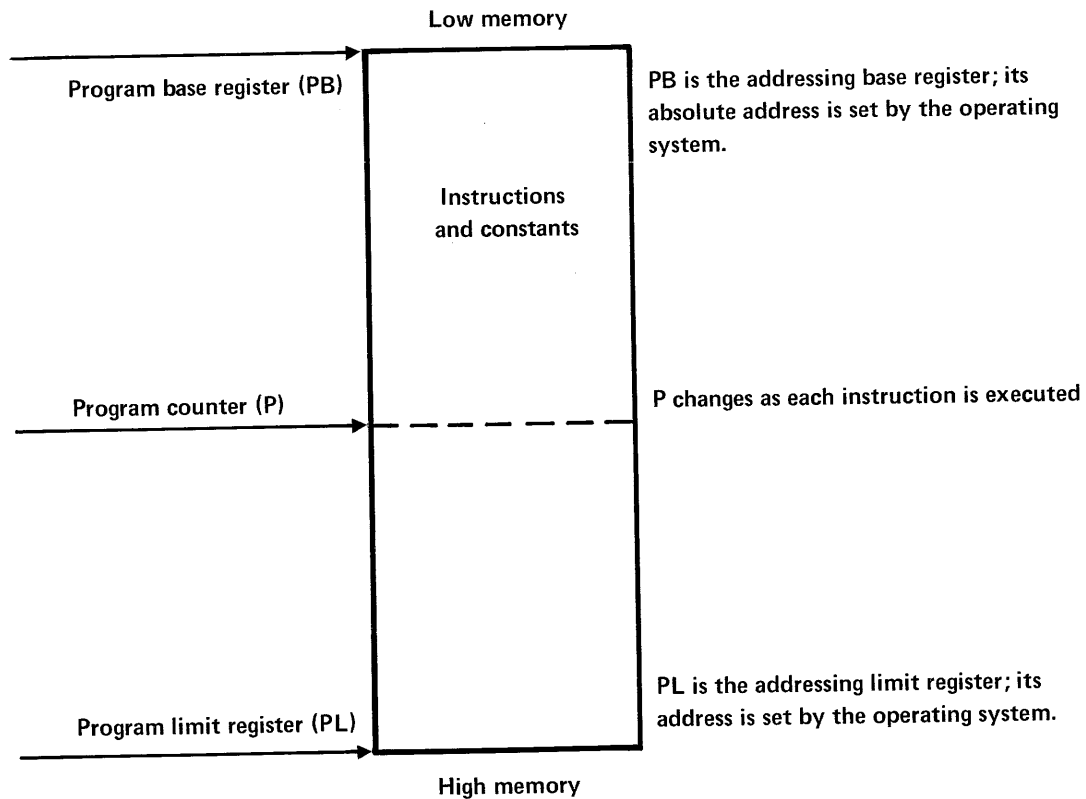
Figure 1-1. Current Code Segment Registers

THE DATA AREA OF A PROCESS

Each process has a completely private storage area for its data. This storage area is called a stack or a data segment. When the process is executing, its stack must be in main memory. A stack is delimited by two stack addressing registers:

DL—Data limit register   Contains the absolute address of the first word of main memory available in the stack.

Z—Stack limit register   Contains the absolute address of the last word of main memory available in the stack.

Between DL and Z, there are separate and distinct areas set off by three other stack addressing registers:

DB—Data base register   Contains the absolute address of the first location of the direct address global area of the stack.

Q—Stack marker register   Contains the absolute address of the current stack marker being used within the stack.

S—Top of stack register   Contains the absolute address of the top element of the stack. Manipulated by hardware to produce a last-in, first-out stack. (The top four items are kept in hardware registers.)

The relationship of the five stack addressing registers is shown in Figure 1-2. Each process is also described by a status register (containing its segment number and status) and a program-accessed, one-word index register used for array indexing and other computing functions.

There is only one set of these hardware registers; their content is established for a process when it starts executing.
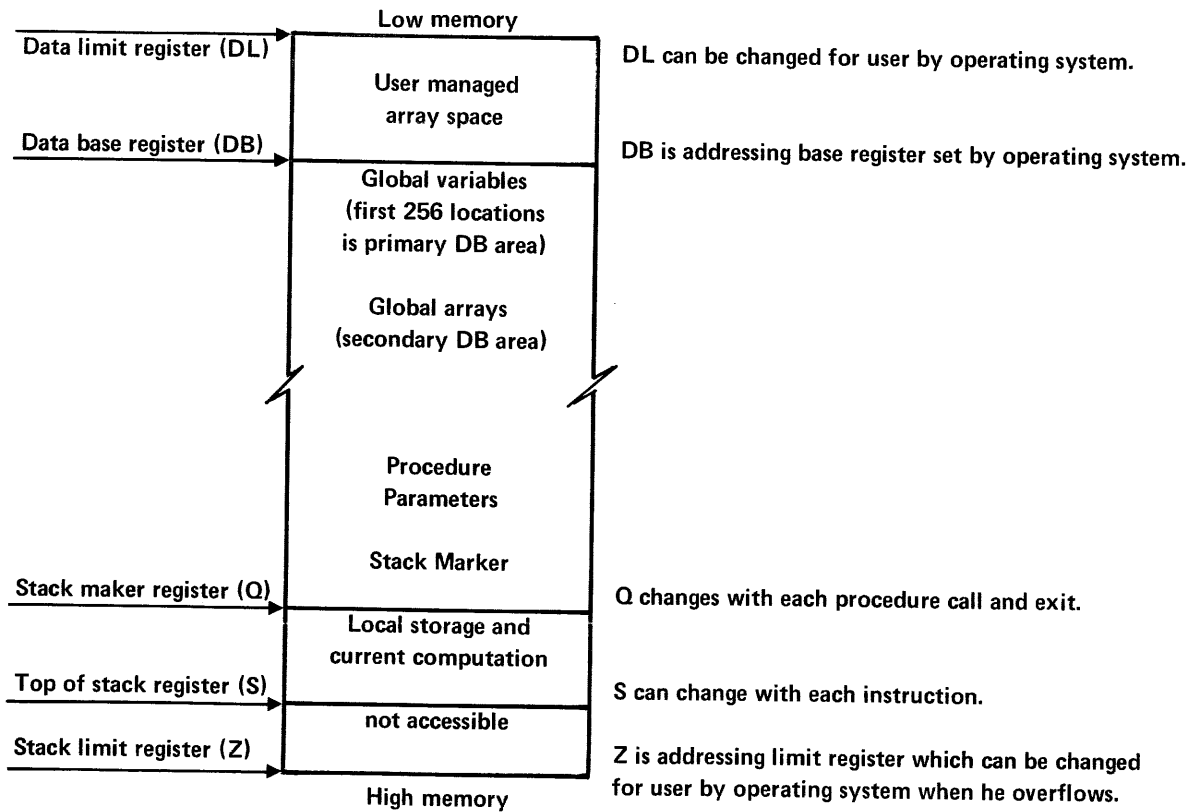
Figure 1-2. Data Stack Registers

HOW HARDWARE REGISTERS ARE USED

The hardware registers we have described (DL, Z, DB, Q, and S) are imposed on the main memory by the hardware to define the bounds of a process.

In the succeeding paragraphs, we show how these registers are referenced by the HP 3000 instruction set.

Memory Reference Instructions

In many conventional computers a memory reference instruction consists of an opcode (LOAD, STOR, ADD, etc.) and an address. The address is usually an absolute location or a location relative to a fixed hardware boundary. In HP 3000, memory reference instructions specify an address relative to one of the hardware registers and have direct addressing ranges of

P register
    ±255 locations

DB register
    +255 locations

Q register
    +127 locations
    −63 locations

S register
    −63 locations

Any memory reference instruction specifies a displacement within the range of one of these registers. This location is used as the operand; if another address is required, it is implicitly assumed to be the top of stack (S − 0).

To address beyond this range, the programmer specifies **indirect addressing** in the instruction and a location containing a DB or self-relative address. This allows him to access any area in the stack except that area between S and Z which is undefined (the DL to DB area is accessed by going indirect through a negative DB relative address).

The basic addressing mode in HP 3000 is word addressing (each word equals 16 bits). Also provided are instructions to load and store bytes (half words equal to 8 bits) and double words (32 bits).

The Stack

Many HP 3000 instructions use the top of the stack (the absolute address in the S register) as an implicit operand. The S register is constantly changing in a last-in, first-out manner such that data is "pushed" onto the stack or "popped" off the stack. "Push" and "pop" can be explained as follows:

PUSH

LOAD DB + 0 is a typical push operation. It causes the address in the S register to increment and then loads the contents of location DB + 0 into that address. The S relative addresses of all other items in the top of the stack are effectively decremented by one.

DB + 0	25	25	
S – 1	0	0	S – 2
S – 0	2	2	S – 1
		25	S – 0

(before) (after)

LOAD DB + 0
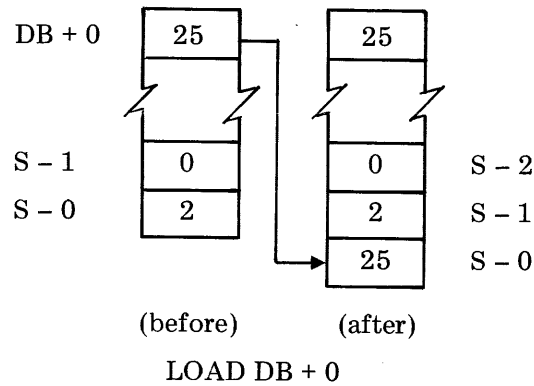
POP

STORE DB + 0 is a typical pop operation. It stores the contents of S – 0 in location DB + 0 and then decrements the address in S. This effectively increments the S relative addresses of all items by one.

		25	DB + 0
S – 2	0	0	S – 1
S – 1	2	2	S – 0
S – 0	25		

(before) (after)

STOR DB + 0
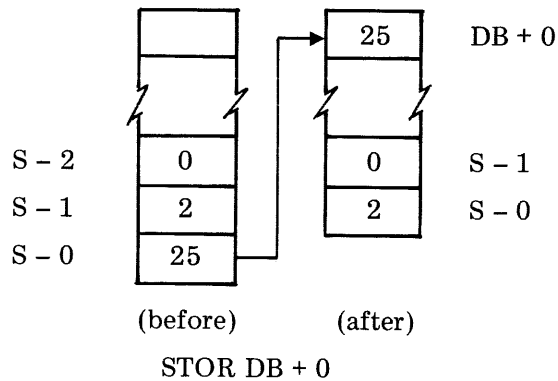
Note that after the "Pop," S – 0 contains the value 2, not 25.

Sixty-four of the HP 3000 instructions are called stack-ops, because they implicitly refer to the top items in the stack. A typical stack-op instruction is ADD, which adds the top two words in the stack (S – 0, S – 1), pops the operands, and pushes the result.

S – 1	15	22	S – 0
S – 0	7		

(before) (after)

ADD
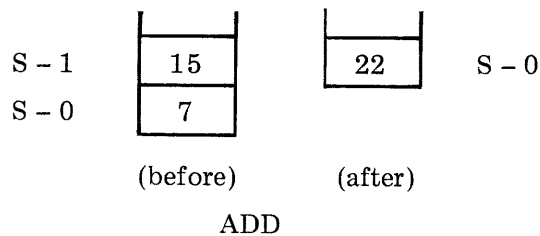
Because no bits are used for an operand address, stack-ops are very compact; in fact, they can be packed two per 16-bit word and allow HP 3000 to provide hardware operations for a broad range of hardware data types:

byte — 8 bits

integer — 16 bit signed

double integer — 32 bit signed

logical — 16 bit positive

floating-point — 32 bit (6.9 digits accuracy)

The firmware or software provides long floating-point arithmetic (64 bit) in Series II systems and long floating-point arithmetic (48 bit) in pre-Series II systems.

Indexing

The index register is a 16-bit data register which is accessible to the program. One use of the index register is for element indexing. When indexing is specified in memory reference instructions, the number in the index register specifies the element desired relative to the zero element (direct or indirect). The HP 3000 hardware provides true element indexing. If a double word is being accessed, the hardware doubles the index before adding it to the effective address and for byte indexing, the byte subscript value is added to the effective byte address.

Procedure Calls and Exits

The Q register plays an important part in the Procedure Call (PCAL) and Procedure Exit (EXIT) instructions. Whenever a procedure is called, a four-word stack marker is loaded onto the top of the stack. These four words preserve the state of the machine at the time of the call. The Q and S registers are changed to the top of this stack marker so that the procedure has a unique area for local storage allocation and addressing.

When the procedure exits, the stack marker (found by looking at Q – 0 to Q – 3) is used to restore the machine to its previous state (reset P, Q, X, and Status). All of this environment set-up and resetting is done by the two hardware instructions PCAL and EXIT.

EVALUATING AN EXPRESSION

Here is a typical HP 3000 expression:

A – (B * (C + D))

*NOTE: * means multiply.*

Assume that A, B, C, and D are integer variables in DB relative locations and that the result is to be stored in a variable R at DB + 0. One way to evaluate this expression using the stack is

1. LOAD DB + 1

    LOAD DB + 2

    LOAD DB + 3

    LOAD DB + 4

(Stack before 1)

		RESULT
DB + 0		
+ 1	457	A
+ 2	30	B
+ 3	10	C
+ 4	22	D
S – 0		

2. ADD

    (C + D)

(Stack before 2)

S – 3	457	Value of A
S – 2	30	Value of B
S – 1	10	Value of C
S – 0	22	Value of D

3. MPY

    (B * (C + D))

(Stack before 3)

S – 2	457	Value of A
S – 1	30	Value of B
S – 0	32	Value of (C + D)

4. SUB

    (A – (B * (C + D)))

(Stack before 4)

S – 1	457	Value of A
S – 0	960	Value of (B * (C + D))

5. STOR DB + 0

    (R := (A – (B * (C + D))));

(Stack before 5)

S – 0	– 503	Value of (A – (B * (C + D)))

(Final stack)

DB + 0	– 503	RESULT (A – (B * (C + D)))
+ 1	457	A
+ 2	30	B
+ 3	10	C
+ 4	22	D
S – 0		
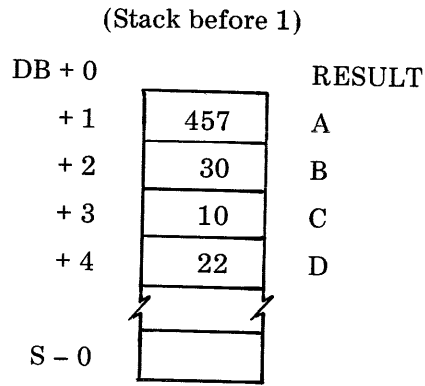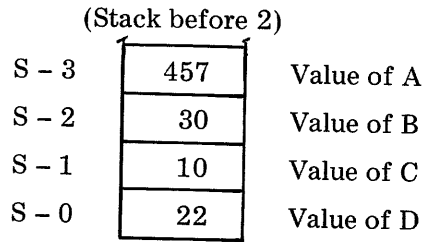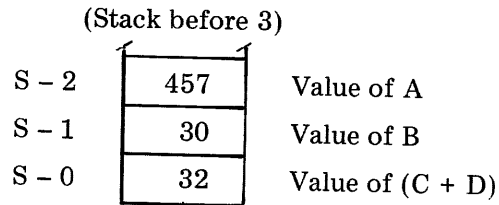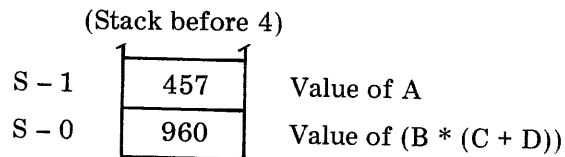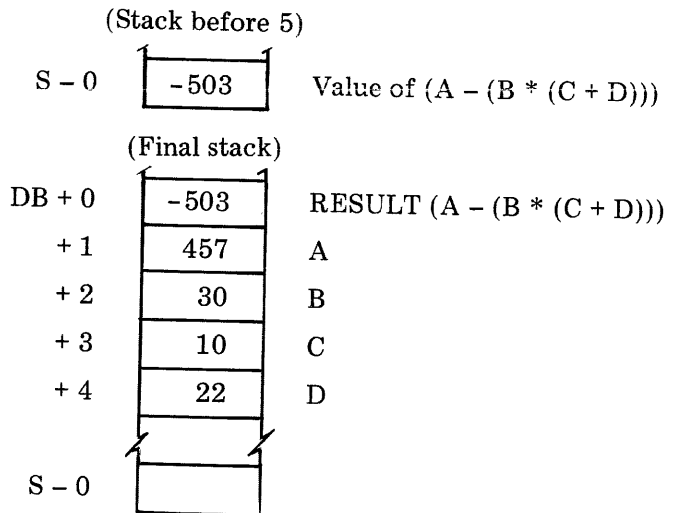
SUMMARY OF PROCESS ENVIRONMENT

Because of the many functions provided by the hardware and software, a HP 3000 process has many important services available to it.

Virtual Memory for Code

Since a process can reference procedures in up to 255 code segments without knowing whether they are present in main memory (the hardware checks for absence of code segments when they are referenced by the PCAL-EXIT instructions), programs are not unduly limited by the physical size of main memory. This ability constitutes a virtual memory for code based on variable length segmentation.

Dynamic Allocation of Local Storage

Local storage required by procedures is addressed relative to the Q register. Since the Q register is updated whenever a procedure is entered, local storage need not be allocated until then. Also, since Q is reset to its previous value when the procedure exits, this temporary storage is deleted. As a result, the amount of data storage required is kept to a dynamic minimum by the hardware.

Relocatability

Since all user instructions address relative to hardware registers, code and data segments can be relocated anywhere in memory simply by moving the information and setting the registers to new values. This provides great flexibility for the operating system.

Protection

Since the hardware checks that instructions stay within the bounds of the user's code and data segments, automatic protection is provided between user processes and the system. HP 3000 has other protection mechanisms (such as privileged versus user mode) which ensure that it maintains ultimate control.

Re-entrant Code

Since code is never modified during execution in HP 3000, it is all naturally re-entrant. This means that when a process is interrupted while executing, the system can allow another process to execute the same code. This is the primary advantage of re-entrant code; it eliminates the need for multiple copies of programs and reduces swapping between disc and main memory.

EXERCISES FOR SECTION I

1. How is a process different from a program?

2. What are the two main components of a process?

3. Explain the major difference between treatment of code and data in the HP 3000.

4. Label these diagrams with the appropriate register names. Describe briefly the function of each register.

Low
address

code segment

data segment

High
address   FUNCTION:

REGISTER   USED TO ADDRESS

_____   _____

_____   _____
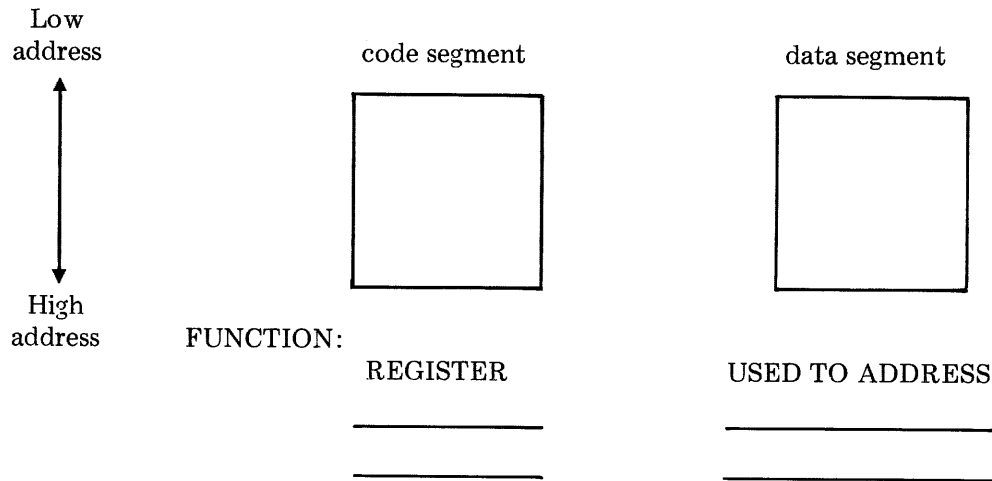
5. Match the addressing range limits shown below with the registers listed. (More than one limit value may be associated with a single register. Limit values may be used more than once.)

REGISTERS	ADDRESS LIMITS
P	+255
DL	−255
DB	+127
Q	−127
S	+63
Z	−63
	+31
	−31
	not used for addressing

6. Complete the stack diagrams provided to show the contents of the stack after each specified instruction has been executed

Assume A = 3, B = 1, C = 3, D = 1

LOAD A	LOAD B	SUB	LOAD C	LOAD D	ADD	MPY
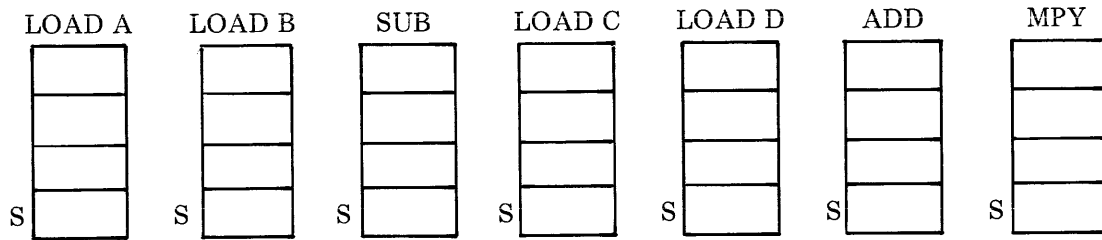
7. Write the expression which was evaluated in problem number 6.

   Use   * for multiply
          − for subtract
          + for add and
          ( ) to show which operation is done first.

8. For each of these capabilities, which feature of HP 3000 hardware makes the capability possible:

   a) Automatic relocatability?

   b) Protection for each user's code and data space?

   c) All code re-entrant?

SECTION II
Basic Elements of SPL/3000

This section introduces the building blocks from which SPL/3000 programs are made: constants, identifiers, delimiters, declarations, operators, expressions, and assignments. Once these concepts are understood it is possible to see how programs are structured in SPL/3000.

CONSTANTS

A constant is a fixed value of some specific type. The following are valid constants:

Constant	Type
1056.21	Decimal real number
1056	Decimal integer number
−539	Decimal integer number
%342	Octal (base 8) integer number
"ABDN"	String constant
"Now is the TIME."	String constant
.1023E6	Decimal real number = $.1023 \times 10^6$
.105621E4	Decimal real number (same value as first example, since E4 means "$\times 10^4$")

This list illustrates the three basic types of constants: integers (positive and negative, octal or decimal), real numbers (fractions with optional base ten exponent), and strings (sequences of characters). If the quote character (") is to appear within a string, it must be represented by a pair of quotes (" ") since the quote character is used to delimit strings.

Integer constants are 16-bit signed integers that can range from −32768 to +32767. Real constants require 32 bits and can range from -10^{77} to $+10^{77}(8.63616) \times 10^{-78} <= |N| <= 1.1579 \times 10^{77}$) with 6.9 digits of accuracy. There are several other SPL/3000 data types: bytes—8 bits, long real—64 bits,* logical—16 bit unsigned, and double integer—32 bits signed.

Two other constant forms:

%(2)010110 (binary)
%(16)0AE9 (hexadecimal)

*48 bits in pre-Series II systems.

IDENTIFIERS

Identifiers are the names used in an SPL/3000 program to represent various syntactic entities. Identifiers consist of up to 15 characters (uppercase alphabetic or numeric) and must begin with an uppercase alphabetic character:

    Valid Identifiers

    INVOICES

    NUMBER'OF'WOMEN   *(Note: ' is the only special character allowed within an identifier.)*

    ENTRY73

    A29765

    Invalid Identifiers

    9ABC

    Z1234567890123456789   {truncated to Z12345678901234}

    AB%9B

    'ABC

If lowercase letters are used, they are upshifted by the compiler to uppercase form by the compiler. Identifiers longer than 15 characters are truncated on the right. Certain combinations of characters are **reserved words**; they cannot be used as identifiers because they have predefined meaning within SPL/3000. A list of reserved words appears in Appendix B.

DELIMITERS

A blank character is generally recognized as a delimiter in SPL/3000, except for the occurrence of a blank in a character string. Thus, an identifier is delimited by one or more blanks on each end: . . . AN'IDENTIFIER . . . . In special contexts, characters other than blank can also serve as delimiters. For example, semicolons are used to delimit statements and colons can delimit a label.

COMMENTS

Comments can be embedded in a program in two ways: 1) through the COMMENT statement, or 2) through the use of $<<$ $>>$. The COMMENT statement consists of the reserved word COMMENT followed by any number of characters delimited by a semi-colon (;).

    COMMENT THIS is a COMMENT;

    COMMENT

        THIS COMMENT

        GOES ON FOR

        SEVERAL LINES;

The $\ll \gg$ characters can be used anywhere in a program (except within an identifier), and the characters between $\ll$ and $\gg$ will be ignored by the compiler.

  $\ll$THIS IS A COMMENT$\gg$

  $\ll$lowercase letters are NOT UPSHIFTED in comments$\gg$

DECLARATIONS

A declaration defines the attributes of an identifier before it is used in the program. All identifiers in SPL/3000 (except labels) must be declared and can be declared only once in a main program. A declaration generally consists of a reserved word specifying the attributes, followed by a list of identifiers (optionally set to initial values) that are separated by commas and delimited by a semi-colon. In this section we will only discuss two of the simplest declarations: simple integer variables and real variables.

  INTEGER A, B, C39, DUMMY;

  REAL SINTL,

    NUMEXTENTS;

The first example above declares four variables of type integer; the second example declares two variables of type real. The type indicates what type of arithmetic to perform on the variable (integer for INTEGER and floating-point for REAL) and how much storage to allocate (one word for each INTEGER variable and two for each REAL).

Initialization

Variables can be given initial values (when they are declared) through this simple construct:

  Valid

  INTEGER A := 2098, B := 4048, C := 8096;

  REAL X10 := 101.93, FGH, LINK;

  Invalid

  INTEGER A := B := C := 2;

  REAL 2.93, 3.25;

Any identifier in the list can be followed by := and a constant; this results in the variable having the value when the program is first executed. (Note that FGH and LINK are declared but not initialized.)

Global Address Mode

When variables are declared in the main program they are called **global variables**. Global variables are assigned consecutive locations (relative to the DB register) as they are declared. The number of words assigned is determined by the data type. Global variables can be referenced throughout the program. Variables with a more limited scope can also be declared (within a procedure); these variables are known as **local variables** and are discussed in Section VI.

EXAMPLE OF GLOBAL ADDRESS ASSIGNMENT

```
BEGIN  <<start of program>>
        INTEGER I,           <<location DB + 0 not initialized>>
                J,           <<location DB + 1 not initialized>>
                K := 11;     <<location DB + 2 set to 11>>
        REAL XIX;            <<location DB + 3 (2 words)>>
        INTEGER BUS;
        .
        .
        .
END.  <<end of program>>
```

ARITHMETIC EXPRESSIONS

Arithmetic expressions combine identifiers, constants, parentheses, and operators (^, *, /, MOD, +, −) to represent a sequence of mathematical operations. All operators, except the unary + and −, require an operand on each side—either a single value, such as a constant, or a variable or another expression enclosed in parentheses.

Valid Expressions

+C

A + B − CDG * 2.53

A − BC / FR

−(A + B) * C

−B * (−A)

Invalid Expressions

*A + C / ^ D

A(C + D)

A * − D

A * + D

An expression defines a single value which is its result; what is done with that value depends on where the expression occurs. In the most common case, an expression is assigned to a variable:

A := (A + B) * (C + D);

Arithmetic Operators

Arithmetic operators are symbols which are used in SPL/3000 to indicate the performance of some arithmetic operation. The operators are ranked in an order of precedence to determine the relative order in which operations are executed (when there are no overriding parentheses). When operations are of the same rank, execution proceeds from left to right. The rank, from highest to lowest precedence, is as follows:

Rank 1: Bit functions (extract, concatenate, shift)
Perform bit functions on one-, two-, or three-word quantities.

Rank 2: Unary plus and minus (+, −).

Rank 3: ^ Exponentiation
Raises a value to a specified power.

Rank 4: * Multiplication
Multiplies two values to produce a result.

/ Division
Divides one value by another to produce a result.

MOD Modulo
Divides one value by another and retains the remainder as the result (allowed only for one-word quantities—not real).

Rank 5: + Addition
Adds two values to produce a result.

− Subtraction
Subtracts one value from another to produce a result.

Bit extracts and concatenates are defined for one-word quantities only. Bit shifts are defined for one, two, or three words.

All other operations (except MOD) are defined for all data types with these exceptions: double integers can only be added and subtracted and exponentiate is not available with logical values.

The order in which operations are performed is determined by the hierarchy:

A − B + C
result
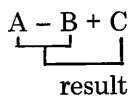
Operations of the same rank are performed from left to right.

A + B * C
result
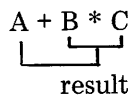
Operations of different rank are performed according to their position in the hierarchy of operators (highest rank first).

(A + B) * C
result
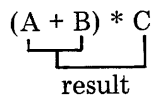
Operations enclosed in parentheses take precedence over operators outside of parentheses, even those of higher rank.

A − B + C * D ^ E
result
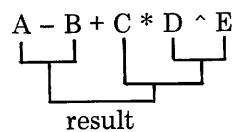
Left to right order is maintained until an operator occurs that is of lower rank than the next operator or the next item is in parentheses.

A ^ (B − C) * D / E MOD F ^ G
result
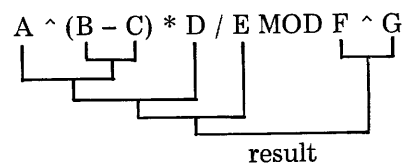
Absolute Value

When an expression is enclosed by backslashes (\ expression\), the result is the absolute value of the expression:

\A + B / C\ - D ^ 2

Type Transfer Functions

SPL/3000 does not allow type mixing in expressions. A particular operator can be executed only if both of its operands are of the same type. When a conflict occurs, it is necessary to convert the type of one of the values by means of type transfer functions.

> *NOTE: Exponentiate allows real raised to an integer power and long real raised to an integer power.*

A function is a subroutine or procedure that returns a value. Type transfer functions take a value of one type and return an equivalent value of another type; many of these functions are provided by hardware. The result of a type transfer is temporary (effective only for the immediate use of the value); the original contents of variables are not changed by type transfer. Not all possible transformations are provided; in some cases, two functions must be used.

- Convert an integer (double, byte, logical or long) expression to type real:
  REAL (*j*) (where *j* is any expression resulting in type integer, double, or long)

- Convert a real expression to type double (see section VI) and truncate:
  FIXT (*r*) (where *r* is any expression resulting in type real)

- Convert a real expression to type double and round:
  FIXR (*r*)

- Convert a double (or byte or logical) expression to type integer:
  INTEGER (*d*) (where *d* is any expression resulting in type double, byte, or logical)

- Convert a real expression to type integer:
  INTEGER (FIXT(*r*))   <<fix and truncate>>
  INTEGER (FIXR(*r*))   <<fix and round>>

All operands of an expression need not be converted to one type—only the pair surrounding each operator:

Assume that A, B, C, D are integers and M, N, O, P are real.

REAL(A + B) * M + P
INTEGER (FIXR(N / O)) + INTEGER (FIXR(M / P)) + C
M * REAL(A + B + INTEGER (FIXR(P)))

Primaries

Primaries are those items which can be operated upon by the arithmetic operators:

Constants

Simple variables

Array elements (Section IV)

An expression in parentheses (*arithmetic expression*)

The absolute value of an expression *arithmetic expression*\\

An assignment statement in parentheses (see "Assignment Statements" in this section)—the value assigned to the variable is also used in the expression to follow.

Function designators:
  Type Transfer Functions
  Bit Functions (this section), and
  Function Subroutines and Procedures (see section VI)

Bit Functions

Functions that perform bit-level operations can be used in arithmetic expressions. Bit functions are performed before other operators. In SPL/3000 expressions the three types of bit functions are

- Bit extraction (one word quantities only)
- Bit concatenation (one word quantities only)
- Bit shifts (one, two, and three word quantities)

BIT EXTRACTION

Bits are extracted to isolate a contiguous bit field from the 16 bits of a one-word value. The result is a right-justified value of type integer with the most significant bits set to zero. Primaries are not altered. The maximum field that can be extracted in a single operation is 15 bits. The format consists of a primary followed by a period and an extract field in parentheses:

primary . (*left bit* : *field length*)

In this example, two bits are extracted from %125 starting at bit 8:

primary = %125	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	0	0	0	0	0	0	0	0	0	1	0	1	0	1	0	1

result = %1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

%125.(8:2)
primary
left bit
field length

BIT CONCATENATION

Concatenation permits the formation of a new value by extracting a bit field from one word and depositing it at a specified position in another word.

The format is

primary CAT primary (*left deposit bit : left extract bit : field length*)
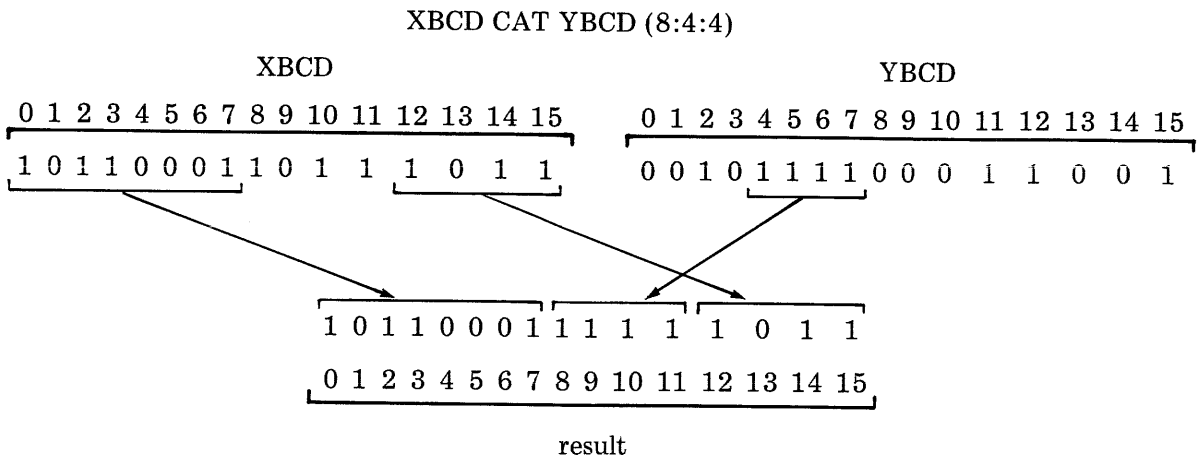
left deposit bit	Indicates which bit position in the first primary (to the left of CAT) to deposit the field extracted from the second primary.
left extract bit	Indicates at which bit in the second primary to begin extracting the bit field.
field length	Indicates how many contiguous bits to extract from the secondary primary.

Concatenation is performed on temporary quantities; the original primaries (if variables) are not altered. Note that the primaries themselves can be bit functions.

In this example, four bits are extracted from YBCD starting at bit 4 and are deposited in XBCD starting at bit 8. The other 12 bits of XBCD are carried over unchanged and the original values of XBCD and YBCD are unaltered.

XBCD CAT YBCD (8:4:4)

XBCD   YBCD

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15   0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

1 0 1 1 0 0 0 1 1 0 1 1 1 0 1 1   0 0 1 0 1 1 1 1 0 0 0 1 1 0 0 1

1 0 1 1 0 0 0 1 1 1 1 1 1 0 1 1

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

result

BIT SHIFTS

A bit shift performs a hardware shift (logical, arithmetic, or circular) upon a quantity in the stack. The format is

primary & shift op (count)

primary   is a primary of any type.

count   is an arithmetic expression specifying how many bits are to be shifted.

shift op   is one of the following hardware opcodes:

Single Word	LSL (logical shift left, 16-bit quantity)
	LSR (logical shift right, 16-bit quantity)
	ASL (arithmetic shift left, sign preserved)
	ASR (arithmetic shift right, sign preserved)
	CSL (circular shift left)
	CSR (circular shift right)
Double Word	DASL (double arithmetic shift left, 32-bit quantity)
	DASR (double arithmetic shift right)
	DLSL (double logical shift left)
	DLSR (double logical shift right)
	DCSL (double circular shift left)
	DCSR (double circular shift right)
Triple Word	TASL (triple arithmetic shift left)
	TASR (triple arithmetic shift right)
	TNSL (triple normalizing shift left)
Four Word	QASL (quadruple arithmetic shift left)
	QASR (quadruple arithmetic shift right)

> *NOTE:* *SPL/3000 does not check for type compatibility in bit shifts. If a double word shift is performed on a single word primary (or other unmatched combinations), other words in the stack will be shifted also.*

The difference between logical, arithmetic, and circular shifts can be explained as follows:

- Logical shifts fill with zero bits as they shift left or right.

- Arithmetic shifts preserve the sign bit on a left shift (and fill with zeros) and propogate the sign bit on a right shift (fill with the sign bit).

- Circular shifts have no fill bit; bits that are shifted off one end are shifted on at the other end.

This example shows the difference in practice:

AKS23 & LSR(3) →

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	0	1	1	0	0	0	1	1	1	1	1	1	0	1	1

AKS23

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	1	0	1	1	0	0	0	1	1	1	1	1	1

result

AKS23 & ASR(3) →

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	0	1	1	0	0	0	1	1	1	1	1	1	0	1	1

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	1	1	0	1	1	0	0	0	1	1	1	1	1	1

AKS23 & CSR(3) →

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	0	1	1	0	0	0	1	1	1	1	1	1	0	1	1

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	1	1	1	0	1	1	0	0	0	1	1	1	1	1	1

In all cases, the original contents of AKS23 remain unchanged.

ASSIGNMENT STATEMENTS

An SPL/3000 statement is an order to perform some action. The purpose of an assignment statement is to *assign* a value (determined by an expression) to a *variable*. These are typical assignment statements:

A := A + 1;

B := (A + C) / (N + 365 * NUM ^ 2);

The general form is

variable := *expression*;

where *variable* and the result of the *expression* must match in type to the extent that they use the same size quantity (1, 2 or 3 words) (type byte is compatible with one-word quantities) and

:= means "is replaced by"

; terminates the statement

Assignment statements have two special forms: multiple assignments and deposit fields.

Multiple Assignments

variable := variable := . . . . . := expression ;

For example the form

A := B := C := N * 2;

assigns the result of the expression to three variables.

Deposit Fields

A **deposit field** specifies a bit field of the variable into which the result of the expression is to be stored:

variable . (left deposit bit : field length) := expression;

For example, A.(8:2), and B2.(5:5). The number of bits required for the field is taken from the result of the expression (least significant bits) and deposited in the variable starting at the bit position specified. In this example, eight bits from the result of the expression are deposited in PAT starting at bit 0. (PAT contains %65137 before the deposit.)

PAT. (0:8) := *expression;*


```
0 1 0 0 1 1 1 0 0 1 0 1 0 1 1 1 1          0 1 0 0 0 0 0 1 0 1 0 0 1 1 1 0
```
New value of PAT   Result of *expression*

All bits in the deposit variable (PAT) which are not part of the deposit subfield remain unchanged. When deposit fields are used with multiple assignments, only the leftmost variable can have a deposit.

EXAMPLES OF ASSIGNMENT STATEMENTS

Bit Operations: INTEGER RSLT, DWORD, ANS, MPX, COMP;

RSLT := DWORD.(0:8) + %60;
ANS.(0:4) := MPX & LSR(3);
COMP := MPX CAT DWORD(14:0:2);

Arithmetic Operations: INTEGER CHAR, DATA, SLOPE;
REAL YPOINT, YVALUE, XPOINT, XVALUE, DELTA, DATA;

CHAR := DATA MOD 8 + %60;
YPOINT := YVALUE + REAL (SLOPE) * (XPOINT – XVALUE);
DATA := SLOPE * INTEGER (FIXR(DELTA));

EXAMPLE 2-1. SUM AVERAGE

This example solves a simple computational problem; it inputs two numbers, computes their sum and average and outputs the results. Example 2-1 illustrates the use of declarations, arithmetic expressions and assignment statements.

Example 2-1 uses several constructs which have not been covered yet:

Item	Description	Reference
BYTE	Indicates attribute "character"	Section V
ARRAY	A linear vector of elements of same type	Section IV

Input/Output

```
ENTER  VALUES
12,24
SUM = 36
AVG = 18
ENTER  VALUES
-50,50
SUM = 0
AVG = 0
ENTER  VALUES
-32768,+32767
SUM = -1
AVG = 0
```

Listing

```
BEGIN  <<EXAMPLE 2-1. SUM-AVERAGE>>
COMMENT:
        INPUT TWO INTEGER VALUES. COMPUTE THEIR SUM AND AVERAGE.
        OUTPUT THE RESULTS.
        NOTE: "INPUT" AND "OUTPUT" ARE DUMMY PROCEDURES WHICH SIMULATE
        INPUT, OUTPUT, AND CONVERSION - THEY ARE NOT PART OF SPL/3000;
BYTE ARRAY ENTR(0:11):="ENTER VALUES";
BYTE ARRAY LSUM(0:5):="SUM = ";
BYTE ARRAY LAVG(0:5):="AVG = ";
INTEGER AVALUE,
        BVALUE,
        SUM,
        AVG;
<<END OF DECLARATIONS>>
        OUTPUT(ENTR);   <<WRITE HEADING>>
        INPUT(AVALUE,BVALUE);   <<READ TWO NUMBERS>>
        SUM:=AVALUE+BVALUE;   <<COMPUTE SUM>>
        AVG:=SUM/2;   <<COMPUTE INTEGER AVERAGE>>
        OUTPUT(LSUM,SUM);   <<WRITE LABEL AND SUM>>
        OUTPUT(LAVG,AVG);   <<WRITE LABEL AND AVERAGE>>
END   <<SUM AVERAGE>>.
```

PROGRAM ORGANIZATION

A **program** is an organized collection of SPL/3000 declarations and statements designed to solve a specific problem. The structure of a program is shown in the following diagram:
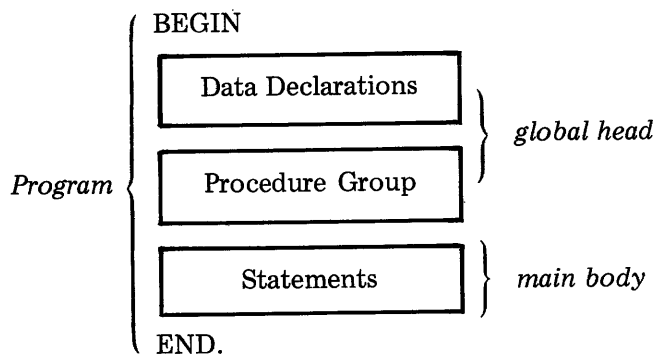
The **global head** consists of data declarations (INTEGER, REAL, and all other data to be used in the program) and then the procedure group (declarations of subroutines, procedures, and system intrinsic routines). Procedures are a variety of subroutine with special properties that make them almost as independent as programs. They are declared in the global head and then are called by statements in the main body or by statements within procedures themselves. Intrinsics are operating system procedures which can be called by user programs.

The **main body** consists of one or more statements such as assignment statements, for loop statements, etc. Statements are orders to perform some action; they are executed sequentially as written (unless one of the statements instructs otherwise). Statements are normally terminated by a semi-colon (;).

It is often useful to treat a group of statements as a single statement. In SPL/3000, any set of statements which is preceded by a BEGIN and followed by an END is called a compound statement and is treated as a unit. The BEGIN-END pairs act as parentheses—they define the collected statements as a unit (a single statement). A compound statement is similar in form to a complete program, *but declarations are not allowed within compound statements.*

Compound statements can be nested to whatever depth is necessary (the statements of a compound statement can themselves be compound statements).

Valid Compound Statements

```
BEGIN
      A := 1;
      B := 2;
END;

BEGIN
      A := 1;
      BEGIN
          B := 2;
          C := 3;
          D := A + B + C;
      END;
      Z := Z + 1;
END;
```

Invalid Compound Statements

```
BEGIN
      INTEGER A; <<declaration not allowed in compound statements>>
      A := A + J;
END;
```

The following example shows all the entries used in a program.

```
                         BEGIN
global  │ data dec.      { INTEGER I, J := 9;
head    │ proc. group    { PROCEDURE P(A) ; VALUE A; INTEGER A;
                               I := 2 + A;

main    │
body    │ statement      { P(2 * J) ; <<procedure call>>
                         END.
```

2-14

EXAMPLE 2-2. COMMAND INTERPRETER

This next example shows a moderately complicated SPL/3000 program containing many data declarations, three subroutines (GETCOMMAND, SKIPBLANKS, and ERROR), a large procedure (CMNDSRCH), and a main body. The program is a command interpreter that reads a command image, determines which command has been entered, and calls a procedure to handle the command.

The command interpreter accepts command images from an input device, skips initial blanks, interprets the next four characters as a command name, and searches a table to see if it is valid. If the command is valid, the command interpreter picks up a procedure call to the appropriate procedure and executes it.

NOTE: The command interpreter uses elements of SPL/3000 which are discussed later in the textbook. You may want to refer back to this example after reading each succeeding section.

Input/Output

```
STJB,A001,ENGR,WX
 FEB17-0810 JOB# 0026
JBID,JOB-26
 JOB#0026  SPOOLING IN
LOCT,JOB-7
 JOB#0007  UNDEFINED
HOLD,JOB-9
SPAC,PRINTER-1
REPEAT,PRINTER-1
 ERR 0
REPT,PRINTER-1
RSTT,PRINTER-1
RELS,JOB-9
JBID,0
 JOB#0009  SPOOLING OUT
 JOB#0011  I/O WAIT
 JOB#0015  EXECUTING
 JOB#0026  SPOOLING IN
```

Listing

```
BEGIN  <<EXAMPLE 2-2. COMMAND INTERPRETER>>
BYTE ARRAY IMAGE(0:81);
ARRAY WORDBUF(*)=IMAGE;
INTEGER LENGTH,
        ERCODE;
BYTE POINTER IMAGPTR:=@IMAGE;
BYTE ARRAY EBUF(0:3):="ERR ";
ARRAY MNEM(0:41):=%100000,%200001,  <<STOPPER>>
                  "ALTR","BKLG","BKSP","DLTD","DLTJ",  <<1>>
                  "DRIN","DSPL","HOLD","JBID","LIST",  <<6>>
```

2-15

```
                    "LOCT","RELS","REPT","RSTT","SPAC",    <<11>>
                    "STAT","STJB","STOP","STRT",           <<16>>
                    %077777,%177777 <<STOPPER>>;           <<20>>
        LABEL READ;
        DEFINE PROCHEAD = (OPERANDPTR,ERRORCODE);
                    VALUE OPERANDPTR;
                    BYTE POINTER OPERANDPTR;    <<ADR COMMAND OPERAND>>
                    INTEGER ERRORCODE;          <<ERROR NUMBER RETURN>>
                    OPTION EXTERNAL
                    #;
        DEFINE DUPLICATE = ASSEMBLE(DUP)#;
        PROCEDURE ALTR PROCHEAD;    <<CHANGE JOB PRIORITY>>
        PROCEDURE BKLG PROCHEAD;    <<DISPLAY QUEUED JOBS>>
        PROCEDURE BKSP PROCHEAD;    <<BACKSPACE DEVICE>>
        PROCEDURE DLTD PROCHEAD;    <<DELETE DEVICE OUTPUT>>
        PROCEDURE DLTJ PROCHEAD;    <<CANCEL JOB>>
        PROCEDURE DRIN PROCHEAD;    <<DRAIN PHYSICAL DEVICE>>
        PROCEDURE DSPL PROCHEAD;    <<DISPLAY SYSTEM INFORMATION>>
        PROCEDURE HOLD PROCHEAD;    <<PREVENT EXECUTION OF A JOB OR JOBS>>
        PROCEDURE JBID PROCHEAD;    <<DISPLAY JOB INFORMATION>>
        PROCEDURE LIST PROCHEAD;    <<MESSAGE LISTING CONTROL>>
        PROCEDURE LOCT PROCHEAD;    <<LOCATE A JOB IN THE SYSTEM>>
        PROCEDURE RELS PROCHEAD;    <<RELEASE HELD JOB OR JOBS>>
        PROCEDURE REPT PROCHEAD;    <<REPEAT CURRENT I/O OPERATION>>
        PROCEDURE RSTT PROCHEAD;    <<ABORT AND RESTART CURRENT OPERATION>>
        PROCEDURE SPAC PROCHEAD;    <<SINGLE SPACE DEVICE>>
        PROCEDURE STAT PROCHEAD;    <<DISPLAY ACTIVITY STATUS>>
        PROCEDURE STJB PROCHEAD;    <<START A JOB>>
        PROCEDURE STOP PROCHEAD;    <<STOP ALL OPERATIONS>>
        PROCEDURE STRT PROCHEAD;    <<START OPERATIONS>>
        LOGICAL PROCEDURE CMNDSRCH;
        BEGIN  <<BINARY SEARCH FOR COMMAND MNEMONIC>>
        INTEGER N1:=0, N2:=20, K=X;
        BYTE ARRAY T1(0:3)=Q;
        LABEL NEXT,LESSL,FOUND,EXIT;
                MOVE T1:=IMAGPTR,(4);    <<TRANSFER COMMAND MNEMONIC>>
        NEXT:   IF N1+1 = N2 THEN RETURN;  <<NOT FOUND>>
                K:=(N1+N2)&LSR(1);          <<FIND TABLE MIDPOINT>>
                ASSEMBLE(DDUP;LDD MNEM,I,X;DCMP);  <<COMPARE MNEMONICS>>
                IF = THEN GOTO FOUND;
                IF < THEN GOTO LESSL;
                N1:=K;    <<MOVE LOWER TABLE LIMIT UP>>
                GOTO NEXT;
        LESSL:  N2:=K;    <<MOVE UPPER TABLE LIMIT DOWN>>
                GOTO NEXT;
        FOUND:  ASSEMBLE(LOAD P+1,X);   <<LOAD CORRESPONDING PCAL>>
                GOTO EXIT;
                ASSEMBLE(
                    PCAL ALTR; PCAL BKLG; PCAL BKSP; PCAL DLTD; PCAL DLTJ;
                    PCAL DRIN; PCAL DSPL; PCAL HOLD; PCAL JBID; PCAL LIST;
                    PCAL LOCT; PCAL RELS; PCAL REPT; PCAL RSTT; PCAL SPAC;
                    PCAL STAT; PCAL STJB; PCAL STOP; PCAL STRT
                <<END ASSEMBLE>>);
```

```
EXIT:     CMNDSRCH:=TOS;   <<RETURN PCAL INSTRUCTION>>
END   <<CMNDSRCH>>;
SUBROUTINE GETCOMMAND;
BEGIN
          @IMAGPTR:=@IMAGE;   <<RESET POINTER TO START OF BUFFER>>
          WORDBUF:=[8/%15,8/%15];   <<RETURN,RETURN>>
          MOVE WORDBUF(1):=WORDBUF,(40);   <<SET IMAGE TO RETURNS>>
          INPUT(IMAGE);   <<READ A COMMAND>>
END   <<GETCOMMAND>>;
SUBROUTINE SKIPBLANKS;
BEGIN
          SCAN IMAGPTR WHILE %6440,1;
          IF CARRY THEN ASSEMBLE(DEL,ZERO);   <<BLANK LINE - RETURN 0>>
          @IMAGPTR:=TOS;   <<RETURN POINTER OR 0>>
END   <<SKIPBLANKS>>;
SUBROUTINE ERROR(NUMBR);
     VALUE NUMBR;
     INTEGER NUMBR;
          OUTPUT(EBUF,NUMBR);   <<PRINT DIAGNOSTIC>>
<<START OF MAIN CODE>>
READ:     GETCOMMAND;
          SKIPBLANKS;
          IF @IMAGPTR=0 THEN GO TO READ;   <<BLANK LINE>>
          TOS:=CMNDSRCH;   <<GET APPROPRIATE PCAL>>
          DUPLICATE;
          IF TOS=0 THEN BEGIN
                            ERROR(*);   <<ERR 0 -INVALID COMMAND>>
                            GOTO READ;
                        END;
          @IMAGPTR:=@IMAGPTR+4;   <<SET POINTER BEHIND MNEMONIC>>
          SKIPBLANKS;            <<LOCATE OPERAND STRING>>
          TOS:=@IMAGPTR;         <<PASS POINTER BY VALUE, 0 IF NO OPRNDS>>
          TOS:=@ERCODE;         <<PASS ERROR CODE BY REFERENCE>>
          ASSEMBLE(XEQ 2);      <<CALL PROCEDURE>>
          DEL;   <<REMOVE PCAL INSTRUCTION FROM STACK>>
          IF ERCODE>0 THEN ERROR(ERCODE);
          GOTO READ;            <<READ ANOTHER STATEMENT>>
END   <<COMMAND INTERPRETER>>.
```

EXERCISES FOR SECTION II

1. Some of these declarations contain errors. Change those which are invalid so that they will be valid SPL/3000 declarations:

   a) INTEGER I, J := 100, K;

   b) REAL NUMBER, INTEGER SUM := 0;

   c) INTEGER A := 123.456, TESTER := % 6412;

   d) REAL ZED := B := C := 0;

   e) INTEGER INTEGERB := % 102;

   f) INTEGER 7BETA = "A";

   g) INTEGER BIGGE := 35767;

   h) SAM := 0;

   i) REAL BEGIN := 1.414;

   j) INTEGER $MIKE := 1;

2. Assume these declarations:

   BEGIN
       REAL NUMBER := 123.45, CRUNCHER := 456.67, TOTAL;
       INTEGER I := 3, K := 6, J := –1, SUM, TEMP,M;

   Some of the assignment statements that follow contain errors. Change those which are invalid so that they will be valid SPL/3000 assignments:

   a) SUM := I + NUMBER / 2;

   b) TOTAL := \ (NUMBER ^ 2 – CRUNCHER) \ / (3.1417/2.);

   c) TEMP := K + <<ADD M TO SUM>>SUM – I;

   d) TEMP := (I + J) – K <<SUBTRACT AND SAVE> ;

   e) TOTAL := REAL (I + J) ^ K / \ CRUNCHER * REAL (TEMP) \ ;

   f) TOTAL := NUMBER ** 2 + CRUNCHER ** 2;

   g) TEMP := K * SUM := I + K + J;

   h) SUM := M MOD8

3. Assume these declarations:

BEGIN
      LOGICAL NBITS := % 177777, R1 := 0, R2 := 0, R3 := 0, R4 := 0, R5 := 0;
      LOGICAL AB := "AB", CD := "CD", EF := "EF";

What is the result in each of the assignment statements below?

a) R1.(1:4) := NBITS;

b) R2 := AB CAT CD (8:8:8);

c) R3 := (AB CAT CD(0:0:8)) CAT (CD CAT EF(0:0:8)) (8:8:8);

d) R4.(15:1) := AB.(8:8) + AB.(0:8);

e) R5.(13:3) := % 423.(13:3) + NBITS.(4:3) + % 100000.(0:3);

4. Assume these declarations:

BEGIN
     INTEGER AB := "AB",
         RA := 0, RB := 0, RC := 0, RD := 0, RE := 0;

What is the octal result in each of these assignment statements?

a) RA := ((AB.(8:8) + 3) & LSL(8) CAT AB(8:0:8)) & CSL(8);

b) RB.(8:8) := -1.(0:1) & ASR(8);

c) RC := "1".(13:3) &LSL(3) + "2".(13:3) & LSL(3) + "3".(13:3) & LSL(3);

d) RD.(15:1) := RE := AB & CSR(8) CAT "Z"(13:13:3);

5. Reorder the lines of code shown below to restructure the program correctly.

     REAL PROCEDURE RAND; OPTION EXTERNAL;
     REAL ANS, X := 1.414, Z := .256;
     BEGIN
        ANS := RAND * X - Z;
     END.

6. What are the five components of an SPL/3000 program?

7. Which of the five program components cannot be omitted from a program?

8. What is a compound statement? How is it treated by the SPL/3000 compiler?

SECTION III
Transfer of Control

In this section these topics related to transfer of control are introduced:

- Labels
- GO TO Statement
- Logical Expressions
- Logical Assignment
- IF Statement
- IF Assignments

LABELS

In order for one statement to refer to another it must be able to identify that statement. In SPL/3000, labels are used to identify statements.

A label must always be followed by a colon (:) to separate it from the statement that it identifies. Here are some typical uses of labels:

```
START: BEGIN
           A := B + 1;
           B := C * 2;     (labels are underlined with dashes for illustration)
       END;
 STOP: ABCOND := 0;
```

Labels are declared in the data declaration portion of the program. One label can be used to identify only one statement in the program. The format of a label declaration is

    LABEL *list of identifiers* ;

For example,

    LABEL START, STOP;
    LABEL SPIN;

> *NOTE: Labels do not have to be declared in SPL/3000 unless the programmer prefers to declare them for consistency and documentation. When they are not declared, labels declare themselves when they are used.*

Position of Labels

In general, the position of a label is valid if it precedes or follows a statement:

Valid Label Position

START: A := B;

IF A < B THEN START2: X := Z;

BEGIN

     START3: A := B;

     X := Z;

     STOP:

END;

Invalid Label Position

A := B + C * START: X + 1;

A := B + C * (START: X := Z + 2 * X);

GOTO STATEMENT

The GOTO statement transfers control to a location somewhere in the program specified by a label. The labeled statement can occur before or after the GOTO and can itself be a GOTO statement. GO *label*, GOTO *label*, and GO TO *label* are equivalent formats.

For example,

GOTO LINKSPOT; <<or GO TO or GO>>

     .

     .

     .

LINKSPOT: A := A + C;

LOGICAL EXPRESSIONS

A logical expression is similar in form to an arithmetic expression—it consists of logical constants, logical variables, logical functions, and logical operators. The purpose of a logical expression is to evaluate certain conditions and relations to produce a value which can be interpreted either arithmetically (as a positive number) or logically (as TRUE or FALSE). A logical expression is not a statement of fact but rather an assertion that may be true or false at any given time.

Logical Constants

Logical quantities in SPL/3000 are 16-bit positive integers. Operations on logical values are provided by the hardware for addition, subtraction, multiplication, division, complement, and comparison. A logical value is considered TRUE if its integer value is odd, FALSE if its value is even (that is, only the last bit is checked when the result of a logical expression is used to make a decision).

Logical constants are TRUE, FALSE, or any integer constant.

Use of TRUE and FALSE is equivalent to the numeric values -1 and 0 (177777_8 and 000000_8).

Logical Variables

An identifier is declared to be type LOGICAL by means of a declaration:

LOGICAL A := "Z.", $\ll$initialized$\gg$
                B := TRUE,
                C, $\ll$not initialized$\gg$
                D := %17;

Whenever a logical identifier is used in the program, the compiler recognizes that it is type LOGICAL and is designed for logical values and arithmetic. Logical variables can be initialized in the same manner as integer and real variables, as shown in the preceding example.

Logical Operators

Logical operators have the following hierarchy (those at the top of the list are performed first in the absence of overriding parentheses; when operators have equal rank, execution occurs from left to right):

Rank 1:   NOT Unary one's complement
          Primaries (see "Forming Logical Expressions")

Rank 2:   * Logical multiply
          / Logical divide
          MOD Logical modulo (remainder)

Rank 3:   + Logical add
          – Logical subtract (no unary logical minus)

Rank 4: $=, >, <, <>, >=, <=$
          Algebraic or logical comparison
          Byte Testing (see Section V).

Rank 5:   LAND Logical conjunction

Rank 6:   XOR Logical exclusive or

Rank 7:   LOR Logical inclusive or
          Integer range check ($a <= b <= c$)

UNARY COMPLEMENT

NOT is an operator which requires only one operand. The value of each bit of the operand is changed in a one's complement fashion. (All one bits are reset to zero, and all zero bits are set to one.)

LOGICAL ARITHMETIC

Logical arithmetic (+, −, *, /, MOD) assumes that the 16-bit operands are always unsigned numbers. A logical value can range from 0 to 65,535 decimal (0 to 177777 octal), but has no sign. A signed integer, on the other hand, can be positive or negative, but (since it has only 15 bits for magnitude) can only range from 0 to 32,767 (and 0 to −32,768). The following example shows the difference between the logical and integer interpretations of a 16-bit pattern.

$$177777_8 \quad \equiv \quad +65535 \quad \equiv \quad -1$$
octal   logical   integer
value   interpretation interpretation

Thus, the same binary pattern can have two interpretations, a logical value and an integer value. This diagram shows how the logical and integer number lines overlap:

LOGICAL NUMBER LINE

```
o--------------------+-----------------------------o- - -
0              +32,767 +32,768              +65,535
```

INTEGER NUMBER LINE

```
|--------------------o------------------------------+- - -
0              +32,767 -32,768                    0
```
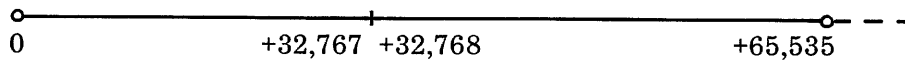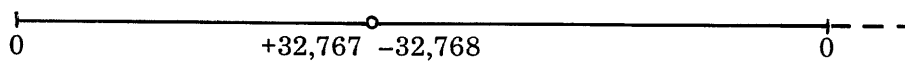
      (Two's complement)

Patterns that would be negative integers if used as integer numbers are treated as very large logical values, and vice versa. For exact details on all types of arithmetic consult the HP 3000 hardware documentation.

RELATIONAL OPERATORS

There are two types of relational comparisons in SPL/3000—logical and algebraic. Logical compares are formed by placing one of the relational operators between two logical quantities. Algebraic compares are formed by placing one of the relational operators between two arithmetic expressions:

  logical relation logical
  arithmetic relation arithmetic

The possible relations are: equal (=), less than (<), greater than (>), not equal (<>), less than or equal (<=), and greater than or equal (>=). Logical compares use the LCMP instruction to perform a 16-bit comparison which treats the sign bit as a data bit. Algebraic compares use one of three instructions (CMP, DCMP, FCMP) to perform comparisons taking into account the sign bit (negative numbers are considered to be smaller than positive numbers). The difference can be seen in this example:

177777_8 LCMP 1 <<177777_8 is greater>>

177777_8 CMP 1 <<1 is greater (since $177777_8 = -1$)>>

The result of a compare is a TRUE or FALSE value (–1 or 0):

100 > 5 → TRUE (–1)

100 < 5 → FALSE (0)

BYTE TESTING

SPL/3000 logical expressions allow the testing of bytes (characters) for various conditions. The complete set of byte testing and comparing facilities is discussed in Section V. Elements of the set allow testing when a string equals another string and when a particular byte is of a particular type (alphabetic, numeric, or other). The result is always TRUE or FALSE.

For example,

INBUF = "/*"LOR INBUF = "AB" <<STRING COMPARISONS>>

CHAR = ALPHA <<Is CHAR alphabetic?>>

CHAR <> NUMERIC <<Is CHAR not numeric?>>

RANGE COMPARISON

The following format tests whether a particular integer expression lies within a range determined by two other integer expressions:

$a <= b <= c$

a, b, and c are integer expressions.

The result is TRUE (–1) only if b is both less than or equal to c and greater than or equal to a. In all other cases the result is FALSE (0). This test is performed by the CPRB (compare range and branch) instruction which uses the index register.

10 <= 20 <= 30 → TRUE (–1)

10 <= 5 <= 30 → FALSE (0)

Forming Logical Expressions

Logical expressions are formed by combined logical primaries, logical operators, and true/false tests on arithmetic expressions. Primaries can be any of the following:

Constants — TRUE LOR 23

Logical variables — ABC / Q39

Logical bit operations — ABC & LSR(2) / 3

Logical expressions in parentheses (ABC + 2) / 3

Logical primaries preceded by NOT — NOT ABC
                                                    NOT ((ABC + 2) / 3)
                                                    NOT (ABC & LSR(2))

Logical assignment statements — ABC + (ABC := ABC LOR CBA)

NOTE: The value is assigned to the variable and used in the expression to follow.

Tests on arithmetic expressions (e.g., relational operators or compare range test) can also be used in logical expressions as primaries:

(100 <= A <= 200) LAND (0 <= B <= 23)

(2 < A) LOR (B <= 3)

The primaries in a logical expression are evaluated from left to right with execution following the hierarchy of operators (high rank before low rank.)

In general, the result of a logical expression is left as a one-word operand on the top of the stack. An exception is when a relational operator is encountered in which case the value of −1 (TRUE) or 0 (FALSE) is left on the top of the stack. Another exception is when the result of a relational operator is used to make a decision in a conditional branch (see "IF Statement"); in this case nothing is left in the stack and the status register is examined for the result.

For example (assume logical variables in all cases),

Valid

(A LOR B) LAND C * (N23 < N24)

A + B = C LAND D

NOT A MOD B LOR C = D LAND E

NOT ((A MOD B LOR C = D) LAND E)
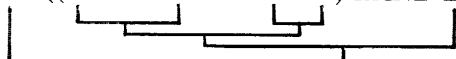
Invalid

A * LAND B

A (B LAND C)

–A LAND B

Type Transfer Functions

Logical expressions can be used in arithmetic expressions by converting them with type transfer functions:

REAL (*logical expression*)

INTEGER (*logical expression*)

DOUBLE (*logical expression*)

BYTE (*logical expression*)

BYTE and INTEGER do not change the 16-bit value of the logical expression; it merely says to use it as an integer value. DOUBLE expands the 16-bit logical result into a 32-bit positive double integer. REAL floats the value to a 32-bit floating point.

Arithmetic expressions can be used in logical expressions by converting them to type logical with type transfer functions

LOGICAL (*integer expression*)

LOGICAL (*double integer expression*) {leaves 32 bit value; special case; see *Systems Programming Language* manual}

LOGICAL (FIXR (*real expression*))

LOGICAL (FIXT (*real expression*))

LOGICAL (*byte expression*)

ASSIGNMENT OF LOGICAL EXPRESSIONS

Logical expressions can be assigned to one-word variables (type logical, integer, and byte) with an assignment statement:

LOGICAL EXIT, LASTRECORD, COMMAND, ANS;

INTEGER LOLIMIT, DATA, HILIMIT, DELTA;

EXIT := LASTRECORD LOR NOT (LOLIMIT <= DATA <= HILIMIT);

COMMAND := DATA < DELTA

ANS := LOGICAL (DELTA) + COMMAND * EXIT;

DATA := (EXIT := %60) + (ANS := FALSE) LOR (%170707 LAND COMMAND);

IF STATEMENT

When logical expressions are used with the IF statement, they give the programmer a very powerful tool for changing the order of execution on the basis of what happens during execution. The IF statement chooses which of two statements to execute based on whether a certain condition is true. For example,

```
IF LASTRECORD LOR NOT (LOLIMIT <= DATA <= HILIMIT)
    THEN
        BEGIN
            EXIT := TRUE;
            LASTVALUE := DATA;
        END
    ELSE
        HILIMIT := DATA;
```

There are two formats for the IF statement:

IF *condition* THEN *statement*; <<Format 1>>

IF *condition* THEN *statement*

ELSE *statement*; <<Format 2>>

condition   consists of logical expressions and/or machine-dependent tests

statement   is any SPL/3000 statement including compound statements (BEGIN-END) and IF statements (unlimited nesting of IF statements is allowed)

Format 1

In this case, control is transferred to the statement following the THEN if the condition is TRUE. If the condition turns out to be FALSE, control falls through to the first statement following the IF statement.
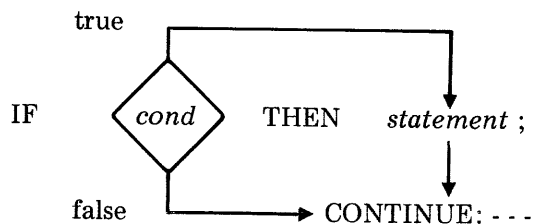
For example,

```
IF A < B THEN NX := A + B;
IF NOT (FINAL LOR SUSPICIOUS) THEN
    BEGIN
        TEST'DONE := FALSE;
        GO TO AGAIN;
    END;
```

Format 2

In this case, there are two alternative statements within the IF statement. If the condition turns out to be TRUE, control transfers to the statement following THEN; if the condition turns out to be FALSE, control transfers to the statement following ELSE.
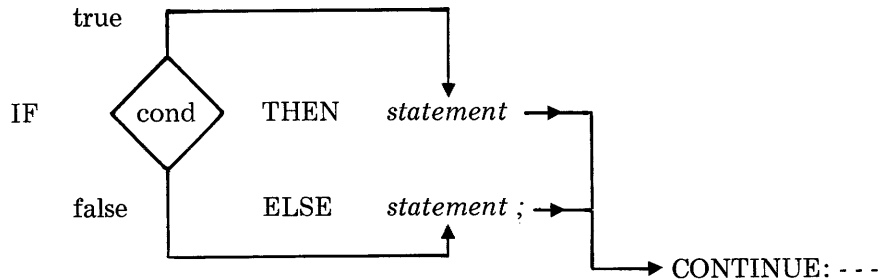
When the statement chosen is complete, execution continues with the first statement following the IF statement.

Valid

IF A < B THEN XA := XA + A
        ELSE XA := XA + B;

IF TESTVAR THEN Y := Y + 1
        ELSE IF EXTRATEST THEN Y := Y – 1;

Invalid

IF TEST THEN A := A + B; ELSE A := A – B;
        <<; should not precede ELSE>>

IF Conditions

The conditions which are used in IF statements to make decisions are composed of two items: logical expressions and branch words. Logical expressions have been covered already; branch words are hardware-dependent branch conditions:

Branch Word	True Condition	
CARRY	Carry bit on	
NOCARRY	Carry bit off	
OVERFLOW	Overflow bit on	
NOVERFLOW	Overflow bit off	
IABZ	Increment TOS (S – 0). True if result is zero.	
DABZ	Decrement TOS (S – 0). True if result is zero.	
IXBZ	Increment Index Register. True if result is zero.	See Section IV
DXBZ	Decrement Index Register. True if result is zero.	

Branch Word	True Condition
<	Condition code equals 1
=	Condition code equals 2
<=	Condition code equals 1 or 2
>	Condition code equals 0
<>	Condition code equals 0 or 1
>=	Condition code equals 0 or 2

Logical expressions and branch words can be combined using two special branch operators:
OR and AND. AND has precedence over OR except that this can be overruled by using parenthesis around the OR:

IF (CARRY OR OVERFLOW) AND A = 0 THEN . . . .

OR and AND generate branch instructions such as BCC, and BOV. They never generate arithmetic instructions such as LOR and LAND. All parts of the condition may not be executed every time, since OR and AND branch out of the condition as soon as the truth value of the condition is determined.

For details on the implications of branch words (overflow, DXBZ, condition codes, etc.) consult the hardware documentation.

Nested IF Statements

IF statements can be nested to any level; they are evaluated in the following manner: the innermost THEN is paired with the closest following ELSE and pairing proceeds outwards.

```
IF condition
    ┌ THEN
    │     IF condition
    │         ┌ THEN
    │         │     IF condition
    │         │         ┌ THEN statement
    │         │         └ ELSE statement
    │         └ ELSE statement;
    └ <<NO OUTERMOST ELSE PART>>
      <<COMPOUND STATEMENTS CAN BE USED TO OVERRIDE STANDARD
        PAIRING>>
```

3-10
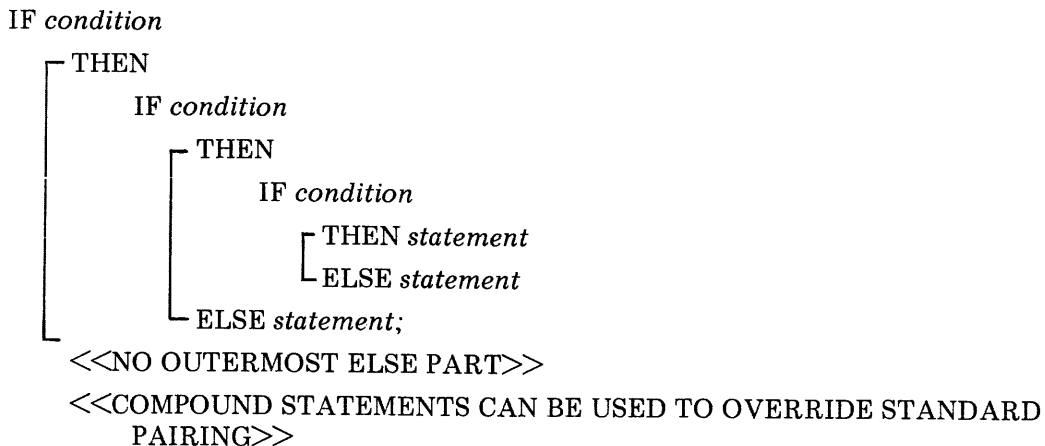
Valid IF Statements

IF A > B THEN C := A;

IF A > B THEN C := B ELSE C := A;

IF B LOR C LAND NOT D THEN GO TO START
    ELSE GO TO FINISH;

IF OVERFLOW AND N = 0 THEN
    BEGIN
        NO := 255;
        GO TO RESTART;
    END
ELSE
    BEGIN
        N := N – 1;
        GO TO START;
    END;

Invalid IF Statements

IF A > B THEN A := B; ELSE B := A; <<invalid ; >>

IF A > B THEN A := B ELSE B := A ELSE XY := A + B; <<too many ELSES>>

IF CARRY LAND A = 0 THEN N := N + 1; <<incorrect use of LAND>>

IF EXPRESSIONS IN ASSIGNMENT STATEMENTS

A powerful use of logical expressions and branch words is in the Expression IF construct. This construct uses a condition (the same rules apply as in IF statements) to choose between two alternative expressions:

> variable := IF *condition* THEN *expression* ELSE *expression*

If the condition results in a TRUE value, the expression after THEN is assigned to the variable; if the condition results in a FALSE value, the expression after ELSE is assigned to the variable. Both the THEN part and ELSE part are required. The expressions can be any valid SPL/3000 expressions, including arithmetic expressions, logical expressions and IF expressions. Both expressions must result in data constructs of the same word size (byte assumed to be word). IF expressions can also be used in the middle of other expressions:

> A := B * C + (IF A = 0 THEN N ELSE A);

Valid IF Expressions

N := IF A > B THEN A ELSE B;

M := IF X = Z + 26 THEN X * Z ELSE X/Z;

MAN := A * B + (IF 15 <= N <= 25 THEN N ELSE 0);

Invalid IF Expressions

N := IF A < B THEN A; <<MISSING ELSE>>
N := IF A < B THEN IF A < C THEN A ELSE B ELSE C
        ELSE D; <<TOO MANY ELSES>>

EXAMPLE 3-1. DATE VERIFICATION

This program illustrates the use of the IF and GOTO statements to verify that a given date
(month, day, year) is valid. The date is entered from the keyboard in the form *mm, dd, yy*
(where *mm* = month, *dd* = day, *yy* = year of century). *mm* is checked for the range 1 <= *mm*
<= 12; *dd* is checked for range according to month, including a check for February and Leap
Year.

Input/Output

```
ENTER A DATE MM,DD,YY
07,15,71
GOOD DATE
ENTER A DATE MM,DD,YY
00,15,42
WRONG INPUT
ENTER A DATE MM,DD,YY
02,29,84
GOOD DATE
ENTER A DATE MM,DD,YY
09,30,85
GOOD DATE
ENTER A DATE MM,DD,YY
04,31,95
WRONG INPUT
```

Listing

```
BEGIN  <<EXAMPLE 3-1. DATE VERIFICATION>>
COMMENT:
        THIS PROGRAM CHECKS A DATE FOR VALIDITY. (THE DATA INPUT
        FORMAT IS MM DD YY . )
        EXAMPLES: FEBRUARY 29 IS A VALID DATE ONLY FOR LEAP YEARS, AND
        SEPTEMBER 31 IS NEVER A VALID DATE.
        NOTE: "INPUT" AND "OUTPUT" ARE DUMMY PROCEDURES WHICH SIMULATE
        INPUT, OUTPUT, AND CONVERSION - THEY ARE NOT PART OF SPL/3000;
BYTE ARRAY GOODMSG(0:8):="GOOD DATE";
BYTE ARRAY ERRORMSG(0:10):="WRONG INPUT";
BYTE ARRAY HEAD(0:20):="ENTER A DATE MM,DD,YY";
INTEGER MONTH,
        DAY,
        YEAR,
        LIMIT;
```

```
LABEL ERROR,DAYCHECK;
<<END OF DECLARATIONS>>
        OUTPUT(HEAD);  <<PRINT HEADING>>
        INPUT(MONTH,DAY,YEAR);  <<READ DATE>>
        IF NOT(1<=MONTH<=12)OR(DAY<=0)OR(YEAR<0) THEN GOTO ERROR;
        IF MONTH=2 THEN  <<FEB IS SPECIAL CASE>>
                        BEGIN
                          LIMIT:=1F YEAR MOD 4=0 THEN 29  <<LEAP YEAR>>
                                                  ELSE 28;
                          GOTO DAYCHECK;
                        END;
        LIMIT:=IF(MONTH=9)OR(MONTH=4)OR(MONTH=6)OR(MONTH=11)THEN 30
                                                   ELSE 31;

DAYCHECK:
        IF 1<=DAY<=LIMIT THEN
                        OUTPUT(GOODMSG)  <<VALID DATE>>
                        ELSE
ERROR:                  OUTPUT(ERRORMSG);  <<PRINT ERROR MESSAGE>>
END  <<DATE VERIFICATION>>.
```

EXERCISES FOR SECTION III

1. Identify those statements containing invalid label positions.

   a) SECTION1 : Y := A * B + X; SECTION2 : C := A ^ 2 + B ^ 2;

   b) Z := (X + Y) + SECTION3: (X − Y);

   c) IF FLAG THEN DOIT : Z := −Z;

   d) BEGIN X := 0; Y := 0; START : IF TOTAL > 10.E17 THEN GO QUIT; END;

   e) BEGIN

           TEMP := 1./A + B;

           IF A < 0 THEN GO AGAIN;

       END; STOP:

2. Rewrite the collection of statements below into an equivalent program section that eliminates as many labels as possible.

LOOP:	IF TESTWORD > 0 THEN GOTO COMPUTE1;
	IF TESTWORD = 0 THEN GOTO COMPUTE2;
	IF TESTWORD < 0 THEN GOTO COMPUTE3;
COMPUTE1:	Y := A + B + C;
	GO TO ENDLOOP;
COMPUTE2:	Y := 1./A + B + C;
	GO TO ENDLOOP;
COMPUTE3:	Y := A * B * C;
ENDLOOP:	COUNT := COUNT + 1;
	IF COUNT <= 10 THEN GOTO LOOP;

3. Single word values (16 bit) in the HP 3000 can be type INTEGER or LOGICAL. In the following exercise specify the INTEGER interpretation, LOGICAL interpretation and TRUE/FALSE interpretation by completing the table provided.

OCTAL NUMBER	AS DECIMAL INTEGER	AS DECIMAL LOGICAL	TRUE/FALSE
177777			
000001			
000377			
000000			
177776			
100000			
000003			
000004			
000005			
000006			

4. Mark the following logical expressions to show the order of evaluation. Assume that all variables are type Logical.

Sample: A LAND B + NOT D

   a) A LOR B LAND C LOR D

   b) NUMBER <> 0 LAND NUMBER * SCALE <= MAX

   c) X + Y / (Y – 5) * NUMBER LAND %77

   d) A * B <> NOT D/C

5. Assume these declarations:

BEGIN
    REAL Z;
    LOGICAL A, B, C, RESULT;
    INTEGER D, E, F;

Identify the invalid logical assignment statements among those below:

   a) A := B OR C;

   b) A := B LOR C;

   c) A := B AND C;

   d) A := B LAND C;

   e) RESULT := FIXR(Z)//B;

   f) A := B + LOGICAL (D MOD 8 + NOT E * 2 / F * 3);

   g) RESULT := A > B LAND B # C;

   h) A := D + E;

   i) RESULT := C LOR A := B;

   j) RESULT := A + B < D * E;

   k) RESULT := NOT A.(6:3) LOR (A := B LAND %177);

6. Assume these declarations:

    LOGICAL A, B, C;
    INTEGER D, E, F;

Examine the logical expressions given below. Rewrite those which are invalid, using type transfer functions where required, to produce syntactically correct logical expressions.

   a) A = B LOR D <> %52

   b) D = E LAND F < D

   c) C MOD D + F

   d) C LOR D <> F/E

3-15

In the following exercises you must construct SPL/3000 statements to perform a given task. In order to make the exercises fun as well as meaningful each test will involve selecting a "secret agent" based on specific criteria. The exercises are arranged to give you practice in constructing IF statements of varying form.

The table below indicates the variable names, data types and codes that you will need to interpret the selection criteria of the individual exercises.

Secret Agent Characteristic Table

CHARACTERISTIC	VARIABLE	TYPE	VALUES-(RANGE) or [CODES]	REMARKS
SERIAL NUMBER	ID	INTEGER	0–100	
AGE	AGE	INTEGER	21–150	YEARS
HEIGHT	HT	INTEGER	48–84	INCHES
WEIGHT	WT	INTEGER	80–250	LBS
COLOR EYES	EYES	INTEGER	1–BROWN 2–BLUE 3–GREEN 4–GRAY 5–OTHER	
SEX	SEX	LOGICAL	TRUE = MALE FALSE = FEMALE	
OCCUPATION COVER	JOB	INTEGER	0–STUDENT 1–ATHLETE 2–ENGINEER 3–TEACHER 4–BUSINESSMAN 5–SPY (DOUBLE AGENT) 6–PLUMBER 7–ARTIST 8–SCIENTIST 9–SERVANT	
LANGUAGE	LANG	INTEGER	0–RUSSIAN 1–CHINESE 2–AMERICAN SLANG 3–SPANISH 4–SWAHILI 5–ENGLISH 6–POLISH 7–LATVIAN 8–ARABIC 9–GERMAN	

Example: Test only the variables AGE and EYES. If the value of age is between 21 and 25 and the code for blue eyes is present then transfer to a label called FOUNDONE. If either test condition fails transfer to a label called NEXT.

Solution: IF (21 <= AGE <= 25) AND EYES = 2 THEN GOTO FOUNDONE
ELSE GOTO NEXT;

7. Construct SPL/3000 statements to perform tests based on the four sets of criteria below. Use the values specified for each variable to construct each test. In cases where no entry is shown the variable need not be tested.

	ID	AGE	HEIGHT	WEIGHT	EYES	SEX	COVER	LANGUAGE
a)	—	30 or less	5' 6"	—	—	M	Student	Am slang
b)	30–70	—	5' 2"	90–110	Blue	F	Servant/ artist/ teacher	Latvian
c)	above 50	21–24/ 37–50	6' 7"	above 190	Brn or M but not *both*		Athlete/ scientist	Swahili/ German
d)	—	45–50	—	—	Brn and M OR Blu and F		Businessman and Russian OR SPY and Chinese	

8. Assume that as a result of the last exercise no agent was found who possessed all of the capabilities required. Write a sequence of statements that will select the most qualified candidate. Assume that each successful test has a point value of one. Unsuccessful tests have a point value of zero. Accumulate the total point value for each agent being tested for each of the selection criteria in Exercise #7.

9. Assume that an agent is required who has the largest number of necessary conditions listed below. The conditions are in priority order, such that, if an agent fails to meet any condition he is not even checked for the next. Each criteria from lowest priority to highest has an increasing point value. Each test is worth a point; perform the tests and return the total (failure to first test returns 0; passing 5 tests returns 5). (Hint—use nested IF assignment statement.)

Most important ←———————— Selection Criteria ————————— Least important

	LANGUAGE	SEX	HEIGHT	WEIGHT	AGE	EYES	COVER
a)	RUSSIAN	MALE	(5' 9"–6' 3")	160–210	21–35	BLUE	SPY

	SEX	AGE	HEIGHT	WEIGHT	EYES	COVER	LANGUAGE
b)	FEMALE	21–25	4' 10"–5' 5"	95–120	GRAY	TEACHER	SWAHILI

10. Examine the two IF statement constructs shown below and determine the appropriate values of X required to complete the truth tables provided.

Assume: INTEGER X; LOGICAL L1, L2;

Case I

```
X := 0
IF L1 THEN BEGIN
    IF L2
        THEN X := 1
    END
ELSE X := 2;
```

Case II

```
X := 0;
IF L1 THEN
    IF L2
        THEN X := 1
ELSE X := 2;
```

Table I

L_1	L_2	X
T	T	
T	F	
F	T	
F	F	

Table II

L_1	L_2	X
T	T	
T	F	
F	T	
F	F	

SECTION IV
Looping Constructs

This section covers some topics related to looping operations:

- Compile-time constants (EQUATE)
- Abbreviations (DEFINE)
- Indexing (index register)
- Sets of data which can be indexed (arrays)
- Looping control (FOR, DO-UNTIL, and WHILE-DO statements)
- Computed transfers (SWITCH and CASE)

EQUATE DECLARATION

An EQUATE declaration assigns an integer value to an identifier. EQUATE is only a compile-time convenience; it does not allocate any storage, but merely provides a form of abbreviation for constants. When an equated identifier is used, the appropriate constant is substituted in its place. When EQUATEs are used instead of actual constants, programs can be changed simply; instead of replacing every occurrence of a constant, only the EQUATE declaration need be changed. The format of an EQUATE declaration is

    EQUATE *list of equates* ;

Each item in the equate list is an identifier followed by "=" and an integer constant expression consisting of operators (+, −, *, /), parentheses, integers, and previously defined equates. Normal rules of precedence apply to the operators. The value defined by the expression is assigned to the identifier.

Valid EQUATE Declarations
EQUATE X = 1, Y = X + 1, Z = Y + 1;
EQUATE T = Z / 2 − 3 + X, N = (T + 3) * Z;
A := (X + Y) * (T − N); <<USE OF EQUATES IN ASSIGNMENT>>

Invalid EQUATE Declarations
EQUATE A = B = C = 4; <<multiple equate assignments not permitted>>

In the preceding valid EQUATEs X, Y, Z, T, and N equal 1, 2, 3, − 1, and 6 respectively.

EQUATE is not used for address equivalencing in SPL/3000 as it is in some languages. The conventions for referencing identifiers to specific addresses are covered in Section VII.

DEFINE DECLARATION

A DEFINE declaration assigns a block of text to an identifier. Whenever the identifier is used in the program thereafter, the assigned text replaces the identifier (except in comments). The format of a DEFINE declaration is

DEFINE *list of definitions* ;

Each definition is constructed as follows:

identifier = text #

A DEFINE is invoked by inserting the identifier into the program where its corresponding text will make sense. DEFINEs can be nested; that is, defined identifiers can be used in the text of subsequent defines. Also, DEFINEs can be used anywhere, even in other declarations. For example,

DEFINE I = A, B, C, D, E #; INTEGER I;	≡	INTEGER A, B, C, D, E ;

The two declarations on the left have the same effect as the declaration on the right. A good example of a DEFINE occurs in Example 2-2 where it is used to abbreviate a long, repeated procedure declaration.

INDEX REGISTER

The index register is a 16-bit program-accessed register which can be used as a subscript for arrays and pointers, as a simple arithmetic register, and as a FOR loop variable. In addition, the index register is used by several specialized instructions (IXBZ, DXBZ, CPRB, TBX, MTBX, SCAN, TNSL, PLDA, PSTA, LLSH). The index register can be used as a variable in SPL/3000 programs by declaring a variable of type LOGICAL, INTEGER, or BYTE and equivalencing it to the hardware index register. The equivalencing is done by following the identifier in the declaration by the sequence "=X." For example,

INTEGER INDX = X;
LOGICAL X = X;

X is not a reserved word which always means the index register; it can be used as an identifier to mean anything the programmer desires. X has a reserved meaning only if it occurs after an equals sign (=) in a declaration such as the above. Thus, the declaration below has nothing to do with the index register; it merely declares a simple variable that happens to be named X:

INTEGER X;

4-2

All variables which are referenced to the index register will have the same value in a program. Since the index register is used implicitly in all array indexing and many other constructs of SPL/3000, the contents of variables equivalenced to the index register can change dynamically. The current value of the index register is the last value established there. The compiler does not save the value of variables referenced to the index register (except that the hardware saves the index register contents upon procedure entry and restores it upon procedure exit). The programmer must maintain the integrity of variables equivalenced to the index register if he desires consistent results.

ARRAYS

An array is a block of contiguous storage which is treated as an ordered sequence of "variables" having the same data type. These "variables" are accessed using a single identifier to denote the array and a subscript number to denote the particular "variable" (element) within the array. Thus array elements are sometimes called subscripted variables.

SPL/3000 provides one-dimensional arrays (only one subscript is allowed,) in all data types (integer, logical, real, byte, long, double). Significant facts about SPL/3000 arrays are that subscripting automatically uses the index register to indicate the element number (since this is what the hardware provides); that there is no compile-time or run-time bounds checking (it is possible to access an array element beyond the declared bounds of the array); that arrays can be given initial values during declaration (but are not automatically initialized to any default value such as 0); and that arrays can be located in any region of the user's domain which can be addressed relative to the DB, Q, S, or P registers.

Several advanced array features are covered in Section VII: explicit address referencing, undefined bounds, and explicit storage allocation.

Array Declarations

Arrays and their attributes, must be declared. The declaration determines the array's identifier, data type, bounds (number of elements), and initial values. The format for global arrays is:

> *type* ARRAY *declaration list* ;

> > *type* is INTEGER, LOGICAL, or REAL (also BYTE, DOUBLE, and LONG — to be covered later); default type is LOGICAL;

> > *declaration list* is a list of elements of the form:

> > > *identifier (lower bound : upper bound)*

> > > *lower bound* and *upper bound* are integer constant expressions (-32768 <= integer <= $+32767$); the upper bound must not be less than the lower bound.

For example,

Valid ARRAY Declarations

INTEGER ARRAY SORT (0:9), SILT (-5:22);

REAL ARRAY RESULTS (10:20);

LOGICAL ARRAY SINCERE (-7:-2);

ARRAY TRUTH'TABLE (1000:1100);

Invalid ARRAY Declarations

ARRAY SOME (5:-5); <<UPPER BOUND LESS THAN LOWER>>

Array Storage Allocation

For each array that is declared, a location in primary DB (DB + 0 to DB + 255) is allocated for a data label (address) which points to the zero element of the array (the element with subscript 0). Then enough space is allocated in the secondary DB area (DB + 256 and above) to accommodate the elements defined by the array bounds. (Integer and logical arrays use the same number of words as they have elements; real arrays use two words for each element.) The data label is set to point at the zero element of the space allocated. If 0 is not within the declared bounds of the array, the data label actually points to a location that is not part of the array. The example below shows the space allocated for the declaration of four arrays in an SPL/3000 main program.

EXAMPLE OF SPACE ALLOCATION

Declarations

BEGIN

    INTEGER ARRAY ADATA (0:15);

    REAL ARRAY TPZD (-5:10);

    LOGICAL ARRAY TABLE (10:20);

    BYTE ARRAY DBUFR (0:80);

    .
    .
    .

END.

NOTE: Byte addresses are inflated by a factor of two in order to allow true element indexing. See Section V for details.

Allocations

Primary DB:
- ADATA (0) address
- TPZD (0) address
- TABLE (0) address
- DBUFR (0) address

Secondary DB:
- ADATA: 0 to 15
- TPZD: -5, 0, 10
- TABLE: 10 to 20
- DBUFR: 0 to 80
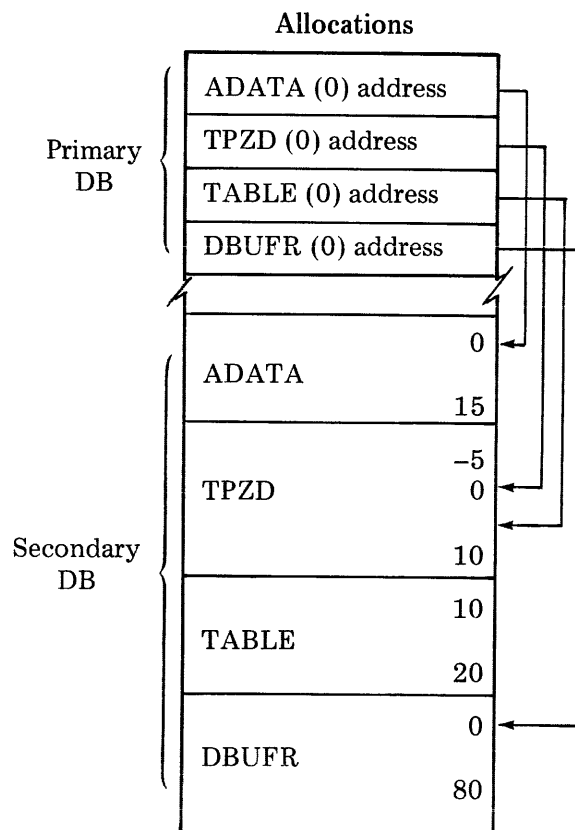
In the diagrams of this book secondary DB is shown as starting at DB + 400_8 (256_{10}) for purposes of illustration. In actual practice the base of DB is adjusted by the compiler and the loader so that no space is wasted in DB.

Array Initialization

Array declarations can be used to initialize array elements with constants. Only the last array of any declaration list can be initialized. The format used to initialize an array is as follows:

    identifier (bound : bound) := list of initial values

    list of initial values   consists of a series of constants and repeat groups (lists of constants which have been enclosed in parentheses and preceded by a decimal repetition factor) separated by commas.

Repetition factors allow large arrays to be initialized without writing out all the constants. For example, the two declarations below are equivalent. The repetition factor in the first causes the constants in parentheses to be used three times to fill the array.

INTEGER ARRAY A(0:8) := 3(1,5,10);	≡	INTEGER ARRAY A(0:8) := 1,5,10,1,5,10,1,5,10;

Array elements are initialized from smallest to largest subscript and initialization will be partial (low subscripts only) if there are not enough constants to fill the entire array. Repetition factors cannot be nested. For example,

    REAL ARRAY DTALOG (–1:3) := 1.,3.,4.7;

This means that DTALOG (–1) = 1.0, DTALOG (0) = 3.0, DTALOG (1) := 4.7, DTALOG (2) is undefined, and DTALOG (3) is undefined.

Arrays of all data types can be initialized:

Valid Array Initialization

ARRAY A(1:10) := 1,2,3,4,5,6,7,8,9,10;

ARRAY N(5:10),MNM(–5:25) := "ABCDE", TRUE,5,93;

INTEGER ARRAY BOYS(100:150) := 10(1,2,3,4,5);

INTEGER ARRAY GIRLS(1:150) := 5("AB"," ", 25), 100(%77);

Invalid Array Initialization

INTEGER ARRAY MEN(0:5) := 6(36), WOMEN(0:5) := 6(32);
    <<only last array can be initialized>>

When strings are used to initialize and array, the bytes of the string are taken from left to right until exhausted. If an odd number of bytes is used to initialize a word array then the next byte is initialized to a blank character to fill out the word.

Accessing Array Elements

A specific array element is accessed by specifying the array identifier followed by a subscript (index) in parentheses. The index can be any expression or assignment statement which results in an integer, logical, or byte value. The zero element of an array can be specified most efficiently by using the identifier without an index.

> *identifier (index)*
>
> *identifier* <<ZERO ELEMENT>>

The index value is loaded into the index register and used to reference the array element. The hardware provides true element indexing for byte, word, and double word elements. That is, the value in the index register specifies the element offset (plus or minus) from the zero element of the array, not the word offset. The memory reference instructions take into account the size of the element (½, 1, or 2 words) when performing indexing. The hardware does not, however, perform any bounds checking to insure that the indexed element lies within the declared bounds of the array. For example,

```
A := NEW(N);
B := NEW(I := I + 1);
C := NEW(N) + NEW(N + 1);
D := NEW(20 + NEW(10));
E := NEW; <<ZERO ELEMENT>>
F := NEW(0); <<ZERO ELEMENT>>
IF STUFF(X + Z + (N * 2) / (N * 3)) = 0 THEN A := X;
```

SAMPLE ARRAY

```
BEGIN
     INTEGER ODDSUM := 0;
     INTEGER ARRAY DATA (1:5) := 5,4,3,2,1;
     <<START MAIN CODE>>
     ODDSUM := DATA(1) + DATA(3) + DATA(5);
END.
```

FOR STATEMENT

A frequent operation in computing is the repetition of some calculation, incrementing a key variable each time the calculation is performed. In SPL/3000 there are several constructs to provide repetition or looping. The FOR statement is one of these; it is used to repeatedly execute one or more statements. Each time the sequence is executed, an integer count variable is incremented or decremented until some predefined limit is exceeded. The FOR statement in SPL/3000 is very machine-dependent because it makes use of special-purpose looping instructions (MTBA, TBA, MTBX, TBX). The other two methods of looping in SPL/3000 (DO-UNTIL and WHILE-DO) are less machine-dependent and more general, but do not provide automatic incrementing or decrementing of a count variable.

Basic Form

Although there are four different formats of the FOR statement, the following one is basic:

FOR *integer var* := *initial expr* STEP *step expr* UNTIL *limit expr* DO *statement*;

integer var	is any variable of type integer (including variables equivalenced to the index register);
initial expr, *step expr*, *limit expr*	are arithmetic expressions resulting in one-word quantities.
statement	is any statement including a compound statement.

When the FOR statement is entered the first time, the value of *initial expr*, *step expr*, and *limit expr* are calculated. *Initial expr* is stored into *integer var* and the other two values are kept on the top of the stack. Then the integer variable is tested to check whether it exceeds the limit value. If not, the statement after DO is executed. Each time the statement is executed, the step value is added to the integer variable, which is then tested against the limit. When the integer variable exceeds the limit, execution falls through to the statement following the FOR statement. A negative step value branches when the variable is less than the limit. For example,

```
BEGIN
        INTEGER I, MAX, RANGE, SUM := 0,
        A := 2, B := 4, C := 5, FOFI;
        <<START FOR STATEMENT>>
        FOR I := MAX STEP-RANGE/4 UNTIL MAX-RANGE
                DO BEGIN
                        FOFI := A*I^2+B*I+C;
                        SUM := SUM + FOFI;
                END;
END.

FOR I := 3 STEP 1 UNTIL LIM DO A(I) := I*2;
FOR J := X + Y STEP A UNTIL B DO NUM(J) := A*B;
FOR K := N STEP 1 UNTIL STOP DO
        BEGIN
                NUM (K) := NUM (K) + 1;
                IF NUM(K) < APPOINT THEN
                        NUM(K) := NUM(K)/2;
                A := NUM(K) + APPOINT
        END;
```

Alternate Forms

If the "STEP *step expr*" part of the FOR statement is left out, a step value of one is assumed:

FOR *integer var* := *initial expr* UNTIL *limit expr* DO *statement*;

For example,

FOR I := 3 UNTIL LIM DO A(I) := I*2;

If the integer variable is preceded by an asterisk (*), the loop statement is executed once before testing the integer variable. This guarantees that the loop statement is executed at least once, even if it would fail on the initial test:

FOR * *integer var* := *initial expr* STEP *step expr* UNTIL *limit expr* DO *statement*;

FOR * *integer var* := *initial expr* UNTIL *limit expr* DO *statement*;

For example,

FOR * I := 1 STEP 1 UNTIL LIM DO SUM := SUM + NARN(I);

FOR * SAMPLE := AVERAGE UNTIL LAST

    DO BEGIN

        RESULT := RESULT + DATA(SAMPLE) * SCALE / TOTAL;

        SCALE := SCALE / 2;

    END;

FOR statements can be nested as deeply as the programmer desires. That is, the loop statement of a FOR statement can be (or contain) another FOR statement. Example 4-1 has a good example of two-level nesting of FOR statements.

Cautions in the Use of FOR Statements

FOR loops in SPL/3000 are very machine-dependent and use specialized machine instructions (TBA, TBX, MTBA, MTBX). These instructions require that certain control values (final value, step value, and address of the loop variable) remain in the stack during execution of the loop statement. After the range of the loop has been completely executed, the modify-test-and-branch instruction (MTBA or MTBX) expects the control values to be on the top of the stack. If they have been displaced from their expected position, the behavior of the loop control operations is unpredictable.

Therefore, it would be prudent for the beginning SPL/3000 programmer to observe the following rules.

- Do not use the stack explicitly within the loop statement without restoring any changes made because this makes it impossible for the compiler to keep track of the control values in the stack. (Do not refer to TOS, S relative variables, or stacked parameters; these are further described in Section VII.)

- Enter FOR statements only from the beginning. Never branch into the loop statement.

- Exit FOR statements only at the end, except for PCALs.
- Do not modify the index register in any way (without also restoring it) within the loop statement if a variable equivalenced to the index register is being used as the loop control variable. (The compare range construct is a little-known implicit use of the index register: A <= B <= C. Use of this construct or subscripted variables within the loop statement will cause unpredictable results if the loop variable is also the index register.)

The WHILE-DO statement and DO-UNTIL statement provide alternatives to the FOR statement that are safer for the beginning programmer to use.

EXAMPLE 4-1. INTEGER SORT

This program uses FOR loops, EQUATES, and arrays to perform a simple sort on SIZE (9) integer values (6 characters, signed or unsigned decimal, or octal). The program converts each number entered to binary, and sorts the values using a straight-forward, ripple sort method. After the values are sorted, they are output (in readable form).

Input/Output

```
ENTER 10 INTEGER VALUES
123
+987
-32768
0
+32767
90
-87
-2
1
17
OUTPUT DATA
-32768
-87
-2
0
1
17
90
123
987
32767
```

```
ENTER UP TO 10 NUMBERS, OR /E
-1
642
%100
9999
0
123
45
/E

OUTPUT DATA
-1
0
45
64
123
642
9999
```

Listing

```
BEGIN   <<EXAMPLE 4-1. INTEGER SORT>>
COMMENT:
        THIS PROGRAM WILL ORDER AN INTEGER ARRAY WITH SUBSCRIPTS
        RUNNING FROM ZERO TO "SIZE". THE ARRAY ELEMENTS WILL BE
        ORDERED ALGEBRAICALLY - SMALLEST VALUE TO LARGEST VALUE.
        NOTE: "INPUT" AND "OUTPUT" ARE DUMMY PROCEDURES WHICH SIMULATE
        INPUT, OUTPUT, AND CONVERSION - THEY ARE NOT PART OF SPL/3000;
EQUATE SIZE=9;
BYTE ARRAY ASCIIBUF(0:23):="ENTER 10 INTEGER VALUES ";
BYTE ARRAY OUT(0:11):="OUTPUT DATA ";
INTEGER ARRAY SORT(0:SIZE);
INTEGER SAVE,
        J,K,
        TEMP;
<<END OF DECLARATIONS>>
        OUTPUT(ASCIIBUF);  <<OUTPUT INSTRUCTIONS>>
        FOR J:=0 UNTIL SIZE DO  <<INPUT DATA LOOP>>
        INPUT(SORT(J));  <<INPUT A VALUE>>
COMMENT:
        ORDER THE ARRAY
;
        FOR K:=0 STEP 1 UNTIL SIZE-1 DO  <<OUTER LOOP>>
          FOR J:=K+1 STEP 1 UNTIL SIZE DO  <<INNER LOOP>>
            IF SORT(K) > SORT(J) THEN  <<NOT IN ORDER>>
              BEGIN
                SAVE:=SORT(K);  <<EXCHANGE VALUES>>
                SORT(K):=SORT(J);
                SORT(J):=SAVE;
              END;
        OUTPUT(OUT);  <<PRINT HEADING>>
        FOR K:=0 UNTIL SIZE DO  <<OUTPUT LOOP>>
        OUTPUT(SORT(K));  <<PRINT A VALUE>>
END  <<INTEGER SORT>>.
```

DO UNTIL STATEMENT

The DO UNTIL statement allows the programmer to execute repeatedly a specified statement (including compound statements) until a specified condition is true. The condition is evaluated and checked after each execution of the statement. The format is

DO *statement* UNTIL *condition* ;

statement   is any SPL/3000 statement including a compound statement or another DO UNTIL statement.

condition   is a sequence of logical expressions and branch words connected exactly as in IF conditions. (See Section III.)

Normally, the loop statement is a compound statement containing at least one statement that modifies one of the variables in the condition. Since the condition is tested after execution, the statement always is executed at least once. When the condition is false, execution transfers to the first statement following the DO UNTIL statement. The following diagram shows this sequence:

```
DO
 └▶BEGIN ◄─────────────┐
        ↓ statement     │
              .         │
              .         │
              .         │
          statement     │
      END ┐             │
 UNTIL    ↓             │
               ◇   false│
       ┌──◇condition◇───┘
  ↓ true
```

For example,

        LOGICAL ARRAY A (0:19);

        BYTE ARRAY STRING (0:71);

        INTEGER SUB := –1, I := 0, J;

            DO SUB := SUB + 1 UNTIL STRING(SUB) <> "0";

            DO BEGIN

                STRING(I) := "X";

                I := I+1;

                END

            UNTIL I = 72;

            I := –1;

            DO J := A(I := I+1) UNTIL I > 18;

WHILE DO STATEMENT

The WHILE DO statement allows the programmer to repeatedly execute a statement as long as a specified condition is true. The condition is always evaluated and tested before executing the loop statement. The format is

WHILE *condition* DO *statement* ;

statement   is any SPL/3000 statement including a compound statement or another WHILE DO,

condition   is any sequence of logical expressions and branch words connected as in IF conditions, except that the "true" sense of IABZ, DABZ, IXBZ, and DXBZ is reversed for this statement only.

The logical value (true or false) of the condition is determined before each execution of the loop statement. The statement is executed as long as the result is true. Since the condition is checked first, the statement is not executed at all if the condition is initially false. When the condition is false, execution transfers to the next statement following the WHILE DO statement. The following diagram shows this sequence:

```
          WHILE        /\        false
                      /    \
            ┌──────>< condition >──────>
            │         \    /
            │           \/
            │           │ true
            │           ▼
            │         BEGIN
            │            statement
            │               .
            │               .
            │               .
            │            statement
            └────────  END;           ▼
```
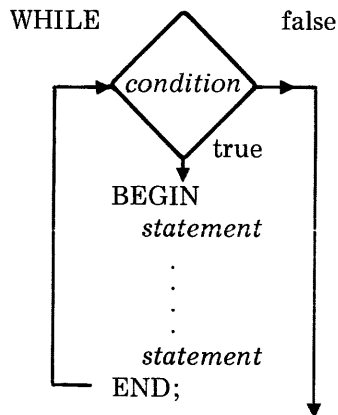
For example,

```
INTEGER I := -10, MAX, FCTR := 1;
INTEGER ARRAY SQR(1:49), DATA (-10:50);
    WHILE DATA (I) < MAX AND I < 50 DO I := I+1;
    WHILE FCTR <= 49 DO
        BEGIN
            SQR(FCTR) := FCTR^2;
            FCTR := FCTR+1;
        END;
```

EXAMPLE 4-2. TABLE SEARCH

This example illustrates the use of the WHILE DO statement in a table search application.
Dummy part numbers and stock quantities are kept as constants in an integer array for purposes
of simplicity. The program requests a part number from the terminal and inputs the number.
If the binary number does not compare to any part numbers in the inventory, the message NOT
IN INVENTORY is printed. If the number corresponds to an inventory item, the quantity
associated with the number is printed and the program loops back to the beginning. The user
inputs /E to terminate the program.

Input/Output

```
ENTER A PART NUMBER (20100<=PN<=20109)
20109
44  ON HAND
ENTER A PART NUMBER (20100<=PN<=20109)
20101
1000  ON HAND
ENTER A PART NUMBER (20100<=PN<=20109)
20105
144  ON HAND
ENTER A PART NUMBER (20100<=PN<=20109)
20199
 NOT IN INVENTORY
ENTER A PART NUMBER (20100<=PN<=20109)
20100
12  ON HAND
```


Listing

```
BEGIN  <<EXAMPLE 4-2. TABLE SEARCH>>
COMMENT:
        THE PROGRAM INPUTS A PART NUMBER (20100<=PN<=20109), SEARCHES
        A TABLE FOR THE PART, AND PRINTS THE QUANTITY ON HAND. AN
        ERROR MESSAGE IS WRITTEN IF THE PART NUMBER IS NOT IN THE
        INVENTORY TABLE.
        NOTE: "INPUT" AND "OUTPUT" ARE DUMMY PROCEDURES WHICH SIMULATE
        INPUT, OUTPUT, AND CONVERSION - THEY ARE NOT PART OF SPL/3000;
BYTE ARRAY NO'PART (0:16):=" NOT IN INVENTORY";
BYTE ARRAY OUTBUF (0:8):="  ON HAND";
BYTE ARRAY HEAD(0:37):="ENTER A PART NUMBER (20100<=PN<=20109)";
INTEGER ARRAY INVENTORY (0:20):= <<PART NUMBER,QUANTITY>>
                          20100,12,
                          20109,44,
                          20103,79,
                          20105,144,
                          20106,0,
                          20101,1000,
                          20104,3,
                          20107,0,
                          20102,20,
                          20108,10,
                             -1;
```

```
INTEGER PART,I;
<<END OF DECLARATIONS>>
        OUTPUT(HEAD);   <<PRINT HEADING>>
        INPUT(PART);   <<READ A PART NUMBER>>
        I:=0;   <<INITIALIZE SUBSCRIPT>>
        WHILE(PART<>INVENTORY(I))AND(INVENTORY(I)<>-1) DO I:=I+2;
        IF INVENTORY(I)=-1 THEN <<PART NOT IN LIST>>
                                OUTPUT(NO'PART)
                            ELSE
                                OUTPUT(INVENTORY(I+1),OUTBUF);
END  <<TABLE SEARCH>>.
```

SWITCH STATEMENT

The purpose of a SWITCH is to transfer execution to one of several labeled statements within a program. A SWITCH is first declared and then invoked (through an indexed GOTO statement). A SWITCH invocation differs from a simple GOTO in that there are several destination statements to choose from, not just a single statement. The format of a SWITCH declaration is

SWITCH *identifier* := *list of labels* ;

The SWITCH declaration defines an **identifier** to represent an ordered set of labels. Each label in the list (from left to right) is assigned a number from 0 to N – 1 (where N is the number of labels) which indicates the position of the label in the list. Note that the SWITCH declaration implicitly defines the list of label identifiers as labels; they need *not* be further defined in a LABEL declaration. For example,

SWITCH SW := L0,L1,L2,L3;

The labels defined in the SWITCH declaration identify various statements in the body of the program (see Section III). A switch of program control to one of these labeled statements is accomplished by using a GOTO statement with the switch *identifier* and an *index*. The format is

GOTO *switch (index);*

GOTO * *switch (index);*

switch	is a previously declared switch identifier.
index	is an arithmetic expression, logical expression, or assignment statement with a one-word result.
*	specifies no bounds checking (this overrides checking done to insure that the index references a valid label in the range of the switch; an invalid index value executes a statement that is not one of the predetermined branches, often with unpredictable results).

NOTE: Any valid form of the GO construction can be used (GO, GOTO, GO TO).

The index is evaluated to an integer value and control is transferred to the switch label specified by that number. If the index falls outside the range of the switch declaration and the * option is not used, control transfers to the next statement following the GOTO statement. When the * option is used, bounds checking code is omitted. This produces a more efficient program, but is only safe when the programmer is certain that the indexes will be in bounds. For example,

```
SWITCH SW := L0, L1, L2, L3;
           .
           .
           .
    GOTO SW(A + B / 2);
           .
           .
           .
    GOTO * SW(A * B);
           .
           .
           .
    L0:  statement; <<INDEX = 0>>
           .
           .
           .
    L1:  statement; <<INDEX = 1>>
           .
           .
           .
    L2:  statement; <<INDEX = 2>>
           .
           .
           .
    L3:  statement; <<INDEX = 3>>
```

CASE STATEMENT

The purpose of a CASE statement is to select *one* of a set of statements for execution by using an index into a list of statements. The first statement has index 0 and the others are ordinal numbers (1, 2, 3, . . .). After the execution of the specified statement, control transfers to the statement following the CASE statement. The format of a CASE statement is

CASE *expr* OF *body*;

CASE * *expr* OF *body*;

expr is an arithmetic expression with a one-word result which specifies the statement number desired.

body is a compound statement (a series of statements enclosed by a BEGIN-END pair).

*   specifies no bounds checking (as in SWITCH, this overrides checking and allows an expression value that can index out of the CASE body.

If the * option is not specified and the expression value does not specify a statement within the CASE body, no statement in the CASE body is executed and control transfers to the statement following the CASE body. For example,

```
INTEGER I,N;
        CASE I OF BEGIN
              N := 3;
              ; <<NULL statement; NO ACTION, BUT HOLDS PLACE>>
              N := 5;
              N := 2;END;
```

EXAMPLE 4-3. INTEGER CALCULATOR

This program uses a CASE statement to simulate a single accumulator calculator capable of performing arithmetic functions with integer values ($-32768 \leqslant I \leqslant +32767$). The program inputs a special character and an integer number. The character determines the function to be performed:

value entry (.)
addition (+)
subtraction (–)
mutliplication (*)
division (/)

The actual binary value of the function character is adjusted downward to be used as an index into the CASE statement. When the operation is complete, the contents of the accumulator are output.

Input/Output

```
INTEGER CALCULATOR
.,453
ANS =  453
*,5
ANS =  2265
-,22
ANS =  2243
/,20
ANS =  112
/,2
ANS =  56
-,42
ANS =  14
#,46
ENTRY  ERROR
.,46
ANS =  46
-,535
ANS =  -489
T
```

Listing

```
BEGIN   <<EXAMPLE 4-3. INTEGER CALCULATOR>>
COMMENT:
          INTEGER CALCULATOR - SINGLE ACCUMULATOR
          INTEGER ARITHMETIC OPERATIONS MAY BE PERFORMED BY ENTERING
          AN OPERATOR CHARACTER FOLLOWED BY AN INTEGER NUMBER. THE
          OPERATOR CHARACTERS ARE:
                              .  LOAD ACCUMULATOR
                              +  ADD
                              -  SUBTRACT
                              *  MULTIPLY
                              /  DIVIDE
                              T  TERMINATE
          NO CHECK IS MADE FOR ARITHMETIC OVERFLOW CONDITIONS.
          NOTE: "INPUT" AND "OUTPUT" ARE DUMMY PROCEDURES WHICH SIMULATE
          INPUT, OUTPUT, AND CONVERSION - THEY ARE NOT PART OF SPL/3000;
BYTE ARRAY MSG(0:17):="INTEGER CALCULATOR";
BYTE ARRAY ERR(0:10):="ENTRY ERROR";
BYTE ARRAY ANSW(0:5):="ANS = ";
BYTE ASC;
INTEGER ACCUM:=0,
        OPERAND:=0,
        INDEX;
LABEL FUNCTION,
      EXIT;
<<END OF DECLARATIONS>>
          OUTPUT(MSG);  <<PRINT HEADING MESSAGE>>
FUNCTION:
          INPUT(ASC,OPERAND);  <<READ OPERATOR AND VALUE>>
          IF ASC="T" THEN GOTO EXIT;  <<TERMINATE>>
          IF %52<=ASC<=%57 THEN  <<VALID OPERATOR>>
                              INDEX:=ASC-%52
                         ELSE  <<INVALID OPERATOR>>
                            BEGIN
                              OUTPUT(ERR);  <<ERROR>>
                              GOTO FUNCTION;  <<RESTART>>
                            END;
          CASE INDEX OF   <<INDEX BY OPERATION>>
            BEGIN
              ACCUM:=ACCUM*OPERAND;   << * FOR MULTIPLY>>
              ACCUM:=ACCUM+OPERAND;   << + FOR ADD>>
              <<NULL STATEMENT>>;     << , NO OPERATION>>
              ACCUM:=ACCUM-OPERAND;   << - FOR SUBTRACT>>
              ACCUM:=OPERAND;         << . FOR LOAD ACCUMULATOR>>
              ACCUM:=ACCUM/OPERAND;   << / FOR DIVIDE>>
            END;
          OUTPUT(ANSW,ACCUM);  <<PRINT LABEL AND ACCUMULATOR>>
          GOTO FUNCTION;  <<RESTART>>
EXIT: END  <<INTEGER CALCULATOR>>.
```

EXERCISES FOR SECTION IV

1. Identify the invalid EQUATE declarations.

   a) EQUATE BROWN = 1, BLUE = 2, GREEN = 3, GRAY = 4, OTHER = 5;

   b) EQUATE BIGNUMBER = 1.73E15;

   c) EQUATE XSQUARE = 144^2;

   d) EQUATE PI = 31417;

   e) EQUATE ARRAYLIMIT = "(0:100)";

   f) EQUATE X = 2, Y = X * 3, Z = X + Y, W = (Z * Z – X) / X + Y;

   g) EQUATE NEGLIMIT = –144, POSLIMIT = \ NEGLIMIT \ ;

   h) EQUATE LETTERS = "AB";

   i) EQUATE Z = X + Y, X = 2, Y = X * 3;

2. Identify the invalid array declarations.

   a) INTEGER ARRAY SAM (0, 10) := 0, 1, 2, 3, 4, 5, 6, 7, 8, 9;

   b) REAL ARRAY ALPHA (–99:0);

   c) ARRAY ATANK (0:8) := " NOW IS THE TIME";

   d) LOGICAL ARRAY (0:4) := "ABCDEFGHIJ";

   e) REAL ARRAY BETA (1:10) := 10(0.0), GAMMA (1:10) := 10(0.0);

   f) LOGICAL ARRAY LETTERS (1:6) = OUTPUT DATA;

   g) INTEGER ARRAY N1 (0:4), N2 (0:4), N3 (0:4), N4 (0:4) := 4 (0.0);

   h) REAL ARRAY RESULT (0:100) := 10(10(0.0));

   i) LOGICAL ARRAY EQUATE (0:6) := 7(FALSE);

   j) INTEGER ARRAY SMALL (0:100) := 100(0, 1, 2, 3, 4, 5, 6, 7, 8, 9);

3. Determine whether each of these statements is true or false.

   a) Arrays must always contain a zero element.

   b) Array bounds must not be negative.

   c) Integer arrays can be initialized with string constants.

   d) Subscripts for arrays must be constants.

   e) HP 3000 hardware permits arrays to be accessed outside their defined bounds.

4. Examine the following assignment statements.

INTEGER ARRAY NUMBER (0:10);
        NUMBER (0) := NUMBER (0) + 1; <<SOLUTION 1>>
        NUMBER := NUMBER + 1; <<SOLUTION 2>>

a) Do both statements achieve the same result?

b) If so, which solution is the most efficient? Why?

5. Write an equivalent declaration for the logical array DATA replacing the equate identifiers with integer constant bounds.

EQUATE MIDPOINT = 52,

    RANGE = 75,

    LOWER = MIDPOINT – RANGE,

    UPPER = MIDPOINT + RANGE;

ARRAY DATA (LOWER : UPPER);

6. Refer to the tables in the back of this book and EQUATE the bit values of the appropriate ASCII characters to the following identifiers.

BLANK

RETURN

LINEFEED

NULL

7. In the program below, change the extract bit from 5 to 7 and the bit field length from 4 to 3 by replacing one line.

EQUATE S = 5, L = 4;

X := ITEM'A.(S:L)+

    ITEM'B.(S:L)+

    ITEM'C.(S:L)+

    ITEM'D.(S:L)+

    ITEM'E.(S:L)+

    ITEM'F.(S:L);

8. Use the DEFINEs given to rewrite the statements shown below in expanded form. Substitute the defined string when the define identifier is invoked.

a) DEFINE RANGE = <= DATA <= #;

    IF 5 RANGE 20 OR X RANGE Y THEN X := X + 1;

b) DEFINE INIT = 1, 3, 5, 7, 9, 11, 13#;

    ARRAY ONE (0:7) := INIT;

    ARRAY TWO (0:7) := INIT;

c) DEFINE AVAIL = (0:2)#,

        STAT   = (2:6)#,

        SEG   = (8:8)#;

    IF LINK. AVAIL = 1 AND LINK. STAT <4 THEN LINK. SEG := 0;

9. The declarations and statements shown below were written by a programmer who liked to abbreviate certain reserved words. In specific cases he also thought that THEN really meant ASSIGN and ELSE was more appropriately called OTHERWISE. Use the DEFINE statement to transform the programmer's abbreviations into the correct reserved words.

        EQ LO = 1, HI = 50;
        INT ARY BUF (LO:HI);
        DEF SUB = (15)#;
        LOG RESULT;
        BEGIN
            RESULT := IF BUF SUB ASSIGN 10
                    OTHERWISE 20;

10. For this array declaration, compute the DB address of the zero element. (The value the SPL/3000 compiler will place into the array data label). Assume array space allocation starts at DB + 256 (decimal).

        ARRAY TOTAL (100:199);

11. Use a FOR statement to find the smallest, largest and average value of an integer array. Assume all values are positive.

        EQUATE N = 99;
        INTEGER ARRAY AGE (0:N);

12. a) Select the correct value of the index variable after completion of the FOR statement.

    FOR INDEX := 1 UNTIL N DO <<statement>>;

    Select one. (N – 1) (N) (N + 1)

    b) In the above exercise is the value of the index variable available to the programmer at the completion of the loop.

13. Determine the number of times each statement marked with a star (★) will execute.

    Assume: INTEGER I, X = X; LOGICAL N;
             INTEGER ARRAY A(0:10);

    a)   X := 5;
        LOOP: IF DXBZ THEN GOTO NEXT;
                A(X) := 0; ★
                GOTO LOOP;

        NEXT:
    b)   I := 7
                DO
                A (I := I + 1) := 0 ★
                UNTIL LOGICAL (I);

4-20

c)   N := 0;

     WHILE (N := N + 1 $\leq$ 10)

       DO A(N) := 0; ★

d)   N := 0;

     WHILE (N := N + 1) $\leq$ 10

       DO A(N) := 0; ★

14. A frog is 500 inches from a wall. With each jump he makes, he jumps half of the remaining distance to the wall. Write an SPL/3000 program to determine the minimum number of jumps he must take until his distance from the wall is less than .025 inches. (Use a WHILE statement to solve the problem.)

15. Examine the SWITCH declaration and invocation shown and the switch index values provided. Determine which statement label is associated with each value of the index (INDEX).

SWITCH SELECT := FIRST, SECOND, THIRD;

GOTO SELECT (INDEX);

NEXT: next statement;

   .
   .

FIRST: statement;

   .
   .

SECOND: statement;

   .
   .

THIRD: statement;

a) INDEX = 0

b) INDEX = 3

c) INDEX = –1

d) INDEX = 2

16. True or False?
The body of a CASE statement can

 a) contain another CASE statement. T/F

 b) not contain a SWITCH invocation. T/F

 c) contain a null statement. T/F

 d) contain a procedure call statement. T/F

 e) not contain a compound statement. T/F

 f) contain a FOR statement. T/F

17. A program which prints mailing labels reads an individual's title, name, and address. In order to save storage space, the titles have been coded.

  0 = "MR ʌʌ"
  1 = "MRS ʌ"
  2 = "MISS"
  3 = not used
  4 = "DR ʌʌ"   (ʌ represents a blank)

Write a CASE statement which uses the title code (TITLE) to select the actual 4-character title. Store the characters selected in a type double (32 bit) variable called PREFIX (e.g., PREFIX := "MR ∧∧"). (Double integer variables are explained in Section VII.)

Assume: DOUBLE PREFIX; INTEGER TITLE;

This section presents a class of statements that is very machine-dependent — the MOVE and SCAN statements. In many cases proper use of these statements requires a knowledge of explicit stack access on the part of the programmer as these statements use the stack for temporary arguments and results. Since the MOVE and SCAN statements use byte data types and pointers, these two concepts are discussed first.

BYTES

A byte is one-half of a word — 8 bits. SPL/3000 provides facilities for the manipulation of byte variables, arrays, constants, and pointers. Each byte is treated as an 8-bit quantity and can represent numeric or ASCII character data. Because the hardware provides true byte addressing (including element indexing for bytes), two distinct bytes can often be packed conveniently into a single memory word. The following constants can be stored in a byte:

       %377
       12
       %(16)A2 (See Section VII; this is base 16.)
       "A"
       "%"
       %(2)10011101 (See Section VII; this is base 2.)
       FALSE

Byte Variables

Each single byte variable is allocated one word of storage in the primary DB area. References to the variable always refer to the left-most byte of the word (bits 0-7); the other bits are not used. The declaration format is

       BYTE *identifier list* ;

       identifier list   consists of a series of identifiers, each optionally followed by an
                           initialization:

                           identifier := *constant*

If the constant requires more than eight bits for its representation, a warning message is given and the variable is initialized with the least significant eight bits, (the rightmost) of the constant. This constant is stored in the left half of the word allocated for the variable; the right half is filled with zeros. For example,

BYTE CHAR := "A", B := "B", C;

.
.
.

	0 7	8 15	
DB + 0	A	0	CHAR
DB + 1	B	0	B
DB + 2	undefined		C

Byte Arrays

A byte array is an ordered sequence of bytes (packed two per word) in a contiguous area of memory and a byte address (data label) that points to the zero element (byte) of that array. Any byte in the array can be accessed by using the array identifier and a subscript value which identifies the byte desired. The declaration format is

BYTE ARRAY *declaration list;*

declaration list	is a list of declarations, each composed of an identifier followed by a pair of *bounds* in parentheses and an optional *initialization.*
bounds	is a pair of integers that specify the element numbers of the lower and upper bytes of the array (upper bound must not be less than the lower); the format is

$$(lower\ bound : upper\ bound)$$

The number of bytes in the array is (upper bound − lower bound + 1).

initialization	is similar to that for integer and real arrays; the constants are packed into the array, starting with the lowest byte (lower bound), eight bits per element. Only the least significant eight bits of each list element are used. Only the last array in a particular list can be initialized. Uninitialized arrays are not filled with any default value.

For example,

BYTE ARRAY MESSAGE (0:10),

    ANSWER (−10:10),

    QUESTION (0:5) := "WHERE?";

BYTE ARRAY INFO (−10:99) := 20("X"),20("Y"),20("Z"),

    20("Z"),20(%377),10(0);

When a byte array is declared, a data label is allocated in the primary DB area; this contains a byte address which points to the space allocated for the body of the array in the secondary DB area. For consistency, SPL/3000 aligns byte arrays so that the zero element is always on a whole word address. Bytes are packed two per word. The amount of space allocated for a byte array of n elements is (n + 1)/2. Therefore, an array of 10 elements requires 5 words while an array of 11 elemeents require 6 words. If the lower bound is odd, the space required equals (n + 2)/2.

BYTE ARRAY QUESTION (0:5) := "WHERE?"

Address	Contents		Meaning
DB + 0	1000_8		QUESTION (byte address)
DB + 400_8	W	H	QUESTION (0), QUESTION (1)
	E	R	QUESTION (2), QUESTION (3)
	E	?	QUESTION (4), QUESTION (5)

The data labels (addresses) for byte arrays have a format different from data labels for other arrays. Bits 0–14 contain a word address relative to DB, while bit 15 specifies the left or right half of the word. Bit 15 equal to 0 refers to the left byte (bits 0–7 of the word) and bit 15 equal to 1 refers to the right byte (bits 8–15 of the word). The byte address must be shifted right one bit to produce the address of the word in which the byte is located. Effectively, byte addresses are word addresses inflated by a factor of two with an extra bit added for byte address resolution.

Data elements in byte arrays can be accessed by using subscripted array identifiers. These subscript values uniquely identify single bytes in an array and appear enclosed in parentheses following the array identifier.

        CHAR(5)
        MESSAGE <<IMPLIES ZERO ELEMENT>>
        QUESTION (VAR + 3)

The compiler emits code to place the subscript value (byte element number) in the index register. Bits 0–14 of the index register then specify to the hardware the word offset (plus or minus) from the zero element of the array and bit 15 specifies which byte of the word is addressed.

HP 3000 provides many instructions which aid in the accessing and manipulation of bytes:

    Load and Store Bytes (LDB, STB)

    Test byte (BTST)

    Immediate operands (LDI, LDNI, CMPI, CMPN, ADDI, SUBI, MPYI, DIVI, ORI, XORI, ANDI, LDXI, LDXN, ADXI, SBXI)

    Move bytes (MVB)

    Move bytes while (MVBW)

    Scan bytes while (SCW)

    Scan bytes until (SCU)

    Compare bytes (CMPB)

Byte Type Transfer Functions

When bytes are used in an expression, they are actually operated upon by INTEGER arithmetic; the hardware does not provide byte arithmetic. When byte values are mixed with other data types in an expression, it is necessary to use the type transfer functions.

The function BYTE causes the result of an integer or logical expression to be treated as type byte:

BYTE (*integer expression*)

BYTE (*logical expression*)

The function INTEGER causes the result of a byte expression to be treated as type integer:

INTEGER (*byte expression*)

The function LOGICAL causes the result of a byte expression to be treated as type logical:

LOGICAL (*byte expression*)

The function REAL floats a byte expression value into a real number:

REAL (*byte expression*)

POINTERS

A pointer is a type of variable which contains the 16-bit address of another data item in the program. The 16 bits of the pointer represent the address of a variable; when the pointer is used in an expression it creates an automatic indirect reference to the variable (the object of the pointer).

A pointer is very similar to the data label (address) which is allocated in primary DB for an array. In both cases the address can be indexed by specifying an element subscript and indexing is done relative to the location specified by the address (the zero element). Pointers, however, do not have any array space allocated to them and are more general-purpose in use.

While the data label of an array always points to the same location (the zero element of the body of the array), a pointer can be made to point to different data items during the life of a program. That is, the address in a pointer variable can be changed during the program so that the pointer can be used for many different purposes.

Pointer Declaration

Pointer identifiers are declared before use, just like every other identifier in an SPL/3000 program. The format is

type POINTER *pointer list* ;

type	is either BYTE, LOGICAL, INTEGER, REAL, DOUBLE, or LONG. (The default type is LOGICAL.)
pointer list	is a sequence of identifiers separated by commas. Each identifier may be optionally followed by an initialization:

        identifier := @*variable*   ⎱ @ means the address of the data
        identifier := @*array element* ⎰ item in this context

Each pointer is allocated a location in the primary DB area and the address of the data item pointed to is stored in that location (if the pointer is initialized). In this context only, the compiler converts byte addresses to word addresses and vice versa. For example,

```
LOGICAL LSIMPVAR;
BYTE ARRAY BARY(0:79) ; INTEGER ARRAY IARY(0:5) ; REAL ARRAY RARY(0:3);
BYTE POINTER CHAR,
      FIRST := @BARY,
      LAST := @BARY(79);
INTEGER POINTER TESTVAL := @IARY(5),
      NEXTNO;
POINTER BOOL1, <<LOGICAL ASSUMED>>
      BOOLZ := @LSIMPVAR;
REAL POINTER X := @RARY,
      Y,
      Z;
```

Accessing Through Pointers

A pointer can be used in three contexts (assume these declarations).

```
INTEGER A, DATA := -1, SAM := 300, B := 7;
INTEGER POINTER IP := @DATA;
```

- A pointer can be used anywhere the object of the pointer could be used; this generates an automatic indirect reference to the object through the pointer.

A := IP ; <<A = -1>>

- A pointer can be used on the left side of assignment statement to change the value of the object of the pointer. This also generates an automatic indirect reference to the object through the pointer.

IP := SAM + B;   <<DATA = 307>>

• A pointer can be preceded by an @ to refer to the actual contents of the pointer (the data label), not the object of the pointer. This generates a direct reference to the pointer itself.

A := @IP;   <<A = address of DATA>>

```
          A                      IP
   ┌──────────┐         ┌──────────────────┐
   │          │◄────────│   addr. of DATA  │
   └──────────┘         └──────────────────┘
```

@IP := @A;   <<IP = address of A>>

```
          IP            result of expression
   ┌──────────┐         ┌──────────────────┐
   │          │◄────────│    addr. of A    │
   └──────────┘         └──────────────────┘
```

Here is an example that shows a pointer being used to check every third character of an input buffer until it finds one that is equal to "A". Remember that <> means "is not equal to".

```
BYTE ARRAY INPUTSTRING(0:80);

BYTE POINTER CHAR := @INPUTSTRING;

LABEL CONTINUE;
        .
        .
        .
CONTINUE:  IF CHAR <> "A"
        THEN
            BEGIN
                @CHAR := @CHAR + 3;
                GO TO CONTINUE;
            END;
```

Pointers can be changed by using @ and assigning a new address. When @ is used with a pointer, it refers to the contents of the pointer (not the object of the pointer). But when @ is used with any other variable, it refers to the address of the data item, not its contents. The following short example shows both uses:

```
INTEGER AVAL, BVAL;

INTEGER POINTER DATAPTR := @AVAL; <<INITIALIZE TO ADDRESS OF AVAL>>

INTEGER SUM;

    SUM := @DATAPTR + @BVAL;
```

<<SUM NOW EQUALS THE TOTAL OF TWO ADDRESSES: THAT CONTAINED IN DATAPTR—THE ADDRESS OF AVAL—AND THE ADDRESS OF BVAL>>

There are many possible combinations of operations that are possible using pointers, variables, and @.

```
LOGICAL VAR;
POINTER PT1,PT2;
    PT1 := PT2;        <<object of PT2 is stored into object of PT1>>
    PT1 := @PT2;       <<address in PT2 is stored into object of PT1>>
    @PT1 := @PT2;      <<address in PT2 is stored into PT1>>
    @PT1 := PT2;       <<object of PT2 is stored into PT1>>
```

```
PT1 := VAR;        <<value of VAR is stored into object of PT1>>
PT1 := @VAR;       <<address of VAR is stored into object of PT1>>
@PT1 := @VAR;      <<address of VAR is stored into PT1>>
@PT1 := VAR;       <<value of VAR is stored into PT1>>

VAR := PT1;        <<object of PT1 is stored into VAR>>
VAR := @PT1;       <<address in PT1 is stored into VAR>>
```

Indexed Pointers

Pointer variables can be indexed in the same manner as arrays to reference data items. The subscript specified is applied as an element index (plus or minus) relative to the address in the pointer. The original contents of the pointer are not changed by any indexing done with it. For example,

```
REAL ARRAY DATA(0:5);

REAL POINTER PNTR := @DATA(1);
```


The short example above (where every third byte of an input buffer was searched for the character "A") can be written equivalently using an indexed pointer. In this case the contents of the pointer are not changed during program execution.

```
BYTE ARRAY INPUTSTRING(0:80);

BYTE POINTER CHAR := @INPUTSTRING;

INTEGER SUB := 0;

LABEL CONTINUE;
        .
        .
        .
        CONTINUE:  IF CHAR(SUB) <> "A"
                        THEN
                            BEGIN
                                SUB := SUB + 3;
                                GO TO CONTINUE;
                            END;
```

Type Compatibility With Pointers

When a pointer is referenced, the object of the pointer is always treated as if it were of the same type as the pointer. This allows any data area to be accessed on 8, 16, 32, or 48-bit boundaries using byte, integer, real or long pointers. Thus, a logical array can be accessed as if it were a

byte array (8 bits at a time instead of 16) by using a byte pointer which contains a byte address pointing to the zero element of the logical array. The same storage can be accessed as a real array (32 bits at a time) by using a real pointer. Pointers whose types denote double word data types contain word addresses but are indexed two words at a time by the hardware to provide true element indexing. For example,

LOGICAL ARRAY LAR(0:5);

BYTE POINTER BPT := @LAR; <<BYTE ADDRESS OF LAR(0)>>

REAL POINTER RPT := @LAR; <<WORD ADDRESS OF LAR(0), 2 WORD INDEXING>>

STORAGE

```
      +--------------------+--------------------+--------------------+
      |   |     |     |    |    |    |     |    |   |     |     |    |
      +--------------------+--------------------+--------------------+
BPT   (0)  (1)   (2)   (3)  (4)  (5)  (6)   (7)  (8)  (9)  (10)  (11)

LAR   (0)        (1)        (2)        (3)        (4)        (5)

RPT   (0)                   (1)                   (2)
```

STORAGE ACCESS

Within declarations, pointers can be initialized to any type of object; the compiler adjusts the address of the object to the type of the pointer (if necessary). That is, word addresses are converted to byte addresses and vice versa.

When the value of a pointer is changed dynamically, however, the compiler does no compatability adjusting. Thus, any value stored into a pointer during execution of the program is assumed to be a correct address for that type of pointer. For example, if the address of a byte variable is stored into an integer pointer, the byte address is assumed to be the word address of an integer data item when the pointer is used. Unless the byte address is shifted right one bit, this address will be incorrect and may cause a memory bounds violation. (Word addresses are converted to byte addresses by shifting left one bit.) For example,

LOGICAL LVR;

BYTE BVR;

LOGICAL POINTER LPT;

BYTE POINTER BPT;

    <<CORRECT OPERATIONS>>

        @LPT := @LVR; <<address of LVR stored into LPT>>

        @BPT := @BVR; <<address of BVR stored into BPT>>

        @LPT := @BVR & ASR(1); <<address of BVR converted to word address and stored in LPT>>

        @BPT := @LVR & ASL(1); <<address of LVR converted to byte address and stored in BPT>>

    <<INCORRECT OPERATIONS>>

        @LPT := @BVR; <<byte address stored in word pointer>>

        @BPT := @LVR; <<word address stored in byte pointer>>

MOVE STATEMENTS

The SPL/3000 MOVE statements provide high-level access to the hardware move-group instructions:

MOVE (move words)

MVB (move bytes)

MVBW (move bytes while alphabetic and/or numeric, with/without upshifting)

The MOVE statements allow the programmer to move data from one location to another within his program. The statements require a specific destination (array or pointer), a source (array, pointer, or constants), and a count or move condition to determine the quantity of data transferred. This information is placed in the stack before the move instruction executes. During the data transfer operation the count and addresses are repeatedly modified by the move instruction; at the end of the operation they reflect the status of the completed transfer. All temporary values associated with the move are deleted from the stack at the end of the operation unless the optional stack decrement operand is used to specify otherwise.

MOVE WORDS STATEMENT

This statement moves a specified number of contiguous words from a source location to a destination location. The format is

$$
MOVE \; \left\{ \begin{array}{l} \textit{non-byte array (index)} \\ \textit{non-byte pointer (index)} \end{array} \right\} := \left\{ \begin{array}{l} \textit{non-byte array (index)} \\ \textit{non-byte pointer (index)} \end{array} \right\} \textit{,(count), sdec;}
$$

$$\qquad\qquad\qquad\text{destination} \qquad\qquad\qquad\qquad \text{source}$$

non-byte pointer (index)   is a reference through a pointer of any type other than byte (logical, integer, real, long, or double); *(index)* is optional. If no index is present, transfer starts with the data value indicated by the pointer.

non-byte array (index)   is a reference to an array element of any non-byte type. If *(index)* is absent, the transfer starts with the zero element of the array.

sdec   is the optional stack decrement operand; it determines the number of words to delete from the stack after moving data:

sdec = 0, leave destination address, source address, and count.

sdec = 1, leave destination address and source address.

sdec = 2, leave destination address only.

default (blank) = 3, delete all values, leave none.

count   is an integer number of words to be moved (count > 0 means move from left to right, count < 0 means move from right to left, and count = 0 means move no words at all).

Left-Right versus Right-Left Move

The difference between a left-right move (count > 0) and a right-left move (count < 0) is shown in the following diagrams:

Left to Right (count = +4);

Low addresses ... High addresses

Source   Destination

Right to Left (count = −4):

Low addresses ... High addresses

Destination   Source

Stack Decrement Operand

The MOVE (words) statement causes three temporary values to be loaded onto the stack before executing the MOVE instruction:

S − 2		destination address
S − 1		source address
S − 0		word count

After the move operation is complete, destination and source address point to the next word (not moved or overlayed) and can be examined, stored, or left in the stack for use by a subsequent MOVE or SCAN statement. The sdec operand is then used to delete 0, 1, 2, or all 3 of the parameters from on the stack. A blank sdec field generates an automatic sdec of 3— delete all three values from the stack. Count always equals 0 and can safely be deleted (sdec = 1). The sdec mechanism is used for all move-scan statements. The temporary values left on the stack appear as follows:

if sdec = 0

S − 2		destination
S − 1		source
S − 0	0	count

if sdec = 1

S − 1		destination
S − 0		source

if sdec = 2

S − 0		destination

if sdec = 3

S − 0	

The parameters left on the top of the stack can be saved in the following way (assume sdec = 0):

COUNT := TOS;   · <<TOS IS A RESERVED WORD THAT MEANS>>
<<TOP OF STACK.>>

@SOURCE := TOS;   <<ADDRESSES CAN BE SAVED IN POINTER>>
<<VARIABLES. THIS ENABLES>>

@DESTINATION := TOS;   <<THE PROGRAMMER TO REFERENCE>>
<<THE DATA ITEMS LATER.>>

TOS is a reserved word that always refers to the current top of stack (the data at location S – 0). Use of TOS in the example statements above causes the values to be popped off the stack, (S is decremented) and stored in the locations specified:

COUNT := TOS;

(Stack before)

S – 2		destination address
S – 1		source address
S – 0	0	count

@SOURCE := TOS;

(Stack before)

S – 1		destination address
S – 0		source address

@DESTINATION := TOS;

· (Stack before)

S – 0		destination address

(Stack after)

S – 0	

Instead of saving these temporary values it is possible to reuse them in a subsequent move operation by leaving them on the stack and specifying asterisk (*) in place of the appropriate part of the MOVE (words) statement. For example,

MOVE DATA (4) := DATAB (4), (4), 2; <<Moves 4 words, leaves only the>>
<<destination address on stack.>>

MOVE * := DATAC(3), (6); <<Moves 6 words from DATAC into DATA>>
<<starting where previous move left off.>>

In the MOVE (words) statement, only one of the addresses can be a stacked value.

Variations on Move Words

There are additional variations on MOVE (words) which are described in the *Systems Programming Language* manual. One variation allows moving a list of constants into an array or to a location specified by a pointer. The format is

$$\text{MOVE } destination := \begin{Bmatrix} \text{``}string\text{''} \\ (number\ list) \end{Bmatrix}, sdec\ ;$$

string	can be any string constant.
number list	is a list of numerical constants with optional repeat factors (described under initialization of arrays in Section IV).
sdec and destination	are the same as in the MOVE (words) statement discussed previously.

There is no count field in this variation because the constant list implicitly specifies the count to be used. The compiler stores the constants required by this statement in the code generated for the program. This statement is very useful for dynamic initialization of arrays:

MOVE DBUF := (5(0,1,2,3,4),10,20,99);

MOVE * := (10(0,1,2,3,4)),1;

Examples of MOVE (words) Statements

LOGICAL ARRAY DBUF(0:100),CBUF(0:100);

LOGICAL POINTER DMARK;

INTEGER WCOUNT;

   .
   .
   .

MOVE DBUF(WCOUNT/10) := "$123.45",2; <<LEAVE DESTINATION>>

@DMARK := TOS;   <<SAVE DESTINATION ADDRESS>>

   .
   .
   .

DBUF := 0;   <<PUT 0 IN ZERO ELEMENT OF DBUF>>

MOVE DBUF(1) := DBUF,(WCOUNT),2; <<ZERO DBUF;LEAVE DESTINATION>>

MOVE * := (1,3,4,7,9,11,23); <<USE STACKED DESTINATION ADDRESS>>

   .
   .
   .

MOVE CBUF := DBUF,(101);

MOVE BYTES STATEMENT

Moving bytes is very similar to moving words—with two exceptions: the destination/source addresses are byte addresses and the count is a number of bytes, not a number of words. There are two formats:

$$\text{MOVE } \begin{Bmatrix} byte\ array\ (index) \\ byte\ pointer\ (index) \\ * \end{Bmatrix} := \begin{Bmatrix} byte\ array\ (index) \\ byte\ pointer\ (index) \\ * \end{Bmatrix}, (count), sdec\ ;$$

                   destination   *source*

$$\text{MOVE} \left\{ \begin{array}{l} byte\ array\ (index) \\ byte\ pointer\ (index) \\ * \end{array} \right\} := \left\{ \begin{array}{l} \text{``string''} \\ (number\ list) \end{array} \right\} \ , sdec\ ;$$

(*index*)   is optional where shown,

   *   means the address is already stacked,

(*count*)   is the number of bytes to move (positive means a left-to-right move; negative means a right-to-left move; 0 means no move),

  sdec   (stack decrement operand) is the same as that discussed with move words.

When constants are used as the source operand they are moved into the destination one byte at a time. With a string constant, one character is moved to each destination byte from the source string. With a number list, the rightmost (least significant) 8 bits of each source number are moved into each destination byte. With these exceptions, the MOVE (bytes) statement works the same way as the MOVE (words) statement (in terms of stacked values, count sign, sdec, etc.). For example,

BYTE ARRAY HERE(0:8),
                 THERE(3:13);

MOVE THERE(13) := HERE(6), (–5);

HERE   0 1 2 3 4 5 6 7 8

| A | B | C | D | E | F | G | H | I |

| X | X | X | X | X | X | C | D | E | F | G |

THERE   3 4 5 6 7 8 9 10 11 12 13

MOVE BYTES WHILE STATEMENT

The MOVE (bytes) WHILE statement moves bytes from one location to another as long as the character being moved is of a specified type (alphabetic and/or numeric). Also, lowercase alphabetic characters can be upshifted as they are moved. The format is

$$\text{MOVE} \underset{destination}{\left\{ \begin{array}{l} byte\ array\ (index) \\ byte\ pointer\ (index) \\ * \end{array} \right\}} := \underset{source}{\left\{ \begin{array}{l} byte\ array\ (index) \\ byte\ pointer\ (index) \\ * \end{array} \right\}} \text{WHILE } cond,\ sdec\ ;$$

(*index*)   is optional where shown,

   *   means a stacked address,

 cond   specifies the conditions for continuing the move:
        A   Move while character is alphabetic
        AN Move while character is alphabetic or numeric
        AS Move while character is alphabetic and upshift lowercase
        N   Move while character is numeric
        ANS Move while character is alphabetic (upshift lowercase) or numeric

 sdec   is the stack decrement operand; in MOVE (bytes) WHILE there are only two values loaded onto the stack (source and destination addresses; no count—the test condition is a part of the physical MVBW instruction). The default value is sdec = 2 or delete both values.

After the characters have been moved and a stop condition has occurred, the source and destination addresses point to the next character not moved in the source and destination buffers. It is essential to save at least the destination address, since this is the only way the programmer can determine the number of characters actually moved. The mechanism for saving these values is the same as with MOVE (words) and MOVE (bytes), except that there are only two of them:

```
       if sdec = 0                    if sdec = 1                 if sdec = 2
       ┌──────────┐                                              ┌──────────┐
S – 1  │          │ destination            ┌──────────┐         │          │
S – 0  │          │ source         S – 0   │          │ destination  S – 0 │          │
       └──────────┘                        └──────────┘         └──────────┘
```

Assume that sdec = 0:

```
        @SOURCE := TOS;                (Stack before)

                                              ┌──────────┐
                                       S – 1  │          │ destination address
                                       S – 0  │          │ source address
                                              └──────────┘

        @DESTINATION := TOS;           (Stack before)

                                              ┌──────────┐
                                       S – 0  │          │ destination address
                                              └──────────┘

                                       (Stack after)

                                              ┌──────────┐
                                       S – 0  │          │
                                              └──────────┘
```

Condition Codes on MOVE (bytes) WHILE

At the termination of a MOVE (bytes) WHILE statement the condition code is set to show the type of the last character examined (but not moved):

```
SPECIAL  = CCL (<)
ALPHA    = CCE (=)
NUMERIC  = CCG (>)
```

These conditions can be checked using an IF statement.

        IF < THEN..........ELSE........;

Examples of MOVE (bytes) WHILE

<<Example One>>

        BYTE ARRAY SORC(0:8) := "A1QRT5*B4";
        BYTE ARRAY DEST(0:11) := 12("X");
        MOVE DEST := SORC WHILE AN;

```
SORC    0   1   2   3   4   5   6   7   8
      +---+---+---+---+---+---+---+---+---+
      | A | 1 | Q | R | T | 5 | * | B | 4 |
      +---+---+---+---+---+---+---+---+---+
       |
       |              move terminates on special character *
       v
      +---+---+---+---+---+---+---+---+---+---+---+---+
DEST  | A | 1 | Q | R | T | 5 | X | X | X | X | X | X |
      +---+---+---+---+---+---+---+---+---+---+---+---+
        0   1   2   3   4   5   6   7   8   9   10  11
```

<<Example Two>>

    BYTE ARRAY STMT (0:79), WORD (0:79);

    BYTE POINTER SORCPTR, DESTPTR;

    LABEL FIXNUMBER, FIXSPECIAL;

        MOVE WORD := STMT WHILE A,0;

        @SOURCPTR := TOS; <<SAVE SOURCE ADDRESS>>

        @DESTPTR := TOS; <<SAVE DESTINATION ADDRESS>>

        IF > THEN GO TO FIXNUMBER <<CHECK TYPE OF TERMINATING
                                            CHARACTER>>

            ELSE GOTO FIXSPECIAL;
                .
                .
                .
    FIXNUMBER: ----------;
                .
                .
                .
    FIXSPECIAL: ----------;

EXAMPLE 5-1. SYMBOL TYPE SORTER

This program first fills a buffer with characters from the terminal. Then numeric and alphabetic fields are moved to separate buffers with upshifting of all lowercase alphabetic characters. When the program finds a special character, it stops processing the input string buffer, outputs the two print buffers and terminates. This illustrates the MOVE (bytes) WHILE statement (with/without upshifting), pointers, and use of @ to specify addresses.

Input/Output

```
INPUT AN ASCII STRING
ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789Ø!"#$%&'()ASDFGHJ1234567
ABCDEFGHIJKLMNOPQRSTUVWXYZ
0123456789Ø
INPUT AN ASCII STRING
ABCDEF123GHIJKL456MNOPQRS789TUVWXYZØ
ABCDEFGHIJKLMNOPQRSTUVWXYZ
1234567890
```

```
INPUT AN ASCII STRING
126CRANKSHAFT0014932#
CRANKSHAFT
1260014932
```

Listing

```
BEGIN  <<EXAMPLE 5-1. SYMBOL TYPE SORTER>>
COMMENT:
        THE PROGRAM INPUTS AN ASCII DATA RECORD (LENGTH <= 72
        CHARACTERS). ALPHABETIC AND NUMERIC CHARACTERS ARE MOVED
        TO SEPARATE BUFFERS FOR OUTPUT. (ALL LOWER CASE ALPHABETIC
        CHARACTERS ARE UPSHIFTED AND MOVED.) WHEN A SPECIAL
        CHARACTER IS FOUND, PROCESSING STOPS AND BOTH BUFFERS ARE
        OUTPUT.
        NOTE: "INPUT" AND "OUTPUT" ARE DUMMY PROCEDURES WHICH SIMULATE
        INPUT, OUTPUT, AND CONVERSION - THEY ARE NOT PART OF SPL/3000;
BYTE ARRAY REQST(0:20):="INPUT AN ASCII STRING";
BYTE ARRAY DATA(0:72):=73(%15);
BYTE ARRAY ALPHADATA(0:72):=73(" ");
BYTE ARRAY NUMBRDATA(0:72):=73(" ");
BYTE POINTER DPNTR:=@DATA,
             APNTR:=@ALPHADATA,
             NPNTR:=@NUMBRDATA;
LABEL SCANR;
<<END OF DECLARATIONS>>
            OUTPUT(REQST);  <<REQUEST RECORD INPUT>>
            INPUT(DATA);  <<READ RECORD>>
SCANR:      MOVE APNTR:=DPNTR WHILE AS,0;  <<MOVE ALPHABETIC-UPSHIFT>>
            @DPNTR:=TOS;
            @APNTR:=TOS;
            MOVE NPNTR:=DPNTR WHILE N,0;  <<MOVE NUMERIC>>
            @DPNTR:=TOS;
            @NPNTR:=TOS;
            IF = THEN GOTO SCANR;  <<LAST CHARACTER ALPHABETIC>>
            OUTPUT(ALPHADATA);  <<PRINT ALPHA STRING>>
            OUTPUT(NUMBRDATA);  <<PRINT NUMBER STRING>>
END  <<SYMBOL TYPE SORTER>>.
```

SCAN STATEMENTS

The purpose of a SCAN statement is to examine a contiguous string of bytes while looking for a specified character without actually moving any data. When the statement ends, pointers and indicators are left to show what was found and where. In SPL/3000 there are two SCAN statements:

SCAN Until Statement (SCU Instruction)

SCAN While Statement (SCW Instruction)

SCAN UNTIL STATEMENT

The SCAN UNTIL statement scans a string of contiguous bytes from left-to-right until a test character or a terminal character is found. The format is

$$\text{SCAN} \left\{ \begin{array}{l} \textit{byte array (index)} \\ \textit{byte pointer (index)} \\ * \end{array} \right\} \text{UNTIL } \textit{testword , sdec} ;$$

(*index*) is optional.

* means the data address is already stacked.

testword is a constant, variable (integer, logical), or *"char* char*"* that determines the test conditions and is interpreted as follows:

terminal character		test character
0	7 8	15

sdec is the stack decrement operand and operates as in move statements (the default is sdec = 2 since there are two temporary values used in this statement).

When the scan operation completes, the carry bit of the status register reflects the terminating conditions. CARRY indicates that the terminal character was found by the scan. NOCARRY indicates that the test character was found. This can easily be tested with an IF statement:

```
IF CARRY THEN . . . . . ELSE . . . . . ;
IF NOCARRY THEN . . . . . ELSE . . . . . ;
```

> CAUTION: *Any arithmetic operation done between the scan and the testing of CARRY can change the setting of CARRY.*

The stack decrement operand determines what values are left on the stack. Note that the scan operation updates the byte address to point to the byte that met the condition but does not change the testword.

	if sdec = 0		if sdec = 1		if sdec = 2
S – 1	byte addr.				
S – 0	testword	S – 0	byte addr.	S – 0	

Sample SCAN UNTIL Operation

```
BYTE POINTER PTR;

BYTE ARRAY CHAR (0:30) := "AAAAAAAAAAAA0AAAAAAAAAAAAAAAA";

SCAN CHAR UNTIL "Z0",1;

@PTR := TOS;
```

Steps in Operation:

1. The source byte address and the testword are loaded onto the stack.

2. The SCU instructions tests the source bytes and increments the byte address.

3. When a match is found the stacked address points to the matching byte.

4. CARRY is set if the terminal character is found, NO CARRY if the test character is found.

5. The testword is popped off (sdec = 1) but the address of the matching byte is left.

6. The address of the match is popped off and stored in PTR.

S →
byte address
Z	0

S →
addr. of match-char
Z	0

S →
| address of match |

S →
| |

Another Sample of SCAN UNTIL

BYTE ARRAY INPUT (0:79);

LOGICAL DOT'DOLLAR := %27044; << .$ >>

LABEL PROCS;

    SCAN INPUT UNTIL DOT'DOLLAR, 1;   <<SCAN FOR PERIOD OR $>>

    IF CARRY THEN GOTO PROCS;   <<PERIOD (.) FOUND, PROCESS IT>>

    SCAN * UNTIL ".X";   <<START SCAN ON $ AND LOOK>>

                                   <<FOR X OR PERIOD>>

PROCS: - - - - - - - - - - - - - - -

SCAN WHILE STATEMENT

This variation of the SCAN statement examines a string of bytes as long as all the characters found match the test character. The scan is terminated when a character which does not match the test character is found or when the terminal character is found. The format is

$$
\text{SCAN} \left\{ \begin{array}{l} byte\ pointer\ (index) \\ byte\ array\ (index) \\ * \end{array} \right\} \text{WHILE } testword,\ sdec\ ;
$$

testword   is a one-word variable or constant and has this format:

terminal character	test character
0 7	8 15

sdec   is as before (default is 2 to pop 2 values.)

The operation of SCAN WHILE is similar to that of SCAN UNTIL. The parameters are located on the stack. Upon completion, the byte address points to the first byte that does not match; CARRY is set if the terminal character was found; the condition code is set to indicate the type of the unmatching character:

special	(CCL,<)
alphabetic	(CCE,=)
numeric	(CCG,>)

In order to determine how many characters were scanned or where the scan stopped, it is necessary to retain the byte address on the stack (sdec = 0 or 1).

Sample SCAN WHILE

<<Example One>>

```
BYTE POINTER PTR;

BYTE ARRAY CHARS;

SCAN CHARS WHILE ".0",1;      <<SCAN WHILE 0 OR UNTIL PERIOD IS FOUND>>

@CPTR := TOS;                 <<SAVE MISMATCH ADDRESS>>
```

<<Example Two>>

```
BYTE ARRAY BUFR (0:100);

BYTE POINTER BPNTR := @BUFR;

LOGICAL CRBLNK := %6440;     <<CARRIAGE RETURN, BLANK>>

LABEL LINEEND;

    @BPNTR := @BUFR(3);

    SCAN BPNTR(2)WHILE CRBLNK,1;  <<SCAN UNTIL FIRST NON-BLANK OR CR>>

    IF CARRY THEN GOTO LINEEND;  <<CHECK FOR CARRIAGE RETURN>>

    @BPNTR := TOS;               <<SAVE ADDRESS OF NON-BLANK CHARACTER>>
        .
        .
        .
LINEEND: - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

TESTING BYTES AND STRINGS

In an SPL/3000 logical expression it is possible to compare two strings of bytes to determine if a relation between them is true or false. The instruction actually used is the Compare Bytes instruction (CMPB) and the format is very similar to MOVE and SCAN. The byte strings are compared, one character at a time, at their numeric values. The format is

$$\left\{ \begin{array}{l} byte\ array\ (index) \\ byte\ pointer\ (index) \\ * \end{array} \right\} \quad relop \quad \left\{ \begin{array}{l} byte\ array\ (index),\ (count) \\ byte\ pointer\ (index),\ (count) \\ *,\ (count) \\ "string" \\ (number\ list) \end{array} \right\} \quad ,\ sdec$$

STRING ONE   STRING TWO

Where all parameters are as described for moves and scans (stack contains address of string one, address of string two, and count). The default for sdec is 3 (pop all three temporary values), and *relop* specifies a relation between string one and string two (=, $<>$ (not equal), $>$, $>=$, $<=$).

The bytes of the two strings are compared one at a time until a comparison fails or the count reaches 0. The result of a comparison is a TRUE value if the strings match when the count reaches 0; otherwise, the result is FALSE. At the end of a byte comparison the addresses point to the first pair of characters that do not meet the comparison. These addresses can be saved using *sdec*, but this is only feasible if the addresses are not destroyed by something else done in the logical expression.

IF TARGET = "BOB SMITH",0 THEN NUM'CHARS := 0;

    ELSE BEGIN   NUM'CHARS := TOS; <<COUNT IN S – 0>>

            TEMP1 := TOS;   <<ADDRESS OF STRING TWO>>

            TEMP2 := TOS;   <<ADDRESS OF STRING ONE>>

   END;

Another construction allows testing single bytes in logical expressions to determine whether they are of a particular type. The format is

$$\left\{ \begin{array}{l} \textit{byte array (index)} \\ \textit{byte pointer (index)} \\ \textit{byte variable} \end{array} \right\} \left\{ \begin{array}{l} = \\ <> \end{array} \right\} \left\{ \begin{array}{l} \text{ALPHA} \\ \text{NUMERIC} \\ \text{SPECIAL} \end{array} \right\}$$

    = means "is type."

    $<>$ means "is not type."

  ALPHA means character is an upper and lower case letter (A to Z and a to z).

 NUMERIC means character is a digit (0 through 9).

 SPECIAL means character is any other character.

The result of a byte test is a TRUE or FALSE.

For example,

  BYTE CHAR;

  IF CHAR = ALPHA THEN . . . . .

       ELSE . . . . . ;

EXAMPLE 5-2. MARK DELIMITER CHARACTERS

This program illustrates the use of the SCAN and MOVE statements. The user inputs a single character to be used as a delimiter. The user then inputs a character string to be tested; the program scans the test string and marks each occurrence of the delimiter character with a marker character (I) below it.

The MOVE statement is used to initialize buffers only; two methods of setting up a MOVE are shown — one using byte data labels and the other using pointers.

Input/Output

```
INPUT A DELIMITER CHARACTER

INPUT A CHARACTER STRING
1234567890*ABCDEF***!"#S%&'()=<>?*@+,./?GHIJKLMNO**PQRSTUV**ZXC********
          I   III                 I              II       II   IIIIIIII
INPUT A DELIMITER CHARACTER
E
INPUT A CHARACTER STRING
NOW IS THE TIME FOR ALL GOOD MEN TO COME TO THE AID OF THE PARTY!
         I   I                    I        I      I          I
```

Listing

```
BEGIN   <<EXAMPLE 5-2. MARK DELIMITER CHARACTERS>>
COMMENT:
        THIS PROGRAM MARKS THE APPEARANCE OF A DELIMITER CHARACTER
        WITHIN A SOURCE ASCII STRING.
        NOTE: "INPUT" AND "OUTPUT" ARE DUMMY PROCEDURES WHICH SIMULATE
        INPUT, OUTPUT, AND CONVERSION - THEY ARE NOT PART OF SPL/3000;
INTEGER CHARPTR=S,
        TESTWORD:=0;
BYTE ARRAY STRING(0:71):="INPUT A DELIMITER CHARACTER ";
BYTE ARRAY MARKER(0:71):="INPUT A CHARACTER STRING";
BYTE POINTER STRINGADR:=@STRING,   <<(STRINGADR)=ADDRESS OF STRING>>
             MARKERADR:=@MARKER,    <<(MARKERADR)=ADDRESS OF MARKER>>
             MARKPOINT;
LABEL LOOP;
        OUTPUT(STRING);   <<PRINT DELIMITER REQUEST>>
        INPUT(STRING);   <<INPUT DELIMITER CHARACTER>>
        TESTWORD.(8:8):=STRING;   <<NULL,TESTCHARACTER>>
        STRING:=0;   <<ZERO STRING BUFFER>>
        MOVE STRING(1) := STRING,(71);
        OUTPUT(MARKER);   <<PRINT STRING REQUEST>>
        INPUT(STRING);   <<INPUT TEST STRING>>
        MARKERADR:=" ";   <<BLANK MARKER BUFFER>>
        MOVE MARKERADR(1) := MARKERADR,(71);
        TOS:=@STRINGADR;   <<SET UP STACK FOR SCAN>>
LOOP:
        SCAN * UNTIL TESTWORD,1;   <<SCAN FOR DELIMITER>>
                                   <<NULL CHARACTER TERMINATES>>
        IF NOCARRY THEN   <<DELIMITER FOUND>>
                        BEGIN   <<COMPUTE MARKER ADDRESS>>
                        @MARKPOINT:=@MARKERADR+(CHARPTR-@STRINGADR);
                        MARKPOINT:="I";   <<PUT MARKER IN BUFFER>>
                        CHARPTR:=CHARPTR+1;   <<BUMP STRING ADDRESS>>
                        GOTO LOOP;
                        END;
        DEL;   <<DELETE STRING ADDRESS>>
        OUTPUT(MARKER);   <<OUTPUT MARKERS>>
END   <<CHARACTER MARKER>>.
```

EXERCISES FOR SECTION V

1. Identify the invalid BYTE declarations.

   a) BYTE A := %101;

   b) BYTE ARRAY ASCIIBUF (0:3) := "AB", -1, "Z";

   c) BYTE CHARS := "AB";

   d) BYTE ARRAY STR1 (0:3) := "ABCD", STR2 (0:3) := "EFGH";

   e) BYTE BIGNO := 12.34;

   f) BYTE FLAG := TRUE;

   g) BYTE ARRAY COMPOSITE (0:5) := 100, "A", %14, "234";

   h) BYTE ONE := "1", TWO := "2";

   i) BYTE ARRAY BIGSTRING (0:9) := "**** ERROR NUMBER ****";

   j) BYTE W, X, Y, Z, LAMBDA := "L";

2. Describe the results of the three assignment statements by filling in the tables provided.

   ```
   BYTE ARRAY ASCII (0:3) := "ABCD";
   BYTE ARRAY NULL (0:3) := 4(0);
   LOGICAL WORD := 0;
   WORD := ASCII;
   NULL := ASCII;
   TOS := ASCII;
   ```

WORD		ARRAY NULL		TOS	

3. Write the SPL/3000 statements to concatenate bytes A and B and place the result in

   ```
                   0    7 8    15
   WORD, such that WORD = [  A  |  B  ]
   ```

   BYTE A := "A", B := "B";

   LOGICAL WORD;

4. How many words of storage does this declaration allocate?

   INTEGER INDEX = X;

5. For each of the octal byte addresses below, convert the address to a word address and then back to a byte address. Is the result always the same as the original byte address?

	Byte Address	Word Address	Final Byte Address
Address 1	%1023		
Address 2	%1460		
Address 3	%2577		

Hint: A byte address is converted to a word address by dividing by 2; a word address is converted to a byte address by multiplying by 2.

6. How many words of storage are allocated for these pointer declarations?

    a) BYTE POINTER INBUF := @ ASCIIBUF;

    b) REAL POINTER DECNO;

    c) INTEGER POINTER FIXIT;

    d) LOGICAL POINTER STICK := @ TABLE;

    e) LOGICAL POINTER INDEX = X;

7. Which of the following constants will fit into a single byte (8 bits) without truncation?

    a) –1

    b) "–1"

    c) %377

    d) 27.3E–6

    e) "$"

8. Evaluate the following statements and compute the result stored in TOTAL.

    INTEGER TOTAL, FIXNO := –1;

    BYTE A := 37, B := 144, C := 3;

    TOTAL := (A + BYTE (FIXNO) + B) * C;

9. Assume this declaration:

    INTEGER ARRAY SAM (1:10) := 1, 2, 3, 4, 5, 6, 7, 8, 9, 10;

    Declare an integer pointer IPNTR that points to the first element of array SAM.

10. Assume these declarations:

    INTEGER ARRAY NUMBER (0:9) := 1, 2, 3, 4, 5, 6, 7, 8, 9, 10;
    INTEGER POINTER NPNTR := @ NUMBER;

    What is the result of executing each of these statements?

    a) NPNTR := NPNTR + 1;
    b) NPNTR(1) := NPNTR(2) + NPNTR(3) + NPNTR(4) + NPNTR(5);
    c) @NPNTR := @NPNTR + 7;
    d) @NPNTR := @NUMBER;
    e) NPNTR := @NUMBER(9);

11. Write SPL/3000 statements to fill each word of an integer array BUFF with a binary pattern of alternating 1 and 0 bits (e.g., 101010 . . . .). Use a MOVE statement to solve the problem.

Assume:

INTEGER ARRAY BUFF (0:99);

12. Examine the MOVE statement and data declarations shown below. Supply the appropriate stack decrement operand for the MOVE statement, and write the additional statements required to save the final source address in INPNTR and the final destination address in OUTPNTR.

LOGICAL ARRAY INBUF (0:1000),

OUTBUF (0:1000);

LOGICAL POINTER INPNTR, OUTPNTR;

MOVE OUTBUF := INBUF, (128), sdec;

13. Write a statement to move characters from SOURCEBUF to DESTBUF until a non-alphabetic character is found. Use an additional statement to make certain that SOURCEBUF contains a non-alphabetic character before the move by storing an asterisk (*) into the buffer as the last character.

Assume:

BYTE ARRAY SOURCEBUF (0:80),

DESTBUF (0:80);

14. Rewrite the solution to exercise 13. After the move has completed, compute the number of characters actually moved, and store the value in an integer variable (COUNT). Determine the type of character which terminated the move. Set a logical variable (NUMBER) to true if the character was numeric, otherwise set it to false.

Assume:

BYTE ARRAY SOURCEBUF (0:80),

DESTBUF (0:80);

INTEGER COUNT;

LOGICAL NUMBER;

15. Assume a byte array STRING starts at DB + %1000 (byte address) and contains the following:

A	B	C	D	E	F	G	H	I	.

↑
DB + 1000

For the execution of this SCAN statement

SCAN STRING UNTIL ".E", 0;

the stack contains

S–1	%1000
S–0	.E

(DB relative byte address)

How does the stack appear after the execution of the SCAN statement?

16. Write a statement to determine whether the content of byte array DATA is the string "END OF INPUT". If that is the content, set the logical variable FLAG to true.

Assume:

BYTE ARRAY DATA (0:11);
LOGICAL FLAG := FALSE;

SECTION VI
Procedures and Subroutines

This section discusses two types of SPL/3000 subprogram units: procedures and subroutines. The PCAL and SCAL machine instructions provide entry to procedures and subroutines, respectively; EXIT and SXIT provide for returning to the point of the call.

PROCEDURES

A **procedure** in SPL/3000 has all the attributes of a conventional subroutine, but is much more powerful. Functionally, a procedure is a contiguous block of machine instructions which is called (with parameters) to perform a specific function. The power and importance of procedures in the HP 3000 cannot be overstressed. They are the organizational principles behind code, data and system control:

- Saving/Restoring Environment

  The calling environment is saved in the stack when a procedure is called and restored when a procedure exits.

- Local variables

  Each procedure called has a unique Q register setting when it executes. This makes it possible to allocate and address local data variables in the Q+ area of the stack.

- Dynamic Temporary storage

  The Q register is given a fresh setting whenever a procedure is entered and reset when the procedure exits.

- Segmentation

  Code segments consist of one or more procedures. The Procedure Call instruction allows control to be transferred from segment to segment.

- Virtual Memory

  The Procedure Call instruction automatically causes an absence trap if the requested procedure is in a segment that is not present in main memory (the operating system then makes the segment present). If the segment is already present, the instruction directly transfers to the procedure. This provides a virtual memory for code.

- Code Sharing

  Since code cannot be modified, it is re-entrant and can be shared (used by several processes concurrently). All procedures are referenced through a system-defined code segment table. Thus all users can concurrently access the same copy of a common procedure (such as the sine or cosine function).

- System Control

  The functions of the operating system are provided by procedures called Intrinsics. A privileged mode bit and an uncallable bit are examined by the PCAL and EXIT instructions to restrict access to these Intrinsics.

Attributes of a Procedure

Each procedure can have many different attributes. Some of the most common are

- Type

  If a procedure is typed, it returns a value of a specified type (integer, real, etc.) in place of its name (in an expression).

- Parameters

  Variables, arrays, constants, labels, pointers, and other procedures can be passed to a procedure as parameters. For each parameter, either a copy of the parameter value or the address of the parameter is passed to the procedure in the Q– area of the stack.

- Local Variables

  Procedures can declare local variables that are known only within the procedure and are allocated space in the Q+ area when the procedure is called. Thus, they occupy space only when a procedure call is active. All items that can be declared in the main program (except for procedures) can also be declared within procedures. Another type of data construct — the own variable — is known only within a procedure, but is allocated in the DB area and thus has a value throughout the life of a process.

- Main Body

  The main body (or code) of a procedure can consist of any sequence of statements, including calls to procedures. **Recursion** — procedures calling themselves — is allowed.

A Typical Procedure

Suppose we have a procedure named BLANKBUF whose purpose will be to fill a buffer (array) with ASCII blanks.

- What data does BLANKBUF need?

  The address of the buffer to be filled and the number of words to blank, minus one.

- What data must BLANKBUF return?

  Only the filled buffer.

● How would a call to BLANKBUF look?

BLANKBUF (ARRAY , 30) ;

    Procedure   Buffer   Number
      name   address of words

● How would these two parameters be received?

They are loaded onto the stack before the call to BLANKBUF. An address is loaded for the buffer and a constant is loaded for the number of words.

● How does BLANKBUF know about its parameters?

All of the parameters of a procedure are defined when the procedure is declared. Following is part of the declaration of BLANKBUF:

      PROCEDURE BLANKBUF (BUFFER, COUNT);

          VALUE COUNT;

          LOGICAL ARRAY BUFFER;

          INTEGER COUNT;

BUFFER and COUNT are called formal parameters. When the procedure is called, each formal parameter must be matched by an actual parameter compatible with the type specified for a formal parameter. Above, the address of a logical array is expected in place of BUFFER and an integer constant value is expected in place of COUNT.

● Where does BLANKBUF get the blank characters to use for filling the buffer?

BLANKBUF declares a local variable (BLANKWORD) which is initialized with two blanks. This follows the specification of parameters:

      PROCEDURE BLANKBUF (BUFFER, COUNT);

          VALUE COUNT;

          LOGICAL ARRAY BUFFER;

          INTEGER COUNT;

          BEGIN

              LOGICAL BLANKWORD := %20040;

● How does BLANKBUF perform the blank fill?

The statements to perform the function of the procedure follow the local variable declarations. The procedure declaration is terminated by an END;. Here is the complete declaration of BLANKBUF:

      PROCEDURE BLANKBUF (BUFFER, COUNT);

          VALUE COUNT; LOGICAL ARRAY BUFFER;

          INTEGER COUNT;

          BEGIN

              LOGICAL BLANKWORD := %20040; <<blank, blank>>

              BUFFER := BLANKWORD;

              MOVE BUFFER (1) := BUFFER, (COUNT-1);

          END <<END BLANKBUF>>;

Declaring Procedures

As was seen in the BLANKBUF example, a procedure must be declared in order to specify exactly what it is to do and what local variables it needs. The parameters must be specified in sufficient detail to enable the compiler to generate proper code for calls upon the procedure.

The declaration of procedures occurs in the declaration part of a main program, *after* all the data declarations. (Procedures cannot be declared locally—within other procedures.) A procedure declaration has two parts: a head and a body.

Elements of the procedure head, in required order of appearance:

Syntactic Element	Explanation
type	If a procedure is typed, it is a function procedure and returns a value of the specified *type* (byte, integer, logical, real, double, long). Functions are discussed later in this section.
PROCEDURE *name*	The *name* is the name which is used to call the procedure; it can be any valid identifier.
formal parameters	The *formal parameters* are dummy identifiers separated by commas and enclosed in parentheses. These identifiers have meaning only within the procedure (they may duplicate global identifiers) and assume the values of the actual (calling) parameters when the procedure is invoked. A semi-colon must follow the rightmost parenthesis.
value part	Parameters can be passed either by value or by reference. Those to be passed by value must appear in a list after the word VALUE. When a parameter is called by value, a copy of the actual parameter value is loaded into the stack. This location can be accessed by the procedure with Q minus addressing. Although the local copy of the value can be changed, the contents of the original actual parameter remain unchanged. When a parameter is passed by reference (the default case), the address of the parameter is loaded onto the stack and the procedure references the data indirectly through it. This means that the procedure can modify the original contents of the actual parameter.
specification part	The *specification part* must include the type of every formal parameter (including those called by value). Variables, arrays, pointers, labels, and procedure names can be passed as actual calling parameters. There is no parameter checking with calls to procedures passed as parameters. The form of specification is the same as a declaration, except that arrays are specified without bounds. (For special considerations in specifications, see *Systems Programming Language (HP 03000-90002)*
option part	The *option part* specifies special options of the procedure. The most common options are

OPTION FORWARD;	Allows two or more procedures to call each other. (See "Recursive Procedures.")
OPTION EXTERNAL;	Allows procedures to be compiled separately. (See "Intrinsics.")

OPTION VARIABLE;   Allows procedures to be called with a
variable number of parameters. (See
"Intrinsics.")

All options are described fully in *Systems Programming Language*.

Elements of the procedure body:

BEGIN

data group   The data group contains any data declarations local to the procedure.
These declarations are allocated space in the Q+ area of the stack and,
thus, can only be referenced within this procedure. All of the SPL/3000
structures can be declared locally except for procedures. A procedure
can be declared within another procedure only if its procedure body
(code) is external (OPTION EXTERNAL). Since intrinsics are external
system procedures they can be declared within procedures. Subroutines
can be declared in procedures. Local arrays are allocated Q+ locations
and are addressed indirectly through a Q+ data label; they cannot be
initialized since the space for them is not allocated until the procedure
is called. Note that calling parameters can be used in declarations (for
example, to specify variable bounds for an array; see Section VII).
The following example shows what space would be allocated on the
stack when a procedure called RUN is entered:

Procedure   **Stack**

PROCEDURE RUN;

  BEGIN   Q →

    INTEGER A;   Q + 1 | A |

    POINTER P;   Q + 2 | P |

    LOGICAL ARRAY LAR(0:4); Q + 3 | pointer to zero element of LAR |

     .
     .   Q + 4 | LAR(0) |
     .
  END; | LAR(1) |

                     | LAR(2) | } LAR

                     | LAR(3) |

             S → Q + 8 | LAR(4) |

statements   The body of a procedure consists of SPL/3000 statements. These
statements can reference the formal parameters, entities declared local
to this procedure, and all globally declared entities (except subroutines).

END;

If we review the first example procedure (BLANKBUF) we can see the parts described above:

```
<<HEAD>>
<<NO TYPE>>
PROCEDURE BLANKBUF <<NAME>>
      (BUFFER,COUNT) ; <<FORMAL PARAMETERS>>
      VALUE COUNT ; <<VALUE PART>>
      LOGICAL ARRAY BUFFER ; <<SPECIFICATION PART>>
      INTEGER COUNT ;
      <<EMPTY OPTION PART>>
<<BODY>>
      BEGIN
            LOGICAL BLANK WORD := %20040 ; <<DATA GROUP>>
            BUFFER := BLANKWORD; <<STATEMENTS>>
            MOVE BUFFER(1) := BUFFER,(COUNT-1);
      END; <<END DECLARATION>>
```

All parts of the procedure head are optional except for PROCEDURE *name*. The data group is an optional part of the body. In addition, if there is no data group (no local declarations) and only one statement in the rest of the body, the BEGIN-END pair can be omitted.

Calling Procedures

Procedures can be called by the main program, by other procedures, or by themselves. There are two methods of calling a procedure. One is in a Procedure Call statement; the other is in an expression (typed procedures only). This second use is discussed later in this section under "Functions." The Procedure Call statement has the following format:

name (*actual parameter list*);

name	is the identifier of the procedure to be called.
actual parameter list	is a list of actual parameters separated by commas that correspond (one-to-one) in type with the formal parameters of the procedure. VALUE parameters can be constants, identifiers, expressions, or assignment statements. Reference parameters can be identifiers only, not constants. The compiler generates code to determine the value or address of each actual parameter in turn and leave them on the stack.

If there are no parameters, the entire parenthetic unit is omitted. When assignment statements are used as parameters, the result of the expression is stored in the variable specified and also left on the stack as a parameter.

It is possible to specify that a parameter has already been placed in the stack by the user. This is done by specifying an asterisk (*) in place of the parameter. Details of this are covered in *Systems Programming Language.*

These statements transfer control to the procedure named with the parameters properly loaded into the stack. When the called procedure exits (reaches the final END or a RETURN statement), control returns to the statement following the procedure call statement. For example,

```
                    ┌──────────────────────────────→ PROCEDURE BLANKBUF
                    │                                       .
                    │                                       .
                    │                                       .
  BLANKBUF (OUTBUF,36);                                     .
                                             BEGIN
  OUTBUF(5) := ".";                                         .
       ▲                                                    .
       │                                                    .
       └────────────────────────────────── END;
```

Procedure Functioning

Entering and exiting a procedure follows a very machine-dependent sequence in SPL/3000. First, the parameters or their addresses are loaded onto the stack in the order they occur in the parameter list. Then a PCAL instruction is executed. The PCAL instruction addresses through a Segment Transfer Table (in the code segment) and either:

1. Transfers to the start of a procedure in the current code segment, or

2. Uses a Segment Transfer Table entry and a Code Segment Table entry to locate the code segment which contains the procedure and then transfers control to the start of the procedure. (The Code Segment Table is a hardware-referenced, system-wide table which is maintained by the operating system.)

During the execution of the PCAL instruction, a four-word stack marker is placed on the stack and the contents of the Q register are changed to point to a fresh area on the top of the stack. The stack marker preserves the definition of the environment at the time of the PCAL and has the following format:

Calling Parameter Storage	Q – 5	⋮	
	Q – 4		
	Q – 3	X REGISTER	Previous contents of index (X) register (restored on exiting).
Stack Marker	Q – 2	RELATIVE P	PB relative return address.
	Q – 1	STATUS	Previous contents of STATUS (restored on exiting).
	S,Q	DELTA Q	Value to be subtracted from Q to obtain previous Q on exit.
Stack available for *local* data storage		⋮	

The procedure's instructions are executed until an EXIT instruction occurs (generated by final END of a procedure or a RETURN statement). EXIT uses the stack marker to return to the point of the call and reset the environment. The contents of the Q, P, status, and index registers are restored. Parameters can be deleted from the stack or left.

The following sample program shows how the complete process of calling and exiting a procedure occurs. The lines are numbered so that they can be referred to from the explanation. The letters (A, B, C, . . .) mark the chronological path during execution of the program. The explanation will follow this path.

```
        1 |  BEGIN <<SAMPLE PROCEDURE DECLARATION AND CALL>>
        2 |       <<GLOBAL DATA DECLARATION>>
        3 |          LOGICAL ARRAY DATA (0:128);
        4 |       <<PROCEDURE DECLARATION>>
        5 |          PROCEDURE BLANKBUF (BUFFER,COUNT);
        6 |               VALUE COUNT;
        7 |               LOGICAL ARRAY BUFFER;
        8 |               INTEGER COUNT;
        9 |          BEGIN
       10 |          <<LOCAL VARIABLE>>
    D  11 |               LOGICAL BLANKWORD := %20040;
       12 |          <<STATEMENTS>>
       13 |               BUFFER (0) := BLANKWORD;
       14 |     MOVE   BUFFER (1) := BUFFER (0), (COUNT-1);
    E  15 |               END;
    A  16 |  <<START OF MAIN PROGRAM>>
       17 |  <<INVOKE PROCEDURE>>
  B C  18 |       BLANKBUF (DATA,128);
    F  19 |  <<RETURN FROM PROCEDURE>>
       20 |  END <<END OF PROGRAM>>.
```

	Sequence of Execution	Address	Contents	Meaning
A	At the start of the main program (line 16). Note that an artificial stack marker is created for the program so that at the end of the program it can EXIT back to the operating system.	DB + 0	400	Address of 0 element of DATA
			. . .	
		DB + 400_8		DATA array elements
			. . .	
			X	Initial Stack Marker
			P	
			STATUS	
	(DB + 605_8) Q,S		DELTA Q	

6-8

Sequence of Execution	Address	Contents	Meaning

B   Before the procedure call, the parameters are loaded into the stack:

400_8 = address of DATA

$200_8 = 128_{10}$ = COUNT

Address	Contents	Meaning
DB + 0		
Q		Stack Marker
Q + 1	400_8	Address of 0 element DATA
(DB + 607_8) S,Q + 2	200_8	Count parameter (by VALUE)

C   Execution of the PCAL instruction (line 18) causes a stack marker to be loaded onto the stack and the contents of P and Q to be changed. Execution now continues with the body of the procedure (line 9). The index register (X) is not changed; it can be used to pass a parameter.

Address	Contents	Meaning
DB + 0		
		Initial Stack Marker
Q – 5	400_8	Calling Parameters
Q – 4	200_8	
Q – 3	X	
Q – 2	P	Procedure Call Stack Marker
Q – 1	STATUS	
(DB + 613_8) S,Q – 0	DELTA Q = 6	

D   Code within the procedure first allocates and initializes location Q + 1 for the local variable (BLANKWORD, line 11). This is done by loading a constant from the code segment onto the stack. The procedure stores a copy of BLANKWORD through the array parameter (address) in location Q – 5, sets up the addresses and count required by MOVE, and executes a MOVE instruction to fill the DATA array with blanks.

Address	Contents	Meaning
DB + 0	400_8	
DB + 400	blank/blank	DATA array
		Initial Stack Marker
Q – 5	400_8	
Q – 4	200_8	
Q		Procedure Call Stack Marker
Q + 1	blank/blank	BLANKWORD
S – 2	401	destination
S – 1	400	source
S – 0	199	word count

Sequence of Execution	Address	Contents	Meaning

After the MOVE completes, it decrements the stack by three words. This is just before the EXIT.   DB + 0

Address	Contents	Meaning
		Initial Stack Marker
	400_8	Parameters
	200_8	
	X	Stack Marker
	P	
	STATUS	
(DB + 613$_8$) Q	DELTA Q	
S	blank/blank	BLANKWORD

E At this point, an EXIT from the procedure returns control to the main program as follows:

Address	Contents	Meaning
DB + 0	400_8	
DB + 400		DATA array
(DB + 605$_8$) S,Q		Initial Stack Marker

Action	Effect
S is moved back to Q	Procedure data stack deleted.
Q is decremented by delta Q (Q changed from DB + 613$_8$ to DB + 605$_8$)	Re-establish base of calling procedure or program stack.

Reset registers and delete values from stack.
  status register ← status
  P register ← P + PB (return addr.)
  index register ← index
	Previous state of machine registers is restored and the stack marker is deleted from the stack. (S points to last parameter.)

Number of parameter words pushed on the stack for the call are deleted. (This may be overridden to return values in the stack.)	Calling program's stack restored to condition which existed prior to the procedure call.

F At this point, an EXIT from the main program through the initial stack marker returns control to the operating system.

RETURN STATEMENT

The RETURN statement is used to generate additional EXIT points within the body of a procedure. The final END of the procedure also generates an EXIT. The format of a RETURN statement is

RETURN *count* ;

count   is the number of words by which the stack is to be decremented. If *count* is not specified, the decrement value equals the number of words used to pass the actual parameters to the procedure. (Note that *count* is in addition to the stack marker which is always deleted on EXIT.)

The optional count in RETURN allows the programmer to leave some or all of the calling parameters on the stack after exiting. The final END of the procedure declaration does not allow this option. A use of RETURN to save parameters is shown in this example:

```
<<main program>>
      NOW (5, 6.7, 92);
      TEMP := TOS; <<SAVE PARAMETER>>

PROCEDURE NOW (A, B, C);
      VALUE A, B, C;
      INTEGER A, C;
      REAL B;
BEGIN
      .
      .
      .
      RETURN 3;
      .
      .
      .
END;
```

Stack on entry

Q – 7	5	Calling Parameters
Q – 6	6.7	
Q – 5		
Q – 4	92	
Q – 3	Index	Stack Marker
Q – 2	Relative P	
Q – 1	STATUS	
S,Q	Delta Q	

Stack after RETURN 3

S – 0	5	A

EXAMPLE 6-1. DATA COMPRESSION

This program demonstrates the use of a procedure to compress input records by deleting blanks and comments. (All comments are enclosed in quotes.) The procedure uses both the MOVE (byte) WHILE and SCAN UNTIL statements; the parameters for moving and scanning remain in the stack and are modified when necessary. The main program's task is to input the data, call the procedure, and output the results.

Input/Output

```
ENTER A DATA RECORD
12 "RECORD NUMBER" 4/13 "MONTH/YR" W-6475-Q "ITEM NO" 342 "QUANTITY"
124/13W-6475-Q342
CHARACTER COUNT = 17
```

```
ENTER A DATA RECORD
"TEST NO" 465 "SAMPLE LOT" A-241 "HARDNESS" 8
465A-2418
CHARACTER COUNT = 9
ENTER A DATA RECORD
"ITEM NUMBER" 347-572-56B "DEVELOPED" 4/17 "MANUFACTURED BY" SAMCO
347-572-56B4/17SAMCO
CHARACTER COUNT = 20
ENTER A DATA RECORD
"TRANSACTION RECORD" EXCHANGE "ITEM" BICYCLE "PRICE" $56.48 "DATE" 8/23
EXCHANGEBICYCLE$56.488/23
CHARACTER COUNT = 25
```

Listing

```
BEGIN   <<EXAMPLE 6-1. DATA COMPRESSION>>
COMMENT:
        THIS PROGRAM INPUTS AN ASCII RECORD (MAXIMUM LENGTH 72
        CHARACTERS), CALLS A DATA COMPRESSION PROCEDURE, AND OUTPUTS
        THE PACKED RESULT.
        NOTE: "INPUT" AND "OUTPUT" ARE DUMMY PROCEDURES WHICH SIMULATE
        INPUT, OUTPUT, AND CONVERSION - THEY ARE NOT PART OF SPL/3000:
BYTE ARRAY IN(0:72):=73(%15);
BYTE ARRAY OUT(0:72):=73(" ");
BYTE ARRAY HEADMSG(0:19):="ENTER A DATA RECORD ";
BYTE ARRAY LABL(0:17):="CHARACTER COUNT = ";
INTEGER PKD'CNT;
PROCEDURE COMPRESS(INBUF,OUTBUF,COUNT);
   BYTE ARRAY INBUF, OUTBUF;
   INTEGER COUNT;
BEGIN
COMMENT:
        THIS PROCEDURE COMPRESSES INPUT DATA RECORDS BY ELIMINATING
        BLANKS AND ALL COMMENTS ENCLOSED IN QUOTATION MARKS. THE
        INPUT BUFFER CONTAINS A RECORD TERMINATED BY A RETURN. A
        PACKED OUTPUT BUFFER AND A CHARACTER COUNT ARE RETURNED TO THE
        CALLING PROGRAM.;
BYTE POINTER SCHAR=S,
            DCHAR=S-1;
LABEL MOVER,
      EXIT;
        TOS:=@OUTBUF;   <<SET DESTINATION ADDRESS>>
        TOS:=@INBUF;   <<SET SOURCE ADDRESS>>
MOVER:      MOVE *:=* WHILE AN,0;   <<MOVE WHILE NOT SPECIAL>>
        IF SCHAR=%40 THEN   <<BLANK>>
                        BEGIN
                           @SCHAR:=@SCHAR+1;   <<SKIP SOURCE BLANK>>
                           GOTO MOVER;
                        END;
        IF SCHAR=%15 THEN GOTO EXIT;   <<RETURN>>
```

```
        IF SCHAR<>%42 THEN   <<NOT " MARK - VALID SPECIAL CHARACTER>>
                    BEGIN
                      DCHAR:=SCHAR;   <<MOVE CHARACTER>>
                      @DCHAR:=@DCHAR+1;   <<INCR DESTINATION>>
                      @SCHAR:=@SCHAR+1;   <<INCR SOURCE>>
                      GOTO MOVER;
                    END;
        @SCHAR:=@SCHAR+1;   <<SKIP OPENING SOURCE " >>
        SCAN * UNTIL %6442,1;   <<SCAN FOR NEXT SOURCE " >>
        IF NOCARRY THEN   <<QUOTE MARK FOUND>>
                    BEGIN
                      @SCHAR:=@SCHAR+1;   <<SKIP CLOSING SOURCE " >>
                      GOTO MOVER;
                    END;
EXIT:       COUNT:=@DCHAR-@OUTBUF;   <<COMPUTE CHARACTER COUNT>>
END   <<COMPRESS>>;
   <<END OF DECLARATIONS>>
        OUTPUT(HEADMSG);   <<OUTPUT HEADING>>
        INPUT(IN);   <<READ ASCII RECORD>>
        COMPRESS(IN,OUT,PKD'CNT);   <<PROCEDURE CALL>>
        OUTPUT(OUT);   <<OUTPUT COMPRESSED RECORD>>
        OUTPUT(LABL,PKD'CNT);   <<PRINT LABEL AND CHARACTER COUNT>>
END   <<MAIN PROGRAM>>.
```

FUNCTION PROCEDURES

Function procedures are simply those procedures assigned a type (INTEGER, LOGICAL, BYTE, REAL, LONG, DOUBLE) when they are declared. When these procedures are used in an expression, they return a value of the specified type in place of their name. (The result is actually returned in the top of the stack and can thus be used in the rest of the expression.)

It is essential that a function assign a value to its name somewhere within the body of the procedure. Failure to assign a value to the function name causes a zero value to be returned. Therefore, the procedure name should occur on the left side of an assignment statement at some point. For example,

```
BEGIN
    INTEGER NUM := 108, NIX;
    INTEGER PROCEDURE VAL(A,B,C); <<FUNCTION DECLARATION>>
        VALUE A,B,C;
        INTEGER A,B,C;
            VAL := (A+B)*C;
<<MAIN PROGRAM>>
    NIX := NUM/VAL(4,5,6); <<THIS IS EQUIVALENT TO THE STATEMENT:>>
                        <<NIX := NUM/((4+5)*6);
END.
```

The method by which the function value is returned in the stack follows:

| | Stack Upon | |
| Address | Procedure Entry | Meaning |

Function procedures require a place in the stack to return the value set into the function name. The required space is allocated automatically (immediately preceding the function parameters) and is set to zero. Space allocation is large enough to contain the type of return value specified by the function type of the procedure.

DB + 0

108	NUM variable
	NIX variable
· · ·	
	initial stack marker
0	space for function return value
4	value parameters
5	
6	
	procedure call stack marker

S,Q

| | Stack After | |
| Address | Procedure Exit | Meaning |

When the procedure exits, the stack marker and the actual parameter, are deleted from the stack. The value returned, however, is left on the top of stack where it may be used to facilitate expression evaluation.

DB + 0

108	NUM variable
	NIX variable
· · ·	
	initial stack marker

Q

S

| 54 | returned function value |

EXAMPLE 6-2. FACTORIAL COMPUTATION

This program uses a function procedure to supply the factorial values required to compute the number of permutations (different arrangements) of seven items taken three at a time according to the formula:

$$_7P_3 = \frac{7!}{(7-3)!}$$

where

n! = n*(n–1)*(n–2)*.....*(1)
0! = 1

The function procedure in this example has limited general usage since it does not check for overflow and would, therefore, return an incorrect result for factorials of numbers greater than seven.

The input/output example shows the declaration and use of function procedures and the use of a WHILE statement to compute the factorial.

Input/Output

```
PERMUTATIONS OF 7 ITEMS TAKEN 3 AT A TIME=210
```

Listing

```
BEGIN  <<EXAMPLE 6-2. FACTORIAL COMPUTATION>>
COMMENT:
        THIS PROGRAM CALLS A FUNCTION PROCEDURE TO COMPUTE THE
        FACTORIAL VALUES REQUIRED  TO DETERMINE THE  NUMBER OF
        DIFFERENT  ARRANGEMENTS (PERMUTATIONS) OF N THINGS TAKEN
        R AT A TIME.
        PERMUTATIONS= N!/(N-R)!
        NOTE: "INPUT" AND "OUTPUT" ARE DUMMY PROCEDURES WHICH SIMULATE
        INPUT, OUTPUT, AND, CONVERSION - THEY ARE NOT PART OF SPL/3000;
BYTE ARRAY PLABL(0:41):="PERMUTATIONS OF 7 ITEMS ",
                        "TAKEN 3 AT A TIME=";
INTEGER  N:=7,
         R:=3,
         PERMUTATIONS;
INTEGER PROCEDURE FACTORIAL(NUMBR);
   VALUE NUMBR;
   INTEGER NUMBR;
BEGIN
COMMENT:
        THIS FUNCTION PROCEDURE COMPUTES FACTORIAL VALUES.
        N! = N(N-1)(N-2)(N-3)...(1)       WHERE   0! = 1
        NO CHECKING IS MADE FOR POSSIBLE OVERFLOWS RESULTING
        FROM REPEATED INTEGER MULTIPLICATION. ;
```

```
INTEGER PROD:=1;
          WHILE NUMBR>0 DO   <<MULTIPLICATION LOOP>>
          BEGIN
             PROD:=PROD*NUMBR;
             NUMBR:=NUMBR-1;
          END;
          FACTORIAL:=PROD;   <<RETURN THE FACTORIAL VALUE>>
END  <<FACTORIAL>>;
<<END OF DECLARATIONS>>
          PERMUTATIONS:=FACTORIAL(N)/FACTORIAL(N-R);
          OUTPUT(PLABL,PERMUTATIONS);   <<OUTPUT ANSWER>>
END  <<MAIN PROGRAM>>.
```

RECURSIVE PROCEDURES

A procedure is recursive if it can be called at any time in a program even though an indefinite number of previous calls to the same procedure are still active (have not exited yet). Recursive procedures are capable of calling themselves. Recursion pertains to operations which are inherently repetitive. The result of each operation is usually dependent upon the result of the previous repetition. The following procedure is recursive:

INTEGER PROCEDURE FACTORIAL (I);

   VALUE I; INTEGER I;

      FACTORIAL := IF I = 0 THEN 1 ELSE I * FACTORIAL(I - 1);

This function procedure calculates the factorial of an integer value (factorial $n = n! = n*(n - 1)*$ $(n - 2)* \ldots *(1)$) by subtracting one from the integer and calling itself again until the integer parameter equals one. Then it returns to itself repeatedly (carrying out the multiplications on the way) until the final result has been found.

This factorial procedure is an example of direct recursion. Recursion can also occur indirectly: procedure A calls procedure B which then calls A again. Eventually A exits back to B which exits back to A. For example,

Recursion provides a simple and elegant means of handling indefinite nesting. One of the most common occurrences of indefinite nesting is in the construction of compilers for programming languages. For example, expressions within parentheses can occur within other expressions in most programming languages. Evaluation of parenthetical expressions can be easily handled using recursive techniques. When the expression analysis procedure encounters a left parenthesis it saves the accumulated partial result and calls itself recursively to handle the expression within the parentheses. When the inner expression has been evaluated, the procedure exits back to itself and continues evaluating the outer expression.

Consider the following construction from a hypothetical programming language and assume that *expr* is the procedure that evaluates expressions (down arrows indicate procedure calls, up arrows indicate procedure exits):

Recursive Levels

```
0        ( 2 * 3 * ( 2 + 3 + ( 4 / 2)      )      )   ⇒   42
           ↓                      ↑      ↑    ↑
1        [2*3 = 6] *              |   [6*7 = 42]
             ↓                    |        ↑
2                    [2+3 = 5] +  |   [5+2 = 7]
                         ↓            ↑
3                            [4/2 = 2]
```

Compilers which are written to analyze programs in this manner are called **recursive-descent compilers**. They allow very powerful programming languages to be developed because of their inherent generality and nesting capability. The compilers are written modularly with analysis procedures for each portion of the language syntax (declarations, array declarations, statements, FOR statements, expressions, etc.). Each procedure attempts to recognize a single type of syntactic entity. A procedure is called when it is known that the next item in the program must be an entity of that type (for example, a condition clause must follow the IF in an IF statement). The SPL/3000 compiler is a recursive-descent compiler.

Re-entrant Code and Recursion

Re-entrant code and recursion are related topics (re-entrant code is required to implement recursion efficiently). Code is re-entrant if it can be used at any time, even if it is entered before a previous use of it has completed. When code is re-entrant it can be shared among many users, thus eliminating the need for duplicate copies in memory. The simplest way to guarantee that code will be re-entrant is not to modify any of the locations in the code. This can be done by separating the code from the data. In the HP 3000 this is automatic, since code and data are separated into different segments and no instructions modifying the code segment are available to the user. All information that must be modified is kept in the user's data segment and each user executing shared code has his own data segment.

However, more than re-entrant code is required for recursion. Recursion also requires that each call upon a procedure allocate fresh temporary storage and retain a clear trail back to all previous calls *within the same program*. In HP 3000 these requirements are met by the PCAL and EXIT instructions. PCAL creates fresh local storage when it establishes the new Q register setting and stack marker. The stack marker saves the environment of the previous call and is used by the EXIT instruction to restore the previous environment when the current call is completed. Thus all procedures in SPL/3000 are inherently recursive with no special effort required on the part of the programmer.

Table 6-1 summarizes the differences between re-entrant and recursive code:

Table 6-1. Re-entrant and Recursive Code Differences

Requires	Provides
Re-entrant Code	
Non-modified code	Code which can be shared by different programs at the same time. (Within a program, only one call to the procedure can occur before the matching exit.)
Separate data storage for each program that calls the code. All information pertinent to the call is saved in this data area (temporaries, return address, etc.).	
Recursive Code	
Re-entrant code (for efficiency)	Code which can be called at any point within a program, even when previous calls from the same program are still active.
Fresh allocation of local temporary storage for each call upon the code. Previously allocated storage must be retained unaltered.	
Clear trail back to all previous active procedure calls within the program.	

Option Forward

Indirect recursion presents an interesting syntactic requirement in SPL/3000, since everything must be declared before it is used. For example, if procedure A calls procedure B which calls procedure A again, B must be declared before A (since A calls B). But, on the other hand, A must be declared before B (since B calls A). The dilemma is resolved by the option forward construction in a dummy procedure declaration.

```
              main
First call to  (     )   Exit
                 A
First call to  (     )   Exit
                 B
Second call to (     )   Exit
                 A
```

The procedure head is declared first, followed by the words OPTION FORWARD. The forward option indicates that the complete definition of the procedure (the head plus the body) will occur later. After all such procedures have been documented with "dummy declarations" and forward options, the complete declarations of these procedures can follow in any order. Since all parameters have been declared, the compiler has enough information on the procedures to allow other procedures to call them. For example,

6-18

PROCEDURE PROC1; OPTION FORWARD; <<dummy declaration>>
PROCEDURE PROC2; OPTION FORWARD; <<dummy declaration>>

PROCEDURE PROC1; IF X = (Y := (Y + 1) THEN PROC2; <<complete declaration>>
PROCEDURE PROC2; IF X = (Z := (Z + 1) THEN PROC1; <<complete declaration>>

EXAMPLE 6-3. BINARY TO DECIMAL CONVERSION

Recursion is very useful in operating system functions and language compilers. However, these are complex examples, not appropriate in a textbook. Recursion can be simply used in the conversion of numbers from octal to decimal. The method used divides the octal number by 12_8 (decimal 10) repeatedly until the result is 0; the remainders read in reverse order are the decimal digits of the converted number:

$$526_8 = ?_{10}$$

$$
\begin{array}{r}
42_8 \\
12_8 \overline{)526_8} \\
524_8 \\
\hline
2
\end{array}
\quad 2
$$

342_{10}

$$
\begin{array}{r}
3 \\
12_8 \overline{)42_8} \\
36_8 \\
\hline
4
\end{array}
\quad 4
$$

$$
\begin{array}{r}
0 \\
12_8 \overline{)3_8} \\
0 \\
\hline
3
\end{array}
\quad 3
$$

This program calls a procedure to convert a binary number (16-bit positive integer) into decimal digits and output the results. The procedure divides a binary number by 10_{10}. The remainder is converted to an ASCII numeric character and is saved in the stack. If the quotient is not equal to zero, the procedure calls itself with the quotient as a parameter. The ASCII characters (remainders) are saved in the stack in order of increasing significance—least significant first, most significant last. When the quotient finally equals zero, the procedure outputs the last computed remainder (the most significant digit) and exits to itself. Each time the procedure exits, another digit is output. When the entire number has been output, the program terminates. The following diagram shows the levels of recursion involved in coverting 526 (octal) to decimal:

Levels of Recursion

Main Program   Main Program

1st call $\dfrac{526_8}{12_8} = 42_8$ rem = 2   OUTPUT 2 ⌐EXIT

2nd call $\dfrac{42_8}{12_8} = 3_8$ rem = 4   OUTPUT 4 ⌐EXIT

3rd call $\dfrac{3}{12_8} = 0$ rem = 3 .... OUTPUT 3 ⌐EXIT

The main program is set up to convert a specific number (526_8) but it could easily be generalized to handle any positive integer.

Output

342

Listing

```
BEGIN  <<EXAMPLE 6-3. BINARY TO DECIMAL CONVERSION>>
COMMENT:
        THE MAIN PROGRAM CALLS A RECURSIVE PROCEDURE TO CONVERT A
        BINARY NUMBER TO DECIMAL AND OUTPUT THE RESULT.
        NOTE: "INPUT" AND "OUTPUT" ARE DUMMY PROCEDURES WHICH SIMULATE
        INPUT, OUTPUT, AND CONVERSION - THEY ARE NOT PART OF SPL/3000;
LOGICAL OCTNO:=%526;
PROCEDURE CONDEC(NUMBER);
   VALUE NUMBER;
   LOGICAL NUMBER;
BEGIN
COMMENT:
        BINARY TO DECIMAL CONVERSION PROCEDURE
        THE OCTAL NUMBER IS DIVIDED BY 10 AND THE REMAINDER IS MADE
        INTO AN ASCII CHARACTER. IF THE QUOTIENT IS NOT ZERO, THE
        PROCEDURE CALLS ITSELF. BEFORE EACH EXIT, THE LAST COMPUTED
        CHARACTER IS OUTPUT.;
BYTE CHAR;
LOGICAL QUOT;
        QUOT:=NUMBER/10;   <<COMPUTE QUOTIENT>>
        CHAR:=%60+NUMBER MOD 10;   <<COMPUTE REMAINDER IN ASCII>>
        IF QUOT <> 0 THEN CONDEC(QUOT);   <<FINISHED?>>
        OUTPUT(CHAR);   <<OUTPUT LAST DIGIT COMPUTED>>
END;  <<CONDEC>>
<<END OF DECLARATIONS>>
        CONDEC(OCTNO);   <<INVOKE PROCEDURE>>
END   <<MAIN PROGRAM>>.
```


INTRINSICS

Intrinsics are compiled procedures that are supplied to users of HP 3000 as part of the Multi-programming Executive (MPE/3000). These procedures provide file access and utility functions. (Intrinsics are described in the MPE/3000 reference manual.)

The programmer could declare his intrinsics procedures as ordinary procedures by adding the EXTERNAL option to the head of each procedure and deleting the body. EXTERNAL means that the procedure body (code) is linked to the main program by the operating system after compilation. Writing out the complete head for some intrinsics can be very time-consuming, so a shortcut is provided in SPL/3000. SPL/3000 provides simple interface to intrinsics because SPL/3000 itself provides no construct for input and output. Any system-known intrinsics (such as those described in the MPE/3000 reference manual) can be declared in a user program by using the INTRINSIC declaration:

INTRINSIC *intrinsic procedure list;*

each intrinsic is identified by its name.

INTRINSIC declarations occur in the procedure group of the main program or the data group of a procedure declaration. The programmer must actually know the declaration format for each intrinsic, since he needs to know the type of the procedure and the number/type of parameters in order to call the intrinsic correctly.

Since many procedures (especially intrinsics) have a large number of parameters that may not all be required for every call, a method has been provided for specifying a variable number of actual parameters. This is done by including OPTION VARIABLE in the option part of the procedure head. The compiler generates code (when the procedure is called) to provide the procedure with a parameter bit mask in location $Q - 4$ (also $Q - 5$ if more than 16 parameters). If an actual parameter is missing; for example, NOW(A,,C); the corresponding bit in the mask is set to zero (0). The correspondence is from right to left (the rightmost bit—bit 15—corresponds to the rightmost parameter). In the procedure call, the occurrence of a right parenthesis before the parameter list is filled implies that the rest of the parameters are missing. When the procedure is entered, it is the responsibility of the procedure to examine the bit mask. Parameters always occur in the same $Q-$ addresses; missing parameters have useless data in their locations.

SUBROUTINES

In SPL/3000, the **subroutine** is a simpler and less powerful form of subprogram than the procedure. Subroutines can have parameters, can be typed functions, and can be called recursively.

However, subroutines are implemented with different hardware instructions (SCAL and SXIT). SCAL does not provide a four-word stack marker as PCAL does; this accounts for all of the advantages/disadvantages of subroutines:

- Values in the Q and index registers remain unchanged.

- A P relative return address is placed on the top of stack.

- All parameters are referenced relative to the S register and labels cannot be passed as parameters.

- Subroutines cannot have local variables.

- Subroutines must be located in the same segment as the caller since SCAL and SXIT do not bridge segment boundaries.

- Subroutines can be entered and exited faster than procedures since there is much less work for the instructions to do. (The machine environment is neither saved nor restored).

- Subroutines can be declared within procedures (while procedures cannot) and can reference procedure-local variables (since Q is not changed when a subroutine is called).

- Subroutines, like procedures, can address all global variables since DB is not effected by calling them.

Declaration of Subroutines

Subroutines can be declared in a main program (global subroutines) or within a procedure (local subroutines).

Global subroutines can be called *only* within the main program (not from within procedures since procedures need not be in the same segment as the main program). Global subroutine declaration must appear after procedure declarations:

BEGIN

```
┌─────────────────────┐
│     data group      │
└─────────────────────┘
┌─────────────────────┐
│      intrinsics     │
└─────────────────────┘
```

and

```
┌─────────────────────┐
│     procedures      │
└─────────────────────┘
┌─────────────────────┐
│     subroutines     │
└─────────────────────┘
┌─────────────────────┐
│      main body      │
└─────────────────────┘
```

END.

Local subroutines can be called only from the procedure in which they are declared. They are declared in the body of the procedure, after any local data declarations, but before the statements of the body:

```
┌─────────────────────┐
│    procedure head   │
└─────────────────────┘
```

BEGIN

```
┌─────────────────────┐
│  data declarations  │
└─────────────────────┘
┌─────────────────────┐
│   subroutine decl.  │
└─────────────────────┘
┌─────────────────────┐
│     statements      │
└─────────────────────┘
```

END;

The declaration format of a subroutine is identical to that of a procedure, except that there is no option part and no local data group:

$$
head \begin{cases} type \\ \text{SUBROUTINE} \\ name \\ formal\ parameters \\ value\ part \\ specification\ part \end{cases}
$$

$$
body \begin{cases} statement \text{ (possibly compound)} \end{cases}
$$

For example,

```
INTEGER SUBROUTINE S(A,B,C);
    VALUE A,B,C;
    INTEGER A,B,C;
    S := (A ^ 2) + (B * C);
```

Invoking Subroutines

Subroutines are invoked by using their identifier in a subroutine call statement and replacing the formal parameters with actual parameters:

identifier (parameter list) ;

Parameters can be stacked (asterisk—*) just as with procedures.

Function subroutines are invoked by using them within an expression:

NIX := S(4,5,6) + S(100,20,1);

Here is a complete program showing the format of subroutine declarations and invocations:

```
            BEGIN <<USE A SUBROUTINE TO SET AN ARRAY TO ZERO>>
                INTEGER ARRAY ADATA(0:50);
                INTEGER I; <<INDEX FOR USE IN SUBROUTINE>>
            SUBROUTINE ZERO(ARRY,HISUB);
                VALUE HISUB;
                INTEGER HISUB;
                INTEGER ARRAY ARRY;
            BEGIN
                I := 0; <<SET INITIAL VALUE INTO SUBSCRIPT>>
                WHILE I <= HISUB DO
                    BEGIN
                        ARRY(I) := 0;
                        I := I + 1;
                    END;
            END <<ZERO>>;

            <<END OF DECLARATIONS>>
                ZERO(ADATA,50); <<CALL SUBROUTINE>>
            END <<MAIN PROGRAM>>.
```

subroutine declaration

subroutine call

Subroutine Functioning

The characteristics of subroutines are determined by the functioning of the SCAL and SXIT instructions, which work in this manner:

SCAL

1. Upon invocation of a subroutine, the parameters are loaded onto the stack and an SCAL is executed.

2. SCAL loads P + 1 (the return address) onto the stack and branches to a relative address within the current code segment.

3. All parameters are referenced using S relative addressing. Since the top of stack changes constantly, the S relative addresses of the parameters also change constantly.

SXIT

1. When the SXIT is executed, the current top of stack value is used as the P relative return address.

2. Note that since S is constantly changing, it is possible for the subroutine to use an incorrect return address if the subroutine has explicitly modified the stack.

The process above can be seen in the following example. Assume we have a subroutine SW which receives two integer values, exchanges them, and exits, leaving them on the stack.

```
BEGIN
    INTEGER A,B;
    SUBROUTINE SW(Y,Z);
        VALUE Y,Z;
        INTEGER Y,Z;
        BEGIN
            ASSEMBLE (
                LOAD S – 2 <<Y>> ;
                LOAD S – 2 <<Z>> ;
                STOR S – 4 <<Y>> ;
                STOR S – 2 <<Z>> ;
                RETURN 0;          );
        END;
    <<MAIN PROGRAM>>
        SW(25,10);
        A := TOS;
        B := TOS;
END.
```

NOTE: *All parameters in the subroutine stack are S relative addresses. Since the current top of stack changes constantly, the addresses of the variables change also.*

Stack
On Subroutine Entry

S – 2	25	Y
S – 1	10	Z
S – 0	P	Return Address

After first instruction (LOAD S – 2 <<Y>>)

S – 3	25	Y
S – 2	10	Z
S – 1	P	Return Address
S – 0	25	

After second instruction (LOAD S – 2 <<Z>>)

S – 4	25	Y
S – 3	10	Z
S – 2	P	Return Address
S – 1	25	
S – 0	10	LOAD S – 2

After third instruction (STOR S – 4 <<Y>>)

S – 3	10	Y
S – 2	10	Z
S – 1	P	Return Address
S – 0	25	

After fourth instruction (STOR S – 2 <<Z>>)

S – 2	10	Y
S – 1	25	Z
S – 0	P	Return Address

After RETURN 0

S – 1	10
S – 0	25

After (A := TOS) and (B := TOS)

DB + 0	25	A
DB + 1	10	B
S – 0		

The natures of the SCAL and SXIT instructions make subroutines much less flexible and power-full than procedures. Subroutines cannot have local variables; the only variables available to a subroutine are parameters and DB relative and Q relative (local subroutines only) variables. Also, the user must not explicitly modify the stack within a subroutine without immediately correcting for any changes. All subsequent parameters addressing may be incorrect and S may not point to the return address when SXIT is executed.

EXAMPLE 6-4. MATRIX MANAGEMENT

This program contains a subroutine that maps a user-specified two-dimensional subscript pair into a linear integer array. The user can request the subroutine to read or store a single value on each call. The subroutine checks for incompatible parameters. The main program provides FOR LOOP indexing to fill a 3 by 3 identity matrix. This example illustrates the declaration, invocation, and simple nature of most subroutines. This example manipulates data in memory and has no input/output.

Listing

```
BEGIN   <<EXAMPLE 6-4. MATRIX MANAGEMENT>>
COMMENT:
        THE MAIN PROGRAM USES THE SUBROUTINE "MATRIX" TO INITIALIZE
        A 3X3 IDENTITY MATRIX.
                                1 0 0
                                0 1 0
                                0 0 1
        NOTE: "INPUT" AND "OUTPUT" ARE DUMMY PROCEDURES WHICH SIMULATE
        INPUT, OUTPUT, AND CONVERSION - THEY ARE NOT PART OF SPL/3000;
INTEGER ARRAY NUMBRS(0:8);
BYTE ARRAY ERR(0:5):="ERROR ";
INTEGER DATA,I,J,SUB,CONTL:=1;
LABEL EXIT;
SUBROUTINE MATRIX(DATA'ARRAY,MAXROW,MAXCOL,IOFLAG,NUMBER,ROW,COL);
   VALUE MAXROW,MAXCOL,ROW,COL;
   INTEGER ARRAY DATA'ARRAY;
   INTEGER MAXROW,MAXCOL,IOFLAG,NUMBER,ROW,COL;
BEGIN
COMMENT:
        ARRAY MANAGEMENT SUBROUTINE
        ALLOWS A LINEAR ARRAY TO BE ACCESSED AS A 2 DIMENSIONAL ARRAY
        USING ROW AND COLUMN SUBSCRIPTS. ALL SUBSCRIPTS MUST BE
        NON-NEGATIVE. THE CALLING PROGRAM MUST PROVIDE THE DATA ARRAY
        NAME, MAXIMUM VALUES FOR ROW AND COLUMN SUBSCRIPTS, I/O CONTROL
        FLAG, NUMBER BEING PROCESSED, AND THE ROW AND COLUMN FOR THE
        CURRENT OPERATION. (MINIMUM ROW AND COLUMN SUBSCRIPTS ARE
        ASSUMED TO BE ZERO).
                IOFLAG <0  RETRIEVE THE VALUE
                IOFLAG >0  STORE THE VALUE
                IOFLAG =0  RETURNED TO INDICATE ERROR CONDITION
                                                                ;
        IF NOT(0<=ROW<=MAXROW) OR NOT(0<=COL<=MAXCOL) OR (IOFLAG=0)
           THEN  <<REQUEST INVALID>>
              BEGIN
                 IOFLAG:=0;  <<SET ERROR RETURN>>
                 RETURN;
              END;
        SUB:=ROW*(MAXROW+1)+COL;  <<COMPUTE SUBSCR IN LINEAR ARRAY>>
        IF IOFLAG<0 THEN NUMBER:=DATA'ARRAY(SUB)  <<RETRIEVE VALUE>>
                    ELSE DATA'ARRAY(SUB):=NUMBER;  <<STORE VALUE>>
END;  <<MATRIX SUBROUTINE>>
<<END OF DECLARATIONS>>
```

```
FOR I:=0 UNTIL 2 DO  <<INDEX ROWS>>
  FOR J:=0 UNTIL 2 DO  <<INDEX COLUMNS>>
    BEGIN
      DATA:=IF I=J THEN 1 ELSE 0;  <<SELECT VALUE>>
      MATRIX(NUMBRS,2,2,CONTL,DATA,I,J);  <<STORE A VALUE>>
      IF CONTL=0 THEN  <<INVALID REQUEST>>
                   BEGIN
                     OUTPUT(ERR);  <<OUTPUT ERROR MESSAGE>>
                     GOTO EXIT;  <<TERMINATE>>
                   END;
    END;
EXIT: END  <<MAIN PROGRAM>>.
```

1. Complete the diagram below by drawing the stack marker that would be created by a procedure call instruction. Assume that this procedure passes three single word parameters that are shown on the stack and that the procedure call instruction is about to be executed.

DL = 34630   PB = 35000
DB = 34630   P = 35123 (PCAL)
Q = 40200   PL = 357621   (40200) Q →
S = 40206   X = 000001
Z = 42316   STATUS = 004076

(40206) S →

> PREVIOUS STACK MARKER

> LOCAL VARIABLES

> PARAMETERS (3 WORDS)

2. Complete the diagram below by drawing the stack as it would appear after a procedure exit operation. Values for all registers are given as well as values for the stack marker. Assume in this case an EXIT (1) operation is about to be executed. This instruction will leave 2 single word values on the stack. Show the new values of all registers affected by the EXIT (1) instruction.

DL = 30000   PB = 002000
DB = 30000   P = 002216
Q = 31011   PL = 002500
S = 31015   X = 177777
Z = 32000   STATUS = 004437

> PREVIOUS STACK MARKER

> PARAMETERS (3 WORDS)

| 000001 |
| 000140 |
| 002037 |
(31011) Q → | 000011 |

STACK MARKER

(31015) S →

> LOCAL VARIABLES AND TEMPORARIES

3. The following procedure declarations contain at least one error. Make the corrections required by re-writing or re-arranging the declarations.

a) PROCEDURE SAM(X, Y, Z);

     VALUE X;

     INTEGER Y,Z;

       BEGIN

          Z := X + Y;

       END;

b) PROCEDURE FIXIT (A,B,X);

     VALUE A,B;

     OPTION FORWARD;

     REAL A,B,X;

c) REAL PROCEDURE BIGGEST (X,Y);

     VALUE X, Y REAL X, Y;

     BIGGEST := IF X < Y THEN Y ELSE X;

4. Write a simple procedure to move words. Three parameters will be supplied:

     FROM — a logical array (by reference)

     DEST — a logical array (by reference)

    COUNT — a signed integer (by value)

5. The declarations below illustrate a main program and several subprograms (procedures and subroutines). Using the table provided indicate with a checkmark where each variable may be legally referenced. Then indicate the scope of addressability for each variable listed (GLOBAL/LOCAL).

a) BEGIN <<MAIN PROGRAM>>
     INTEGER X; REAL Y;
     .
     .
     .
     PROCEDURE ALPHA;
       BEGIN
         INTEGER A; REAL B;
         .
         .
         .
       END;
     PROCEDURE BETA;
       BEGIN
         INTEGER I, REAL J, LOGICAL K;
         .
         .
       END;
     .
     .
     .
     .
END <<MAIN PROGRAM>>

VARIABLE	MAIN	PROCEDURE ALPHA	PROCEDURE BETA	TYPE
X				
Y				
A				
B				
I				
J				
K				

b)
```
┌ BEGIN<<MAIN PROGRAM>>
│     REAL A,B;
│        .
│        .
│        .
│  ┌ PROCEDURE THING;
│  │     BEGIN
│  │         INTEGER I,J;
│  │            .
│  │            .
│  │            .
│  │                 SUBROUTINE SAM;
│  │                     BEGIN
│  │                        .
│  │                        .
│  │                     END;
│  └──── END;
│  ┌ SUBROUTINE FIXIT;
│  │     BEGIN
│  │        .
│  │        .
│  │        .
│  └──── END;
└ END <<MAIN PROGRAM>>.
```

VARIABLE	MAIN	PROCEDURE THING	SUBROUTINE SAM	SUBROUTINE FIXIT	TYPE
A					
B					
I					
J					

6. A procedure whose purpose is to plot X – Y points must be able to scale floating-point values. Write a subroutine for this purpose; it receives two real parameters by value (a floating-point number and a scale factor). Multiply the floating-point number by the scale factor, convert the result to a 16 bit integer (use rounding), and return the result to the calling procedure in the hardware index register. (Assume the calling procedure contains the local declaration INTEGER X = X;.)

7. Write a REAL function procedure that will convert degrees to radians. The function will accept a single real parameter by value (angle in degrees) and return the result (Radians) in the function name.

Angle in Radians = Angle in Degrees * .017453

8. Each of the three subroutines shown contains an error. Circle the statements in error and explain why they are incorrect.

a) REAL SUBROUTINE ANS (A,B);
        VALUE A,B; REAL A,B;
        BEGIN
            INTEGER X = X;
            IF X > 0 THEN ANS := A + B
                ELSE ANS := A − B;
        END;

b) SUBROUTINE EVALUATE (DATA, GOODRANGE);
    REAL DATA, GOODRANGE;
    OPTION EXTERNAL;

c) SUBROUTINE MOVER (SOURCE, DEST, COUNT);
    BEGIN
    ARRAY SOURCE, DEST;
    INTEGER COUNT;
        MOVE DEST := SOURCE, (COUNT);
    END;

9. Examine the code shown below. Determine the numeric value of RESULT at point Ⓐ and at point Ⓑ .

```
BEGIN
     INTEGER A := 1,
             B := 2,
             C := 3,
             RESULT;
     PROCEDURE SUMUP;
         BEGIN
             INTEGER A := 10,
                     B := 20,
                     C := 30;
             RESULT := A + B + C;
         END;
         RESULT := A + B + C;   ←—— Ⓐ
         SUMUP;   ←——————— Ⓑ
END.
```

10. What value is returned by the function procedure shown below? Assume A = 3, B = 4, C = 5.

```
    LOGICAL PROCEDURE VERIFY (A,B,C);
    INTEGER A,B,C;
    BEGIN
        LOGICAL RESULT;
        RESULT := IF A <= B <= C THEN 5 ELSE 4;
    END;
```

11. Rewrite the procedure shown to remove any existing errors.

```
    PROCEDURE COMPUTE (X,Y);
    BEGIN
        VALUE X;
        INTEGER X,Y;
        Y := X MOD 8;
    END;
```

SECTION VII
Data Access Concepts

This section discusses these SPL/3000 data access concepts:

- Based and Composite Integers
- Double and Long Data Types
- Base Register Reference in declarations
- Indexed Identifier Reference in declarations
- Variable Reference in declarations
- Own variables
- Advanced array declarations
- Advanced pointer declaration
- Advanced variable declaration
- Explicit stack access

SPECIAL INTEGER CONSTANTS

There are two more convenient forms of representing integer constants in SPL/3000: based integers, and composite integers.

Based Integer Constants

SPL/3000 allows the programmer to use any number base from 2 through 16 in his constants. Decimal numbers (base 10) are represented without a special symbol (123, –567, 79). Octal numbers (base 8) are represented by putting a % before the digits (%234, %444, %77, %102).

All other bases are represented in the following manner:

sign % (base) number

sign	is +, –. (If the sign is omitted the constant is assumed to be positive.)
base	is the number's base value (2 through 16).
number	is any legal number in that base (A, B, C, D, E, F are used to represent 10, 11, 12, 13, 14, 15, in number systems with bases greater than 10).

If the number is greater than 16 bits, it is truncated on the left. For example,

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
%(16) 12A7	0	0	0	1	0	0	1	0	1	0	1	0	0	1	1	1
%(9) 238	0	0	0	0	0	0	0	0	1	1	0	0	0	1	0	1
%(4) 230	0	0	0	0	0	0	0	0	0	0	1	0	1	1	0	0
%(16) ABCD	1	0	1	0	1	0	1	1	1	1	0	0	1	1	0	1
−%(8) 473	1	1	1	1	1	1	1	0	1	1	0	0	0	1	0	1

Composite Integer Constants

Composite integers are composed of bit fields; these bit fields are concatenated from left to right and the result is right-justified.

A composite integer is represented as follows:

[*bit field list*]

    bit field list   is a list of *bit fields* separated by commas.

        bit field   is of the form *n/i*

            n   is the number of bits in the field (an integer constant),

            i   is an integer constant (decimal, based, or composite) to be used in the field.

NOTE: Composite integer definitions can be nested.

The *integer* to the right of the slash (/) is truncated on the left to the size specified by *n* (number of bits).

The following diagram shows how a composite integer is constructed from right-to-left:

0 fill during right justification of the connected bit fields.

Composite integers are very convenient for constructing constants from several small bit fields. Bit field lengths can be equated to symbolic identifiers to document the use (or meaning) of each subfield. One of the most common needs for this capability is in the construction of special floating-point constants because real numbers are composed of fields:

S = sign
E = exponent
F = fraction

When composite integers are used as type real, they are not floated. The example below shows composite integers used to build the largest and smallest positive real numbers:

```
EQUATE                        <<BIT FIELDS OF A REAL NUMBER>>
      SIGN = 1,               <<1 SIGN BIT>>
      EXPONENT = 9,           <<9 EXPONENT BITS>>
      FRACTION = 22;          <<22 FRACTION BITS>>

REAL LARGEST := [SIGN/0, EXPONENT/%777, FRACTION/17777777D] E,
     SMALLEST := [SIGN/0, EXPONENT/0, FRACTION/1] E,
         ZERO := [SIGN/0, EXPONENT/0, FRACTION/0] E;
```

DOUBLE AND LONG DATA TYPES

In addition to BYTE, INTEGER, LOGICAL, and REAL data types, SPL/3000 has two other data types:

    DOUBLE — double-word integers

    LONG — quadruple word floating-point numbers**

Both data types can be used wherever a type is required, for example, in constants, simple variables, expressions, pointers, arrays, typed procedures, typed subroutines, and parameters. The programmer should be aware that hardware to do LONG arithmetic is optional on pre-Series II systems. If the hardware is not installed, use of LONG arithmetic results in calls to system procedures for each operation. All Series II systems have the firmware to do LONG arithmetic.

Type Double

Double integers are 32-bit signed integers which can range in magnitude from –2,147,483,648 to +2,147,483,647.

Any integer constant followed by a D is treated as a 32-bit signed integer:

```
123 D
–7779 D
1234567890 D
[2/211,15[31%(2)101,12/0],10/123] D
%(2)01111011110111101111 D
%(16) 1023ABCD D
```

In declarations, DOUBLE can be used wherever other types can be used:

```
DOUBLE A,B,C,D; <<SIMPLE VARIABLE>>
DOUBLE ARRAY NOW (0:10);
DOUBLE POINTER NAW;
DOUBLE PROCEDURE NEW (A,B);DOUBLE A,B;
      NEW := A + B;
```

**Triple word floating-point numbers in pre-Series II systems.

The SPL/3000 programmer cannot mix double integers with other data types. Therefore, type transfer functions are used to create type compatibility within expressions where variable types must be mixed:

REAL (D)   <<CONVERT DOUBLE TO REAL>>
DOUBLE (I)   <<CONVERT INTEGER TO DOUBLE>>
DOUBLE (L)   <<CONVERT LOGICAL TO DOUBLE>>
DOUBLE (B)   <<CONVERT BYTE TO DOUBLE—RESULT IS ALWAYS POSITIVE>>
INTEGER (D)   <<CONVERT DOUBLE TO INTEGER>>
FIXT (R)   <<TRUNCATE REAL TO DOUBLE>>
FIXR (R)   <<ROUND REAL TO DOUBLE>>

The only arithmetic operations defined for double integers are bit shifts, addition and subtraction.

Type Long

Long floating-point numbers are 64-bit numbers with 32 more bits of fraction than REAL numbers. Long numbers can range from -10^{77} to $+10^{77}$ ($8.6362 \times 10^{-78} <= |N| <= 1.1579 \times 10^{77}$) with 16.5 digits of decimal accuracy (versus 6.9 digits for REAL).**

LONG constants are indicated by the letter L and have this format:

fractional part L *power*

fractional part   is a signed or unsigned decimal fraction (with decimal point) or decimal integer (no decimal point).

power   is a signed or unsigned decimal integer which specifies the power of ten to be multiplied by the fractional part.

For example:

137 L 20
123456789 L–2
–100 L 7
–55.678329 L 1

In declarations, LONG can be used anywhere that other types can be used:

LONG   NEW := 1.2345678910 L 23,
       OLD;
LONG   ARRAY THEN (5:15); <<4 words per element>>
LONG   POINTER SAW;
LONG   PROCEDURE FIND(N,M);
       LONG N,M;
       FIND := (N*M) + (N/M);

**For pre-Series II systems: Long floating-point numbers are 48-bit numbers with 16 more bits of fraction than REAL numbers. Long numbers can range from -10^{77} to $+10^{77}$ ($8.6362 \times 10^{-78} <= |N| <= 1.1579 \times 10^{77}$) with 11.7 digits of decimal accuracy (versus 6.9 digits for REAL).

As with DOUBLE, type mixing is not allowed in expressions, so type transfer functions are provided:

LONG (R) <<CONVERT REAL TO LONG>>

LONG (D) <<CONVERT DOUBLE TO LONG>>

REAL (L) <<CONVERT LONG TO REAL>>

An exception is that a long value can be exponentiated to an integer power without type transfer. For example (assume LONG L):

L := L ^ 2;

DECLARATIONS

Many of the powerful features of SPL/3000 are extensions to the declaration facility. In Section IV we saw how simple variables of type byte, logical, or integer could be equivalenced to the index register by following the identifier declaration with = X.

LOGICAL INDX = X;

BYTE BINDX = X;

INTEGER X = X;

This same mechanism is used to provide other types of address equivalencing or register-relative references in declarations to allow the programmer to control exactly where each data item is to be located. Options are employed to re-use, relabel, and retype storage which is already allocated. One location or block of locations can be associated with several identifiers (of differing data types) in the same program. For example, an array can be filled as words by accessing it with an integer array identifier and manipulated as bytes by referencing it as a byte array.

The general mechanisms (base register reference, indexed identifier reference, variable reference, and OWN variables) will be discussed first. Then we will see how they apply in practice to the declaration of array, pointers, and simple variables.

Base Register Reference

To assign data items explicitly to register-relative locations (when declaring them) follow the identifier with a register name and an offset:

identifier = DB + *usi*255

identifier = Q + *usi*127

identifier = Q – *usi*63

identifier = S – *usi*63

usi   means an unsigned integer that is less than or equal to the number that follows.

*usi*255 (0 to 255) *usi*127 (0 to 127) *usi*63 (0 to 63)

Pointers, and simple variables can all be address referenced by using base register reference in declarations. (Arrays can also be register referenced; see the SPL/3000 reference manual.) No new space is allocated when this is done—the space may already be allocated to some other identifiers. The result is to access the specific location(s) assigned to an identifier when that identifier is used. For example,

INTEGER A = DB + 30;

Indexed Identifier Reference

In SPL/3000 the programmer can specify that the zero element of a new array is to coincide with some previously-defined array or pointer element, instead of having new space allocated for it. This is accomplished in the declaration by following the identifier with an indexed reference to an array or pointer:

identifier = ident (index)

(index)   is optional.

Thus, two arrays of different type can overlap in the actual data locations they occupy. Since no new space is allocated when using indexed identifier reference, initialization is not provided.

For example,

INTEGER ARRAY REUSE (*) = POINT (6);

Variable Reference

The programmer can specify the location of a data item relative to the location of a previously defined identifier by following the identifier in the declaration with the identifier of another data item plus a signed word offset:

identifier = identifier ± integer

integer   is optional and is the number of words offset from the location of the referenced identifier.

Whenever the new identifier is used, the location relative to the referenced identifier is accessed.

INTEGER INTB, A = INTB + 2;

OWN Variables

Local arrays, pointers, and simple variables can be declared type OWN. This is done by preceding the complete local declaration with the reserved word OWN. The space for the data item is allocated in the DB area, not in the Q area. This means that the contents of OWN data items are retained between calls to a procedure. Strict local variables (allocated Q+), disappear when a procedure exits. If OWN variables are initialized, they are only initialized once at the beginning of the program.

OWN variables are like global variables, except that they are recognized only within the procedure where they are declared. They allow a procedure to keep data in local storage throughout the execution of a process, regardless of the number of times the procedure is invoked (called and exited).

ARRAYS

The type of array discussed in Section IV is only one of the many types of arrays that are provided in SPL/3000. These other arrays use the declaration facilities just described and can be classified as follows:

Bounded Arrays (defined bounds)
    Indirect (this is the type covered in Section IV)
    Direct
    P Relative
    Local OWN
    Dynamic Local
Equivalenced Arrays (undefined bounds)

Before discussing these arrays, it is necessary to describe the difference between *direct* and *indirect*. HP 3000 memory reference instructions can directly address a limited range of register-relative addresses; the address can be contained in the instruction itself.

If more data items are required, they must be addressed indirectly through a data label located within one of these register-relative ranges:

P − 255 to P + 255

DB + 0 to DB + 255

Q + 0 to Q + 127

Q − 0 to Q − 63

S − 0 to S − 63

Bounded Arrays

An array is bounded if it has defined upper and lower bounds (as did the arrays discussed in Section IV). For example,

INTEGER ARRAY SUMS (0:10);

A fixed number of unique locations (determined by the bounds) is allocated for each bounded array. There are five types of bounded arrays:

Indirect

Direct

P Relative

Local OWN

Dynamic Local

BOUNDED, INDIRECT ARRAYS

These are the normal (default) type of arrays. They can be either global (declared in the main program) or local (declared in a procedure) and can be of any data type (BYTE, LOGICAL, INTEGER, REAL, DOUBLE, or LONG). In both cases they are accessed indirectly through a pointer to the zero element (in the DB+ or Q+ directly addressable area). The actual data space is allocated outside the area for pointers and simple variables and is determined by the upper and lower bound; the size of the array does not change during execution. Global arrays can be initialized, but local arrays cannot.

Here is an example which shows the space allocated:

```
BEGIN                                              Stack
      INTEGER X,Y,Z;                        DB + 0  [        ]  X
      INTEGER ARRAY A(0:10);
                                            DB + 1  [        ]  Y
      PROCEDURE B(C);                       DB + 2  [        ]  Z
            VALUE C; INTEGER C;             DB + 3  [  400₈  ]  A(address)
            BEGIN
                  INTEGER ARRAY N(0:10);    DB + 400 [       ]  } Space for
                        .                                         array A
                        .
                        .
                        .                            [       ]  } Stack
            END;                                                   marker
                  .                         Q – 0 → [        ]
                  .                         
                  .          allocated      Q + 1 → [  500₈  ]  N(address)
                  .          upon    DB + 500,Q + 2 → [      ]  } Space for
END.                         procedure                            local
                             entry                                array N
                                            S
```

BOUNDED, DIRECT ARRAYS

These arrays differ from bounded, indirect arrays in only one way: they are allocated space in the areas which can be accessed without indirection. Each element is accessible independently; there is no pointer for the whole array. These arrays are accessed more efficiently but at the expense of using up the limited range of direct, register-relative locations. The method for declaring a bounded, direct array is to follow the identifier and bounds by either = DB (for global) or = Q (for local):

> *identifier (bounds)* = DB
>
> *identifier (bounds)* = Q

The words required by the elements of the array are allocated contiguously, starting with the next available DB or Q location, provided that the zero element of the array falls in the direct addressing area. If the array uses all of the remaining primary DB or Q space, no more simple variables or pointers can be declared. The array is accessed using direct memory reference instructions indexing from a zero element. Note that this does not necessarily limit the size of a direct array:

```
BEGIN                               DB + 0
     INTEGER A,B,C;
          .
          .                         DB + 100₈
          .
     ARRAY DIR'LOG (0:1000) = DB;
     <<MAIN BODY>>
END.

                                    DB + 2062₈

                                    S,Q
```

declarations

0 element
(DIRE'LOG(0))

1001
elements
of
DIR'LOG

P RELATIVE LOCAL ARRAYS

Local arrays of any type which are to contain only constants can be declared to be P relative.
These arrays must have fixed bounds and must be initialized. Space is allocated in the code
stream of the procedure for them and is filled with the initial values when compiled. These
arrays can only be read, never changed. Indirect pointers are generated only if references to
the array are out of direct range (P ± 255). P relative indirect references are self-relative (the
offset is added/subtracted to the location containing the address). ,

The format is

    identifier (bounds) = PB := *initialization*

The array must be fully initialized when compiled, since it can not be modified during execution.
For example,

```
PROCEDURE . . . .
     BEGIN
     ARRAY MSG (0:25)=PB:="ABCDEFGHIJKLMNOPQRSTUVWXYZ";
          .                          Code Segment
          .                         ┌─────────────┐
          .                         │  A        B  │
          .                         │  C        D  │
          .                         │  E        F  │
     A := MSG(10); <<WITHIN DIRECT RANGE>>  │   .   │
          .                         │   .   │
          .                         │  W        X  │
          .                         │  Y        Z  │
          .                         │    code     │
     B := MSG(20); <<OUT OF RANGE>> │  LOAD P–20,X │   Direct
          .                         │  LOAD P+2,I,X│   Indirect
     END;                           │   BRANCH    │
                                    │    –500     │   self-relative
                                    └─────────────┘   address
```

7-9

OWN LOCAL ARRAYS

An OWN array is a bounded array which belongs to a specific procedure but is located in the secondary DB area and is accessed indirectly through a Q relative pointer. OWN arrays can be accessed only by the procedure in which they are declared or by procedures to which the declarer passes the array as a calling parameter. OWN arrays can be of any type and are declared by specifying the reserved word OWN before a normal array declaration:

OWN *type* ARRAY *name* (*lower : upper*) := *initialization*;

The array space is allocated when the process begins execution and remains throughout the life of the process. A Q relative pointer is allocated by the procedure each time it is entered. This structure is shown below:

DYNAMIC LOCAL ARRAYS

Since local arrays are allocated in the stack relative to Q+ each time a procedure is entered, there is no necessity for their bounds to be constants. Therefore, in SPL/3000, local arrays can have variables as bounds; the procedure contains code which evaluates the bounds and allocates space for the array when the procedure is entered. The size of the array is fixed only for this one execution of the procedure; next time the procedure is called the array bounds can be completely different. The format is

type ARRAY *identifier* (*simple var : simple var*);

simple var   can be a global variable or a parameter variable of type integer, logical, or byte.

These arrays are always accessed indirect through a pointer allocated Q+. Therefore, this type of array cannot be initialized. For example,

```
BEGIN
    .
    .
    .
        PROCEDURE DOSOMETHING(M,N);
            INTEGER M,N;
            BEGIN
                INTEGER ARRAY LOC(M:N);
                    .
                    .
                    .
            END;
        .
        .
        DOSOMETHING(0:10);
        .
        .
        .
    END.
```

Stack after
procedure sets up
local array

DB + 0		
Q – 5	0	M
Q – 4	10	N
		Stack marker
Q – 0		
Q + 1	403	DB address of LOC(0).
DB+403,Q + 2	11 elements of LOC	space for LOC
S,Q + 12		

Equivalenced Arrays

An equivalenced array is one which reuses space already allocated to other data items. Equivalenced arrays are declared with undefined bounds (they can have no new space allocated for them since they have no defined size) and are referenced to previously defined data items. By using equivalencing, one block of storage can be associated with several identifiers, each of a different data type. For example, a buffer can be filled as words by accessing it through an integer array identifier and manipulated as bytes by referencing it as a byte array.

The address of the zero element of an unbounded array can be specified by equivalencing the array to a previously defined identifier. An array declared in this manner takes its addressing convention (direct/indirect) from the convention established for the identifier to which it is equivalenced.

These arrays can be global or local, but they cannot be initialized. The general form of their declaration is to replace the bounds with an asterisk (*) and follow the identifier by an equivalence:

$$type\ \text{ARRAY}\ name\ (*) = \left\{ \begin{array}{l} array\ (index) \\ pointer\ (index) \\ identifier \pm integer \end{array} \right\} ;$$

If the referenced item is a direct array or a variable, the result is a direct array whose zero element is the location specified. A pointer cell is therefore not allocated. For example,

```
BEGIN
      INTEGER A := 0;
      ARRAY DRAW (0:5) = DB;  <<Bounded, direct>>
      INTEGER ARRAY IDRAW (*) = DRAW;  <<Equivalenced, direct>>
      DOUBLE ARRAY DA (*) = A + 7;  <<Equivalenced, direct>>
      .
      .
      .
END.
```


If the referenced item is a pointer or an indirect array, the result array is indirect. Indirect addressing requires a data label (address). An equivalenced array can share the data label already established for the referenced array if

1. The new array is equivalenced to the zero element of referenced array, and

2. The new array and the referenced array have compatible data types; that is, they both must use byte data labels (both byte) or word data labels (both non-byte).

In all other cases, a separate data label address is allocated and initialized for the newly declared array. For example,

```
BEGIN
      POINTER P;
      BYTE ARRAY CHAR (0:71); <<Bounded indirect>>
      BYTE ARRAY BUF (*) = CHAR; <<Unbounded indirect>>
      BYTE ARRAY INPT (*) = CHAR(5); <<Unbounded, indirect—not 0 element new
          data label>>
      ARRAY OUTP (*) = CHAR; <<Unbounded, indirect—not same address type, new
          data label>>
      DOUBLE ARRAY DOUB (*) = P; <<Unbounded, indirect—0 element, uses pointer
          cell as data label>>
          .
          .
          .
END.
```

DB + 0	word address	P, address of DOUB(0)(user established)
+ 1	byte address	address of CHAR,BUF
+ 2	byte address	address of INPT(0)
+ 3	word address	address of OUTP(0)
DB + 400$_8$		◄ CHAR(0),BUF(0),OUTP(0).
	72 elements of CHAR	▲INPT(0)

Array Summary

Table 7-1 summarizes the types of arrays discussed in this manual.

Table 7-1. Summary of Array Types

Bounded arrays	Allocate array space?	Allocate a pointer cell?	Specify address of pointer explicitly?	Specify address of 0 element explicitly?	Can be initialized?
Indirect ARRAY BOX(0:25);	YES, new space	YES, in DB+ or Q+ Cell	NO, next available cell	NO, allocated in next available space	YES, if global
Direct ARRAY BOX(0:25)=DB;	YES, new space	NO, 0 element is directly accessed	NO, no pointer exists.	NO, allocated in next available space	YES, if global
P Relative ARRAY BOX(0:25)=PB:=25(0);	YES, new space in code	If array is referenced out of P±255 range.	NO, set in code if needed	NO, allocated in available code space	MUST be completely initialized
Local OWN OWN ARRAY BOX(0:25);	YES, new space	YES, Q+ cell	NO, next Q+ cell	NO, allocated in secondary DB	YES
Dynamic Local ARRAY BOX(M,N);	YES, allocated dynamically in procedure stack	YES, upon procedure entry	NO, allocated in Q	NO, array is allocated in available Q+	NO, space not allocated until proc. entry
Equivalenced Arrays (Undefined Bounds)					
Equivalenced ARRAY BOX(*)=RAR(7);	NO, re-uses space	DIRECT—NO ‾ ‾ ‾ ‾ ‾ ‾ INDIRECT—YES if needed	NO, if needed, pointer is allocated in DB+ or Q+	YES, by reference	NO, no new space

POINTERS

Pointers can be declared locally or globally and can be initialized or unitilized (see Section V). To review, the format is:

type POINTER *identifier* := @ *indexed array* or *pointer* ;

The next available DB+ or Q+ cell is allocated for the pointer and the DB-relative address specified by the indexed identifier is stored in the location

Local pointers can be declared OWN (allocated in the global area but accessed only by the procedure where declared); OWN pointers can be initialized or unitialized:

OWN POINTER P;

OWN BYTE POINTER QUICK := NOW(15);

As with arrays, global and local pointers can be equivalenced to previously allocated locations using base register reference or variable reference. New locations are not allocated for equivalenced pointers; they re-use allocated space: There are two declaration formats:

- Variable Reference

  type POINTER *name* = *identifier* ± *n*;

- Base Register Reference

$$
type\ POINTER\ name = \begin{Bmatrix} DB + m \\ Q + n \\ Q - p \\ S - q \end{Bmatrix}
$$

The specified location is accessed indirectly as a word or byte address (depending on type of pointer) whenever the pointer is used.

SIMPLE VARIABLES

Simple variables of all types can be declared globally or locally. Simple variables of types byte, logical, and integer can be equivalenced to the index register. In addition, all simple variable types can be equivalenced to specific locations (without allocating any new space) by using base register reference or variable reference. The formats are

- Variable Reference

  type name = *identifier* ± *m*;

- Base Register Reference

$$
type\ name = \begin{Bmatrix} DB + n \\ Q + p \\ Q - q \\ S - r \end{Bmatrix}
$$

The specified location is accessed whenever the declared variable identifier is used. Variables declared in this manner cannot be initialized since, like equivalenced arrays, they reuse space that is allocated to other variables or arrays.

Local simple variables are declared own by preceding the type with OWN:

OWN INTEGER A,B := 3;

EXPLICIT STACK ACCESS

Sometimes an SPL/3000 program can refer explicitly to the top of stack area:

Base Register Reference
TOS—Top of Stack
Stacked Parameters (*)

Base Register Reference

When a variable referenced to an S relative location is accessed, the stack is accessed. It is important to remember that these accesses are made by means of *memory reference instructions*. In other words, if a variable is loaded onto the stack from location S − 5, the location S − 5 is not "popped" off the stack. Within a single statement (such as an assignment statement), the compiler guarantees that all references to S relative variables are made to the same location. In other words, the compiler adjusts the S relative addresses as things are pushed and popped. This does not hold for compound statements, nor if TOS is used.

TOS—Top of Stack

When the reserved identifier TOS is used, it refers to the current top of stack. If TOS occurs on the left side of an assignment statement the effect is to eliminate the normal store at the completion of an expression and to leave the result on the top of stack.

When TOS occurs in an expression the code uses the current top of stack contents for the next operand instead of loading a new value. For example,

TOS := 2;   <<LOAD IMMEDIATE 2>>
TOS := 3;   <<LOAD IMMEDIATE 3>>
TOS := TOS + TOS;   <<ADD>>
        <<TOS NOW EQUAL 5 NOT 6>>

TOS is not a variable identifier. Use of TOS never loads or creates a new value on the stack; it merely tells the compiler to perform the indicated arithmetic operation with whatever is in the stack at the moment. Since the stack is used (implicitly) to evaluate an expression, the value of the current top of stack varies with each data operation. Programmers who use the TOS construction within an expression must know how the expression will be evaluated in order to determine what the top of stack (TOS) will contain when its value is used.

The following simple case demonstrates the dangers involved in using TOS:

TOS := %77;
B.(0:16) := TOS.(10:4); <<EXTRACT IS DONE FROM B NOT %77>>

The value of B is loaded onto the stack (to prepare for the deposit) before performing the expression, changing the value of TOS from its expected value (%77) and leading to unpredictable results.

In general, whenever TOS is used in a program, the programmer should investigate the code that is generated to insure that the expected sequence of events will occur.

Stacked Parameters (*)

The asterisk character (*) can be used in place of some parameters in procedure and subroutine call statements and in place of some address values in move-scan statements. The asterisk indicates that the value has already been loaded onto the stack.

In procedure calls, if one parameter is stacked, all parameters to the left of it must be stacked. For example

Invalid   PROC(A,B,*);

Valid   PROC(*,*,C);

This mechanism requires additional preparation for use with typed procedures or subroutines. The programmer must push a zero value of the proper type onto the stack for the returned result before he loads the first stacked parameter.

In a MOVE or SCAN statement the addresses can sometimes be specified as stacked arguments by using *. In this case the compiler allows the second parameter to be stacked even if the first is not (the compiler loads the first, then switches S – 0 for S – 1). In move words only one of the addresses can be stacked.

1. Represent the following binary number as based integer values in the bases indicated.

   $1111\ 1110\ 1100\ 0101_2$

   (base)

   a) 2 _____

   b) 4 _____

   c) 8 _____

   d) 16 _____

2. Represent the following based integer values as binary numbers.

   a) %(2) 101101111

   b) %(4) 132113

   c) %7742

   d) %(16) AC6F

3. a) Without using composite integers, construct octal constants which have the specified fields. The notation (*starting bit position : field length*) is used to describe the bit fields.

   For example,

Bit Field	Contents	
0:8	%377	→ %177523
8:8	%123	

   The first field starts at bit 0 and is 8 bits long. This field contains 377_8. The second field starts at bit 8 and is also 8 bits long. This field contains 123_8. The single word octal constant %177523 is the composite result of these two fields.

   Form single word octal constants from these bit field specifications.

BIT FIELD	CONTENTS	OCTAL NUMBER
(0:3)	4	
(3:5)	5	
(8:4)	17_8	
(12:4)	10_8	
(0:5)	23_8	
(5:8)	137_8	
(13:3)	2	
(0:8)	100_{10}	
(8:8)	255_{10}	

b) In this exercise use composite integers to form constants from the following bit field specifications.

BIT FIELD	CONTENTS	COMPOSITE INTEGER
(0:3) (3:5) (8:4) (12:4)	4 5 17_8 10_8	
(0:5) (5:8) (13:3)	23_8 137_8 2	
(0:8) (8:8)	100_{10} 255_{10}	

4. Use composite integers to form testwords (Terminal character: Test character) for the SPL/3000 SCAN statement based on the following specifications.

    a) Test character = ASCII Carriage Return
       Terminal character = ASCII Line Feed

    b) Test character = ASCII End of Text
       Terminal character = ASCII Bell

    NOTE: A table of standard ASCII characters is provided in Appendix A.

5. Write the minimum SPL/3000 code required to test three bit fields of the logical variable DATA. (Use composite integers.)

Bit field	Value
(0:2)	1
(5:1)	0
(9:3)	5

If all the bit fields specified have the values shown in the table, set DATA to TRUE, otherwise set DATA to FALSE.

6. Fill in the appropriate table entries with the names of the type transfer functions available to convert a data item from one type to another. (Some entries may remain blank.)

From	LONG	REAL	DOUBLE	To INTEGER	LOGICAL	BYTE
LONG						
REAL						
DOUBLE						
INTEGER						
LOGICAL						
BYTE						

7. Write the code required to convert a long data value (DATA), to an integer, and store the result in an integer SHORT. (Use rounding in the conversion process.)

Assume the following declarations:

LONG DATA; INTEGER SHORT;

8. Select the most appropriate data types for the variables specified. Use minimum core space in your selection. Create identifiers and initialized declarations for each item.

π = 3.14159

Eulers Constant = .577215664901533

radius = 3959

gallons = 18627235

Winn's Constant = $-4.537269537 \times 10^{23}$

Planck's Constant = 6.62559×10^{-27}

9. For each of the data types in the chart below, fill in the number of bits used, the largest possible decimal number, and the smallest possible decimal number. Choose your answers from among these choices:

# of bits	Largest$_{10}$	Smallest$_{10}$
16	(ILLEGAL TYPE)	-2147483648
32	255	(11.7 digits)-10^{77}
(ILLEGAL TYPE)	$+2147483647$	-32768
48	$+65535$	0
32	(11.7 digits)$+10^{77}$	(6.9 digits)-10^{77}
8	$+32767$	(ILLEGAL TYPE)
16	(6.9 digits)$+10^{77}$	0

TYPE	# OF BITS	LARGEST$_{10}$	SMALLEST$_{10}$
BYTE	_____	_____	_____
LOGICAL	_____	_____	_____
INTEGER	_____	_____	_____
REAL	_____	_____	_____
DOUBLE	_____	_____	_____
TRIPLE	_____	_____	_____
LONG	_____	_____	_____

10. Write a procedure which computes the average value of the elements of an integer array and returns the result to the calling program in the index register. (Remember that the value of the index register is "restored" on exit.) The procedure receives two parameters—an integer array by reference and an integer element count by value.

Hint:

Q – 3	Index
Q – 2	Relative P
Q – 1	Status
Q – 0	delta Q

11. How many total words of memory are used by these declarations?

```
BEGIN
LOGICAL ARRAY HELLO'MESSAGE (0:35) := " ***HELLO***";
BYTE ARRAY INBUF (*) = HELLO'MESSAGE;
INTEGER ARRAY SCRATCH (*) = INBUF;
END
```

12. Some of these array declarations contain errors. Make changes where required to correct each statement error.

a) BYTE ARRAY INBUF (0:71) := "***ERROR NUMBER***";

b) INTEGER ARRAY NUMBER (0:9) = DB := "1234567890";

c) BYTE ARRAY ALPHA (0:6) = NUMBER;

d) LOGICAL ARRAY DATA (0:6) = PB;

e) REAL ARRAY BIGNO (*) = DB;

f) BYTE ARRAY CHARSTRING (0:7) = PB := "RESULTS ";

g) INTEGER ARRAY TOTAL (*) = NUMBER := %77,123,–1;

h) BYTE ARRAY STRINGCHAR (0:3) = Q := "ABCD";

i) ARRAY SAM (1:6) = DB := 0,1,2,3,4,5,6;

13. Write a REAL function procedure which sums the contents of a dynamic local array and returns the result in its name. This procedure will be passed an integer number (by value). Use this parameter as the upper bound for the dynamic local array (assume the lower bound is always 0.) The dynamic local array is filled by a procedure called GETDATA:

```
GETDATA (ARRY, SIZE);
     VALUE SIZE; INTEGER SIZE; REAL ARRAY ARRY;
     <<ARRY IS THE DYNAMIC LOCAL ARRAY AND
        SIZE IS THE SIZE OF THE ARRAY>>
```

Assume procedure GETDATA is declared in the main program.

14. Write a procedure TIME'STAMP which will be called by another procedure (COMPUTE) to collect performance data. COMPUTE calls TIME'STAMP every time it is entered (and passes TIME'STAMP an initial time) and again just before it exits (to pass TIME'STAMP a final time). The current clock readings are passed to TIME'STAMP as a double integer by value. TIME'STAMP must save the initial time (first call of a pair) until the final time is passed in the second call. TIME'STAMP then computes the elapsed time between the two calls and saves the result in a double own array of 100 elements. TIME'STAMP collects 100 elapsed times to fill the own array and then calls procedure WRITETAPE to write the array on a tape:

```
WRITETAPE(ARRY);
     DOUBLE ARRAY ARRY;
     OPTION EXTERNAL;
```

After emptying the own array, TIME'STAMP resets itself to collect 100 more elapsed time values (200 calls).

Write the procedure TIME'STAMP in SPL/3000.

15. Assume that the following statement has executed completely.

   MOVE BUFB := BUFA WHILE A,0;

Write the declarations and statement which would be required to transfer the non-alphabetic character in BUFA (which terminated the move) to the character position in BUFB immediately following the moved string. Leave all the move addresses unmodified in the stack (i.e., do not use a move bytes statement.) Hint: Use equivalenced pointers.

16. In the program below, insert the statements required to "stack" the parameters needed by the call to integer procedure SAM. Start inserting after the comment <<START OF MAIN PROGRAM>>.

   BEGIN <<DECLARATIONS>>
        BYTE ARRAY STRING (0:71);
        REAL NUMBER := 1.2345; DOUBLE BIGWORD := 978633D;
        INTEGER ZERO := 0, SMALL := –1;
            .
            .
            .
        INTEGER PROCEDURE SAM (A, B, C, D);
            VALUE B, C, D; BYTE ARRAY A;
            DOUBLE B; REAL C; INTEGER D;
                BEGIN
                    .
                    .
                    .
                END;
            <<START OF MAIN PROGRAM>>

        TOS := SAM (*, *, *, SMALL); <<CALL INTEGER PROCEDURE SAM>>
   END.

   NOTE: Without stacked parameters, the call to SAM would be:
   TOS := SAM (STRING, BIGWORD, NUMBER, SMALL);.

17. a) How many words of storage are allocated as a result of these declarations

        INTEGER I := 1;
        LOGICAL L = I;
        BYTE B = L;

   b) Using the declarations from Exercise 17a determine the values of variables I, L and B after execution of the following statements. The answers should take variable type into account.

```
B := 5;
I := 10;
L := INTEGER (B) + I;
I := L –21;
```

Results

I = _____

L = _____

B = _____

SECTION VIII
ASSEMBLE Statements

The ASSEMBLE statement allows the programmer to generate any HP 3000 machine code he chooses. To use this statement, therefore, the reader must be familiar with the machine instructions as described in the HP 3000 hardware documentation. The same mnemonics described there are used in SPL/3000.[1]

Although the ASSEMBLE statement should seldom be required, there are several areas where the programmer will find it useful:

- Access to otherwise inaccessible instructions (e.g., XCH, SIO, TBC)

- Additional code optimization

- Implementation of a special sequence of instructions not generated by any high-level SPL/3000 construction

NOTE: *Example 2-2 in Section II shows an ingenious use of ASSEMBLE statements to execute (XEQ instruction) a PCAL instruction loaded onto the stack.*

SYNTAX OF ASSEMBLE STATEMENT

An ASSEMBLE statement consists of the reserved word ASSEMBLE followed by a list of instructions in a set of parentheses. Each instruction consists of an optional label (plus colon) and mnemonic. Instructions are separated by semicolons. Each mnemonic of the instruction set is specified in one of nine formats (depending upon the type of operands required). The format of an ASSEMBLE statement is

ASSEMBLE (*instruction; instruction; . . . . ; instruction*);

instruction consists of an optional label (Section III) followed by a mnemonic and its arguments:

label: mnemonic; mnemonic;

---

[1] BCC (branch on condition code) is replaced by six opcodes specifying the particular branch conditions:

BL	Branch if less than	BG	Branch if greater than
BE	Branch if equal	BNE	Branch if not equal
BLE	Branch if less than or equal	BGE	Branch if greater than or equal

Here is an ASSEMBLE statement and an equivalent high-level sequence:

<<Assemble>>	<<High-level>>
ASSEMBLE(	<<assume integer X equivalenced to Index Register>>
STAX, DECX;	X := TOS – 1;
PUT: STOR ARY, I, X;	PUT: ARY(X) := TOS;
BR GET);	GO TO GET;

<<MNEMONICS NEED NOT BE PLACED ONE PER LINE, BUT THAT IS OFTEN
THE MOST READABLE FORM>>

MNEMONIC FORMATS

There are nine mnemonic formats designed for different instructions:

Format 1:	Memory reference instructions
Format 2:	Stack ops
Format 3:	Bit shifts Bit tests Test and branches
Format 4:	Immediates Extract/deposit Push/set registers
Format 5:	Read switch register Link list search Privileged load/store
Format 6:	Pause/halt Exchange DB Execute I/O instructions
Format 7:	Calls/exits Boolean immediates S register arithmetic X register immediate arithmetic Load double from program Load label Test and set bits in memory
Format 8:	Moves, scans, and compare bytes
Format 9:	Constant definition
Format 10:	Extended precision arithmetic Decimal arithmetic

Conventions Used

The mnemonic formats are described using the following conventions:

Item	Explanation
I	Indirection
X	Index register
Capital letters	1. Opcodes 2. Literals, options, and other non-variable items
Italics	Variable items or classes of items
[ ]	Pick one item from within the brackets; the entire item is optional
{ }	Pick one item from within the brackets; one item is required.

Format 1

1a
$$
\begin{Bmatrix} \text{LOAD} \\ \text{LDX} \\ \text{LRA} \\ \text{CMPM} \\ \text{ADDM} \\ \text{SUBM} \\ \text{MPYM} \end{Bmatrix}
\begin{Bmatrix} \textit{label id} \\ \textit{variable id} \\ \text{DB} + \textit{usi}255 \\ \text{P} + \textit{usi}255 \\ \text{P} - \textit{usi}255 \\ \text{Q} + \textit{usi}127 \\ \text{Q} - \textit{usi}63 \\ \text{S} - \textit{usi}63 \end{Bmatrix}
\quad [,\text{I}] \quad [,\text{X}]
$$

1b
$$
\begin{Bmatrix} \text{LDB} \\ \text{LDD} \\ \text{STOR} \\ \text{STB} \\ \text{STD} \\ \text{INCM} \\ \text{DECM} \end{Bmatrix}
\begin{Bmatrix} \textit{variable id} \\ \text{DB} + \textit{usi}255 \\ \text{Q} + \textit{usi}127 \\ \text{Q} - \textit{usi}63 \\ \text{S} - \textit{usi}63 \end{Bmatrix}
\quad [,\text{I}] \quad [,\text{X}]
$$

1c
$$
\text{BR} \quad
\begin{Bmatrix} \textit{label id} \\ \text{P} + \textit{usi}255 \\ \text{P} - \textit{usi}255 \end{Bmatrix}
\quad [,\text{I}] \quad [,\text{X}]
$$

$$
\text{BR} \quad
\begin{Bmatrix} \text{DB} + \textit{usi}255 \\ \text{Q} + \textit{usi}127 \\ \text{Q} - \textit{usi}63 \\ \text{S} - \textit{usi}63 \end{Bmatrix}
\quad ,\text{I} \quad [,\text{X}]
$$

1d
$$
\begin{Bmatrix} \text{BL} \\ \text{BE} \\ \text{BLE} \\ \text{BG} \\ \text{BNE} \\ \text{BGE} \end{Bmatrix}
\begin{Bmatrix} \textit{label id} \\ \text{P} + \textit{usi}31 \\ \text{P} - \textit{usi}31 \end{Bmatrix}
\quad [,\text{I}]
$$

1e
$$
\begin{Bmatrix} \text{TBA} \\ \text{MTBA} \\ \text{TBX} \\ \text{MTBX} \end{Bmatrix}
\begin{Bmatrix} \textit{label id} \\ \text{P} + \textit{usi}255 \\ \text{P} - \textit{usi}255 \end{Bmatrix}
$$

where

> *variable id*   is a simple variable, pointer, or array identifier, (indirection is not supplied automatically).

> *usi*   is an unsigned integer less than or equal to the number following.

> *label id*   is a label which is used to label a statement within the range of the instruction.

For example,

    ASSEMBLE(STB S − 1, I, X; DECM VAR);

Format 2

> *stackop*

>   or

> *stack op*, *stack op*

In the first case the compiler fills in the second half of the instruction word with a NOP.

The legal *stackops* are as follows:

NOP	DNEG	XCH	FLT	NOT
DELB	DXCH	INCA	FCMP	OR
DDEL	CMP	DECA	FADD	XOR
XROX	ADD	XAX	FSUB	AND
INCX	SUB	ADAX	FMPY	FIXR
DECX	MPY	ADXA	FDIV	FIXT
ZERO	DIV	DEL	FNEG	INCB
DZRO	NEG	ZROB	CAB	DECB
DCMP	TEST	LDXB	LCMP	XBX
DADD	STBX	STAX	LADD	ADBX
DSUB	DTST	LDXA	LSUB	ADXB
MPYL	DFLT	DUP	LMPY	
DIVL	BTST	DDUP	LDIV	

For example,

    ASSEMBLE(DDUP, DELB; STAX);

Format 3

3a $\begin{Bmatrix} \text{IABZ} \\ \text{IXBZ} \\ \text{DXBZ} \\ \text{BCY} \\ \text{BNCY} \\ \text{CPRB} \\ \text{DABZ} \\ \text{BOV} \\ \text{BNOV} \\ \text{BRO} \\ \text{BRE} \end{Bmatrix}$ $\begin{Bmatrix} label \\ \text{P} \pm usi31 \\ * \pm usi31 \end{Bmatrix}$ [,I]

In these branch instructions, the address can be specified as a *label* or a P relative address (P± or *± are the same thing). If the label location is not within 31 locations of P (P ± 31), the compiler tags this as an error; indirection is *not* supplied automatically within an ASSEMBLE statement.

3b
$$\left.\begin{array}{l} \text{ASL} \\ \text{ASR} \\ \text{LSL} \\ \text{LSR} \\ \text{CSL} \\ \text{CSR} \\ \text{SCAN} \\ \text{TASL} \\ \text{TASR} \\ \text{TNSL} \\ \text{DASL} \\ \text{DASR} \\ \text{DLSL} \\ \text{DLSR} \\ \text{DCSL} \\ \text{DCSR} \\ \text{TBC} \\ \text{TRBC} \\ \text{TSBC} \\ \text{TCBC} \\ \text{QASL} \\ \text{QASR} \end{array}\right\} \quad usi63 \qquad [,X]$$

*usi*63 is a shift count or number of bits less than or equal to 63. For example,

ASSEMBLE(LSL 1; BRE QUIT);

Format 4

4a
$$\left.\begin{array}{l} \text{LDI} \\ \text{LDXI} \\ \text{CMPI} \\ \text{ADDI} \\ \text{SUBI} \\ \text{MPYI} \\ \text{DIVI} \\ \text{PSHR}^\dagger \\ \text{LDNI} \\ \text{LDXN} \\ \text{CMPN} \\ \text{SETR}^\dagger \end{array}\right\} \quad usi255$$
          † = a privileged instruction for some registers

4b $\left\{\begin{array}{l} \text{EXF} \\ \text{DPF} \end{array}\right\}$ *usi*15 : *usi*15

For example,

ASSEMBLE (LDI 255; ADDI 5; EXF 7:9);

Format 5

$$\left\{\begin{array}{l} \text{RSW} \\ \text{LLSH}\dagger \\ \text{PLDA}\dagger \\ \text{PSTA}\dagger \\ \text{LSEA}\dagger \\ \text{SSEA}\dagger \\ \text{LDEA}\dagger \\ \text{SDEA}\dagger \\ \text{IXIT}\dagger \\ \text{LOCK}\dagger \\ \text{PCN}\dagger \\ \text{UNLK}\dagger \end{array}\right\}$$

$\dagger$ = a privileged instruction

For example,

ASSEMBLE (RSW; PLDA; . . . LLSH; . . . PSTA);

Format 6

$$\left\{\begin{array}{l} \text{PAUS} \\ \text{SED} \\ \text{XCHD} \\ \text{SMSK} \\ \text{RMSK} \\ \text{XEQ} \\ \text{SIO} \\ \text{RIO} \\ \text{WIO} \\ \text{TIO} \\ \text{CIO} \\ \text{CMD} \\ \text{SIN} \\ \text{HALT} \\ \text{LST} \\ \text{PSDB} \\ \text{DISP} \\ \text{PSEB} \\ \text{SCLK} \\ \text{RCLK} \\ \text{SST} \end{array}\right\}$$ *usi*15

For example,

ASSEMBLE (XEQ 4);

All of these instructions except XEQ and RMSK are privileged.

Format 7

$$\left\{\begin{array}{l} \text{PCAL} \\ \text{SCAL} \\ \text{EXIT} \\ \text{SXIT} \\ \text{ADXI} \\ \text{SBXI} \\ \text{LLBL} \\ \text{LDPP} \\ \text{LDPN} \\ \text{ADDS} \\ \text{SUBS} \\ \text{ORI} \\ \text{XORI} \\ \text{ANDI} \end{array}\right\} \quad usi255$$

        PCAL *procedure identifier*

        SCAL (user must load label onto stack)

        LLBL *procedure identifier*

For example,

        ASSEMBLE (PCAL READ; . . . . SCAL LOOPER; . . . ORI %377);

Format 8

8a
$$\left\{\begin{array}{l} \text{MOVE} \\ \text{MVB} \\ \text{CMPB} \end{array}\right\} \quad [\text{PB}] \quad \left[\begin{array}{l} ,0 \\ ,1 \\ ,2 \\ ,3 \end{array}\right]$$

If item two is empty, a DB relative move is assumed.

If item three is empty, the stack decrement is 3.

8b
$$\text{MVBW} \quad \left\{\begin{array}{l} \text{A} \\ \text{N} \\ \text{AN} \\ \text{AS} \\ \text{ANS} \end{array}\right\} \quad \left[\begin{array}{l} ,0 \\ ,1 \\ ,2 \end{array}\right]$$

If item three is empty, the stack decrement is 2.

8c
$$\left\{\begin{array}{l} \text{MVBL†} \\ \text{MVLB†} \\ \text{SCW} \\ \text{SCU} \end{array}\right\} \quad \left[\begin{array}{l} ,0 \\ ,1 \\ ,2 \end{array}\right]$$
                                   †Privileged instruction.

If item two is missing, the stack decrement is 2. For example,

        ASSEMBLE (SCW, 1);
        ASSEMBLE (MVBW AN, 0);
        ASSEMBLE (CMPB PB, 1);

$$8d \left\{ \begin{array}{l} \text{MABS†} \\ \text{MTDS†} \\ \text{MDS†} \\ \text{MFDS†} \end{array} \right\} \left[\begin{array}{l} 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \text{ for MABS and} \\ \text{MDS} \end{array} \right]$$

Format 9

        CON   *constant list*

This format is actually a psuedo-mnemonic for constant generation; it is not a hardware instruction.

CON stores a series of constants in the code starting at the current location. In addition to all numerical and string constants, P relative address constants can be created by listing label identifiers (this is used to create addresses for indirect references). The CON instruction itself can be labeled so that other instructions can reference the constants symbolically.

        ASSEMBLE(
            BR P + 1, I;
            CON LABELNAME);
        ASSEMBLE (TAB: CON "ABCDEFGH"; . . . . .
            LDB TAB, X; . . . .);

Format 10

    10a   DMUL
            DDIV
            EADD
            ESUB
            EMPY
            EDIV
            ENEG
            ECMP
            DMPY

$$10b \left\{ \begin{array}{l} \text{CVAD} \\ \text{CVBD} \end{array} \right\} \left[\begin{array}{l} 0 \\ 1 \end{array} \right]$$

If item 2 is 0, 2 words are deleted from the stack.
If item 2 is 1 or empty, 4 words are deleted from the stack.

10c
$$\text{CVDB} \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

If item 2 is 0, 2 words are deleted from the stack.
If item 2 is 1 or empty, 3 words are deleted from the stack.

10d
$$\begin{Bmatrix} \text{ADDD} \\ \text{SUBD} \\ \text{MPYD} \\ \text{CMPD} \\ \text{SLD} \\ \text{NSLD} \\ \text{SRD} \end{Bmatrix} \begin{bmatrix} 0 \\ 1 \\ 2 \end{bmatrix}$$

If item 2 is 0, no words are deleted from the stack.
If item 2 is 1, 2 words are deleted from the stack.
If item 2 is 2 or empty, 4 words are deleted from the stack.

10e
$$\text{CVDA} \begin{bmatrix} 0 \\ 1 \\ \text{ABS} \\ \text{NABS} \end{bmatrix} \left[, \begin{Bmatrix} 0 \\ 1 \end{Bmatrix} \right]$$

If 0 is specified, 1 word is deleted from the stack.
If 1 is specified, 3 words are deleted from the stack.
If neither 0 nor 1 is specified, 3 words are deleted from the stack.
If ABS is specified, the target sign will be negative if the source
is negative; otherwise, the target will be unsigned.
If NABS is specified, the target will be unsigned.
If neither ABS nor NABS is specified, the target sign will be the
same as the source.

USES OF THE ASSEMBLE STATEMENT

The ASSEMBLE statement *does not* provide automatic indirection when references to identifiers
are out of range of a particular instruction; out of range conditions are flagged as errors. It is
the programmer's responsibility to specify indirect addressing and provide an indirect address
(using the CON psuedo-op).

These are certain hardware instructions that do not require an argument (SCAN, TNSL) even
though one is specified in the format syntax. The compiler accepts them with or without an
argument (the argument is ignored if not needed and the compiler substitutes a zero to allow
for future hardware use.

Alphabetical Listing of Instructions

Mnemonic	Function	Format
ADAX	Add A to X	2
ADBX	Add B to X	2
ADD	Add	2
ADDD	Decimal add	10d
ADDI	Add immediate	4a
ADDM	Add memory	1a
ADDS	Add to S	7
ADXA	Add X to A	2
ADXB	Add X to B	2
ADXI	Add immediate to X	4a
AND	And, logical	2
ANDI	Logical AND immediate	7
ASL	Arithmetic shift left	3b
ASR	Arithmetic shift right	3b
BCC	Branch on Condition Code	1d
BCY	Branch on carry	3a
BE	Branch on equals	
BG	Branch on greater than	
BGE	Branch on greater than or equal	See BCC
BL	Branch on less than	
BLE	Branch on less than or equal	
BNE	Branch on not equal	
BNCY	Branch on no carry	3a
BNOV	Branch on no overflow	3a
BOV	Branch on overflow	3a
BR	Branch	1c
BRE	Branch on TOS even	3a
BRO	Branch on TOS odd	3a
BTST	Test byte on TOS	2
CAB	Rotate ABC	2
CIO	Control I/O	6
CMD	Command	6
CMP	Compare	2
CMPB	Compare bytes	2
CMPD	Compare decimal	10d
CMPI	Compare immediate	4a
CMPM	Compare memory	1a
CMPN	Compare negative immediate	4a
CPRB	Compare range and branch	3a
CSL	Circular shift left	3b
CSR	Circular shift right	3b
CVAD	Convert ASCII to packed decimal	10b
CVBD	Convert binary to packed decimal	10b
CVDA	Convert packed decimal to ASCII	10e
CVDB	Convert packed decimal to binary	10c
DABZ	Decrement A, branch if zero	3a
DADD	Double add	2
DASL	Double arithmetic shift left	3b
DASR	Double arithmetic shift right	3b
DCMP	Double compare	2
DCSL	Double circular shift left	3b

Mnemonic	Function	Format
DCSR	Double circular shift right	3b
DDEL	Double delete	2
DDIV	Double divide	10a
DDUP	Double duplicate	2
DECA	Decrement A	2
DECB	Decrement B	2
DECM	Decrement memory	1b
DECX	Decrement X	2
DEL	Delete A	2
DELB	Delete B	2
DFLT	Double float	2
DISP	Dispatch	6
DIV	Divide	2
DIVI	Divide immediate	4a
DIVL	Divide Long	2
DLSL	Double logical shift left	3b
DLSR	Double logical shift right	3b
DMPY	Double logical multiply	10a
DMUL	Double multiply	10a
DNEG	Double negate	2
DPF	Deposit field	4b
DSUB	Double subtract	2
DTST	Test double word on TOS	2
DUP	Duplicate A	2
DXBZ	Decrement X, branch if zero	3a
DXCH	Double exchange	2
DZRO	Double push zero	2
EADD	Extended-precision floating point add	10a
ECMP	Extended-precision floating point compare	10a
EDIV	Extended-precision floating point divide	10a
EMPY	Extended-precision floating point multiply	10a
ENEG	Extended-precision floating point negate	10a
ESUB	Extended-precision floating point subtract	10a
EXF	Extract field	4b
EXIT	Procedure and interrupt exit	7
FADD	Floating add	2
FCMP	Floating compare	2
FDIV	Floating divide	2
FIXR	Fix and round	2
FIXT	Fix and truncate	2
FLT	Float	2
FMPY	Floating multiply	2
FNEG	Floating negate	2
FSUB	Floating subtract	2
HALT	Halt	6
IABZ	Increment A, branch if zero	3a
INCA	Increment A	2
INCB	Increment B	2
INCM	Increment memory	1b
INCX	Increment index	2
IXBZ	Increment X, branch if zero	3a
IXIT	Interrupt exit	5
LADD	Logical add	2

Mnemonic	Function	Format
LCMP	Logical compare	2
LDB	Load byte	1b
LDD	Load double	1b
LDEA	Load double word from extended address	5
LDI	Load immediate	4a
LDIV	Logical divide	2
LDNI	Load negative immediate	4a
LDPN	Load double from program, negative	7
LDPP	Load double from program, positive	7
LDX	Load Index	1a
LDXA	Load X onto stack	2
LDXB	Load X into B	2
LDXI	Load X immediate	4a
LDXN	Load X negative immediate	4a
LLBL	Load Label	7
LLSH	Linked list search	5
LMPY	Logical multiply	2
LOAD	Load	1a
LOCK	Lock resource	5
LRA	Load relative address	1a
LSEA	Load single word from extended address	5
LSL	Logical shift left	3b
LSR	Logical shift right	3b
LST	Load from system table	6
LSUB	Logical subtract	2
MABS	Move using absolute address	8
MDS	Move using data segment	8
MFDS	Move from data segment	8
MOVE	Move words	8a
MPYD	Decimal Multiply	10d
MPY	Multiply	2
MPYI	Multiply immediate	4a
MPYL	Multiply Long	2
MPYM	Multiply memory	1a
MTBA	Modify, Test, Branch, A	1e
MTBX	Modify, Test, Branch, X	1e
MTDS	Move to data segment	8
MVB	Move bytes	8a
MVBL	Move from DB+ to DL+	8c
MVBW	Move bytes while	8b
MVLB	Move from DL+ to DB+	8c
NEG	Negate	2
NOP	No operation	2
NOT	One's complement	2
NSLD	Normalizing shift left decimal	10d
OR	Or, logical	2
ORI	Logical OR immediate	7
PAUS	Pause	6
PCAL	Procedure call	7
PCN	Push CPU number	5
PLDA	Privileged load from absolute address	5
PSDB	Pseudo interrupt disable	6
PSEB	Pseudo interrupt enable	6
PSHR	Push registers	4a

Mnemonic	Function	Format
PSTA	Privileged store into absolute address	5
RCLK	Read clock	6
RIO	Read I/O	6
QASL	Quadruple arithmetic shift left	3b
QASR	Quadruple arithmetic shift right	3b
RMSK	Read Mask	6
RSW	Read Switch register	5
SBXI	Subtract immediate from X	7
SCAL	Subroutine Call	7
SCAN	Scan bits	3b
SCLK	Store clock	6
SCU	Scan until	8c
SCW	Scan while	8c
SDEA	Store double word into extended address	5
SED	Set enable/disable external interrupts	6
SETR	Set registers	4a
SIN	Set interrupt	6
SIO	Start I/O	6
SLD	Shift left decimal	10d
SMSK	Set Mask	6
SRD	Shift right decimal	10d
SSEA	Store single word into extended address	5
SST	Store in system table	6
STAX	Store A into X	2
STB	Store byte	1b
STBX	Store B into X	2
STD	Store double	1b
STOR	Store	1a
SUB	Subtract	2
SUBD	Subtract decimal	10d
SUBI	Subtract immediate	4a
SUBM	Subtract memory	1a
SUBS	Subtract from S	7
SXIT	Subroutine exit	7
TASL	Triple arithmetic shift left	3b
TASR	Triple arithmetic shift right	3b
TBA	Test, branch, A	1e
TBC	Test bit and set condition code	3b
TBX	Test, branch, X	1e
TCBC	Test and complement bit and set CC	3b
TEST	Test TOS	2
TIO	Test I/O	6
TNSL	Triple normalizing shift left	3b
TRBC	Test and reset bit, set condition code	3b
TSBC	Test, set bit, set condition code	3b
UNLK	Unlock resource	5
WIO	Write I/O	6
XAX	Exchange A and X	2
XBX	Exchange B and X	2
XCH	Exchange A and B	2
XCHD	Exchange DB	6
XEQ	Execute	6
XOR	Exclusive OR, logical	2
XORI	Logical Exclusive OR immediate	7

Mnemonic	Function	Format
ZERO	Push zero	2
ZROB	Zero B	2
ZROX	Zero X	2

EXAMPLE 8-1. DECIMAL TO HEX CONVERSION

This program reads a decimal or octal number, converts the number to binary, uses an ASSEMBLE Statement to convert the binary number to ASCII hexadecimal digits, and outputs the resulting hexadecimal value.

The machine language portion performs these functions:

- Isolates four binary bits

- Tests the value to determine whether the digit is represented by 0–9 or A–F and adds an appropriate number to the original value to form the corresponding ASCII character

- Stores characters right to left in the buffer, shifts the binary word right four bits, and repeats the operation (steps 1, 2, and 3) until all four hexadecimal digits have been determined

- Loads the stack with the array address.

- Makes a procedure call to output the results

This program illustrates the first five formats of ASSEMBLE mnemonics:

Format 1: Memory reference

Format 2: Stack ops [not packed stack op]

Format 3: Branches

Format 4: Immediates

Format 5: PCAL — logic functions

Input/Output

```
ENTER A VALUE
8
 0008
ENTER A VALUE
16
 0010
ENTER A VALUE
1274
 04FA
ENTER A VALUE
3247
 0CAF
ENTER A VALUE
79
 004F
```

8-14

Listing

```
BEGIN   <<EXAMPLE 8-1. DECIMAL TO HEX CONVERSION>>
COMMENT:
        THIS PROGRAM INPUTS A DECIMAL INTEGER, CONVERTS ITS BINARY
        VALUE TO HEXADECIMAL, AND OUTPUTS THE HEX EQUIVALENT.
        (ONE HEX DIGIT REPRESENTS FOUR BINARY BITS.)
        0000 = 0                1010 = A
        ::::                    ::::
        1001 = 9                1111 = F
        NOTE: "INPUT" AND "OUTPUT" ARE DUMMY PROCEDURES WHICH SIMULATE
        INPUT, OUTPUT, AND CONVERSION - THEY ARE NOT PART OF SPL/3000;
BYTE ARRAY DATAMSG(0:13):="ENTER A VALUE ";
BYTE ARRAY HEX(0:4):="    ";
LOGICAL BIN;
LABEL NEXTDIGIT;
<<END OF DECLARATIONS>>
        OUTPUT(DATAMSG);   <<REQUEST DATA VALUE>>
        INPUT(BIN);   <<READ DECIMAL NUMBER>>
        ASSEMBLE(
            LDXI 4;         <<SET LOOP INDEX>>
            LOAD BIN;       <<PLACE BINARY NUMBER ON STACK>>
NEXTDIGIT:  DUP;            <<COPY>>
            ANDI %17;       <<ISOLATE LOW ORDER 4 BITS>>
            DUP;            <<COPY>>
            CMPI 10;        <<COMPARE VALUE TO 10>>
            BL P+2;         <<BRANCH IF LESS THAN 10>>
            ADDI %7;        <<ADD %67 IF DIGIT IS A-F>>
            ADDI %60;       <<ADD %60 IF DIGIT IS 0-9>>
            STB HEX,I,X;    <<STORE HEX DIGIT IN ARRAY RT TO LFT>>
            LSR 4;          <<SHIFT BINARY NUMBER RT ONE HEX DIGIT>>
            DXBZ P+2;       <<STOP CONVERSION IF INDEX=0>>
            BR NEXTDIGIT;   <<CONVERT ANOTHER DIGIT>>
            <<END OF CONVERSION LOOP - OUTPUT HEX NUMBER>>
            LOAD HEX;       <<LOAD STARTING ADDR OF  ARRAY>>
            LSR 1;          <<CONVERT TO WORD ADDRESS>>
            PCAL OUTPUT;    <<CALL PROCEDURE>>
          <<END ASSEMBLE>> );
END   <<DECIMAL TO HEX CONVERSION>>.
```

Note: LOAD means "load TOS from memory",

LDXI means "load Index Register with constant",

STOR means "store TOS into memory."

Consult the *HP 3000 Reference Manual (HP 03000-90019)* for details.

1. Assume these declarations:

INTEGER IA := 5,

        IB := 4,

        IC := 3;

INTEGER POINTER IPTR := @IA;

LOGICAL LA := 2,

        LB := 1;

BYTE ARRAY STRING (0:3) := "ABCD";

DB + 0	5
1	4
2	3
3	0
4	2
5	1
6	1000
	.
	.
	.
400	A B
	C D

Examine the following ASSEMBLE statements and complete the diagrams to show the contents of the stack after each ASSEMBLE statement has been executed. (Use scratch paper to record intermediate results.)

a) ASSEMBLE (LDXI 1; <<Load index register immediate>>

        LOAD IA, X;

        LOAD STRING;

        LOAD IPTR;

        LOAD IPTR, X;

        LOAD IPTR, I, X);

S – 6	
S – 5	
S – 4	
S – 3	
S – 2	
S – 1	
S – 0	

b) ASSEMBLE (LOAD STRING;
                    LOAD IC:
                    LOAD S – 0, I;
                    STOR S – 3;
                    LOAD LB, I)

```
S – 4  ┌─────────────┐
       │             │
S – 3  ├─────────────┤
       │             │
S – 2  ├─────────────┤
       │             │
S – 1  ├─────────────┤
       │             │
S – 0  └─────────────┘
```

2. Assume these declarations:

        LOGICAL COUNTER, MASK;
        PROCEDURE OUTPUT;
        LABEL NEXT;

Identify the invalid instructions below:

              BR DB + 40;
              INCM COUNTER;
    NEXT: DIV, DELB;
              ZERO;
              ASL 32;
              RSW, DUP;
              PCAL OUTPUT;
              MVBW S;
              ORI MASK;
              BCY NEXT;

3. Select the non-privileged instruction mnemonics:

SIO	HALT	RSW	PSHR	EXF
XEQ	XCHD	PLDA	CMD	DPF

APPENDIX A
ASCII Character Set

Graphic	Decimal Value	Octal Value	Comments
	0	0	Null
	1	1	Start of heading
	2	2	Start of text
	3	3	End of text
	4	4	End of transmission
	5	5	Enquiry
	6	6	Acknowledge
	7	7	Bell
	8	10	Backspace
	9	11	Horizontal tabulation
	10	12	Line feed
	11	13	Vertical tabulation
	12	14	Form feed
	13	15	Carriage return
	14	16	Shift out
	15	17	Shift in
	16	20	Data link escape
	17	21	Device control 1
	18	22	Device control 2
	19	23	Device control 3
	20	24	Device control 4
	21	25	Negative acknowledge
	22	26	Synchronous idle
	23	27	End of transmission block
	24	30	Cancel
	25	31	End of medium
	26	32	Substitute
	27	33	Escape
	28	34	File separator
	29	35	Group separator
	30	36	Record separator
	31	37	Unit separator
	32	40	Space
!	33	41	Exclamation point
"	34	42	Quotation mark
#	35	43	Number sign
$	36	44	Dollar sign
%	37	45	Percent sign

Graphic	Decimal Value	Octal Value	Comments
&	38	46	Ampersand
'	39	47	Apostrophe
(	40	50	Opening parenthesis
)	41	51	Closing parenthesis
*	42	52	Asterisk
+	43	53	Plus
,	44	54	Comma
−	45	55	Hyphen (Minus)
.	46	56	Period (Decimal)
/	47	57	Slant
0	48	60	Zero
1	49	61	One
2	50	62	Two
3	51	63	Three
4	52	64	Four
5	53	65	Five
6	54	66	Six
7	55	67	Seven
8	56	70	Eight
9	57	71	Nine
:	58	72	Colon
;	59	73	Semicolon
<	60	74	Less than
=	61	75	Equals
>	62	76	Greater than
?	63	77	Question mark
@	64	100	Commerical at
A	65	101	Uppercase A
B	66	102	Uppercase B
C	67	103	Uppercase C
D	68	104	Uppercase D
E	69	105	Uppercase E
F	70	106	Uppercase F
G	71	107	Uppercase G
H	72	110	Uppercase H
I	73	111	Uppercase I
J	74	112	Uppercase J
K	75	113	Uppercase K
L	76	114	Uppercase L
M	77	115	Uppercase M
N	78	116	Uppercase N
O	79	117	Uppercase O
P	80	120	Uppercase P
Q	81	121	Uppercase Q
R	82	122	Uppercase R
S	83	123	Uppercase S
T	84	124	Uppercase T
U	85	125	Uppercase U
V	86	126	Uppercase V
W	87	127	Uppercase W
X	88	130	Uppercase X
Y	89	131	Uppercase Y

Graphic	Decimal Value	Octal Value	Comments
Z	90	132	Uppercase Z
[	91	133	Opening bracket
\	92	134	Reverse slant
]	93	135	Closing bracket
^	94	136	Circumflex
—	95	137	Underscore
`	96	140	Grave accent
a	97	141	Lowercase a
b	98	142	Lowercase b
c	99	143	Lowercase c
d	100	144	Lowercase d
e	101	145	Lowercase e
f	102	146	Lowercase f
g	103	147	Lowercase g
h	104	150	Lowercase h
i	105	151	Lowercase i
j	106	151	Lowercase j
k	107	152	Lowercase k
l	108	154	Lowercase l
m	109	155	Lowercase m
n	110	156	Lowercase n
o	111	157	Lowercase o
p	112	160	Lowercase p
q	113	161	Lowercase q
r	114	162	Lowercase r
s	115	163	Lowercase s
t	116	164	Lowercase t
u	117	165	Lowercase u
v	118	166	Lowercase v
w	119	167	Lowercase w
x	120	170	Lowercase x
y	121	171	Lowercase y
z	122	172	Lowercase z
{	123	173	Opening (left) brace
		124	174	Verical line
}	125	175	Closing (right) brace
~	126	177	Tilde
	127	177	Delete

The following symbols have special meaning in SPL/3000 and thus, cannot be used as identifiers:

ABSOLUTE	ELSE	LABEL	PUSH
ALPHA	END	LAND	REAL
AND	ENTRY	LOGICAL	RETURN
ARRAY	EQUATE	LONG	SCAN
ASSEMBLE	EXTERNAL	LOR	SET
BEGIN	FALSE	MOD	SPECIAL
BYTE	FIXR	MODD	STEP
CARRY	FIXT	MOVE	SUBROUTINE
CASE	FOR	NOCARRY	SWITCH
CAT	FORWARD	NOT	THEN
CHECK	GLOBAL	NOVERFLOW	TO
COMMENT	GO	NUMERIC	TOS
DABZ	GOTO	OF	TRUE
DDEL	IABZ	OPTION	UNCALLABLE
DEFINE	IF	OR	UNTIL
DEL	INTEGER	OVERFLOW	VALUE
DELB	INTERNAL	OWN	VARIABLE
DO	INTERRUPT	POINTER	WHILE
DOUBLE	INTRINSIC	PRIVILEGED	XOR
DXBZ	IXBZ	PROCEDURE	

APPENDIX C
Brief Summary of Commands

Before the compiler commands can be used, the programmer must enter the operating system and the SPL/3000 compiler itself.

OPERATING SYSTEM COMMANDS

The operating system (MPE/3000) is accessed using the :HELLO command (on-line terminal mode) or the :JOB command (batch mode). These two commands require specification of a job name, a user and account pair, and necessary passwords. The command for accessing the SPL/3000 compiler is :SPL with optional parameters specifying source file, object file, list file, and editing. The object code generated by the SPL/3000 compiler is prepared using the :PREP command (or by the compiler with :SPLPREP which compiles and prepares) and is executed using the :RUN command (or by the compiler with :SPLGO which compiles, prepares, and runs). All of these commands are treated in detail in the MPE/3000 reference manual.

COMPILER COMMANDS

Commands to the compiler are specified by a $ in the first column followed by a command word, a space, and a list of parameters separated by commas. The most common command is the $CONTROL command; its parameters, in simplified form, are as follows:

$CONTROL	*parameter list*
LIST	Send each source record to the list file.
NOLIST	Send only invalid source records and error messages to the list file.
WARN	Send warning messages and the records causing them to the list file.
NOWARN	Do not send warning messages to the list file.
MAP	Send a symbol table dump to the list file after each procedure or main body.

ERRORS = *ddd*	Set the maximum severe-error count to *ddd*; exceeding this number terminates compilation (0 <= *ddd* <= 999).
CODE	Send record of actual code emitted (in octal) to the list file at appropriate points.
NOCODE	Do not sent code listing to the list file.
ADR	After each declaration, if the LIST command is in effect, send the address mode and location of each variable to the list file.
INNERLIST	Send an innerlist of code emitted for each statement to the list file if the LIST command is in effect.
MAIN = *program-name*	Establish *program-name* (up to 8 alphanumeric characters, starting with alpabetic).

NOTE: A NOLIST parameter "turns off" ADR and INNERLIST if they are in effect.

Default Conditions

LIST, WARN, ERRORS = *constant*, NOCODE, MAIN = OB'

MAP, ADR, and INNERLIST are off at the beginning of a compilation.

A $CONTROL command overrides parameters of previous $CONTROL commands. Thus, parts of a program can be listed while others are not.

APPENDIX D
Answers to Exercises

ANSWERS TO SECTION I EXERCISES

1. A process is the unique execution of a program by a particular user at a particular time.

2. Code segments and data segment.

3. Code cannot be modified; therefore, it is re-entrant and can be shared by several users. Data can be modified; therefore, each user process has a private data storage area.

4.

code		data	
	← PB		← DL
			← DB
	← P		← Q
			← S
	← PL		← Z

Each register listed has a special addressing function.

REGISTER	USED TO ADDRESS
PB	Start of code segment
P	Currently executing instruction
PL	End of code segment
DL	Start of data segment
DB	First word of stack global area
Q	Current stack marker
S	Current top of stack location
Z	End of data segment

5.
P	+255, −255
DL	not used for addressing
DB	+255
Q	+127, −63
S	−63
Z	not used for addressing

6.

LOAD A	LOAD B	SUB	LOAD C	LOAD D	ADD	MPY
				2		
	3		2	3	2	
S 3	S 1	S 2	S 3	S 1	S 4	S 8

7. (A – B) * (C + D)

8. a) Addressing relative to registers.

   b) Bounds checking of each effective address against limit register contents by hardware.

   c) Code segments cannot be modified.

ANSWERS TO SECTION II EXERCISES

Note: [   ] indicates section corrected. There may be other correct solutions.

1. a) INTEGER I, J := 100, K: Correct statement as is.

   b) REAL NUMBER [;] INTEGER SUM := 0;

   c) INTEGER A := [123], TESTER := %6412;

   d) REAL ZED [:= 0, B := 0, C := 0];

   e) INTEGER INTEGERB:= % 102; Correct statement as is.

   f) INTEGER [BETA] := "A";

   g) INTEGER BIGGE := [32767] ; or DOUBLE BIGGE := 35767;

   h) INTEGER SAM := 0;

   i) REAL [BEGINER] := 1.414;

   j) INTEGER [MIKE] := 1;

2. a) SUM := I + [INTEGER (FIXR (NUMBER))] / 2;

   b) Correct statement as is.

   c) Correct as is.

   d) TEMP := (I + J) – K <<SUBTRACT AND SAVE> [>];

   e) Correct statement as is.

   f) TOTAL := NUMBER [^] 2 + CRUNCHER [^] 2;

   g) TEMP := K * [(SUM := I + K + J)];

   h) SUM := M MOD [ ] 8 [;]

3. a) R1 = %74000

   b) R2 = "AD" = %040504

   c) R3 = "CE" = %041504

   d) R4 = %1

   e) R5 = %6

4. a) RA = "AE" = %40505

   b) RB = %0

   c) RC = %60

   d) RD = 0, RE = "BB" = %41102

5. BEGIN
   REAL ANS, X := 1.414, Z := .256;
   REAL PROCEDURE RAND; OPTION EXTERNAL;
   ANS := RAND * X – Z;
   END.

6. BEGIN
   Data Group
   Procedure Group
   Statements
   END.

7. BEGIN ⎫
   END. ⎬ a null program
         ⎭

8. A compound statement is actually a group of SPL/3000 statements (separated by semi-colons) surrounded by a BEGIN — END pair. The compiler treats this statement group as a single statement unit.

ANSWERS TO SECTION III EXERCISES

1. a) valid
   b) invalid
   c) valid
   d) valid
   e) invalid

2. LOOP: IF TESTWORD $>$ 0 THEN Y := A + B + C
                         ELSE
           IF TESTWORD = 0 THEN Y := 1./A + B + C
                         ELSE
                              Y := A * B * C;
           COUNT := COUNT +1;
           IF COUNT $<=$ 10 THEN GOTO LOOP;

3.

$NUMBER_8$	AS $INTEGER_{10}$	AS $LOGICAL_{10}$	TRUE/FALSE
177777	−1	65535	TRUE
000001	+1	1	TRUE
000377	+255	255	TRUE
000000	0	0	FALSE
177776	−2	65534	FALSE
100000	−32768	32768	FALSE
000003	+3	3	TRUE
000004	+4	4	FALSE
000005	+5	5	TRUE
000006	+6	6	FALSE

4. a) A LOR B LAND C LOR D

   b) NUMBER $<>$ 0 LAND NUMBER * SCALE $<=$ MAX

   c) X + Y / (Y − 5) * NUMBER LAND %77

d) A * B <> NOT D / C

5. a) Illegal statement — OR is legal only in branching conditions such
      IF, WHILE and DO. (IF B OR C THEN GO QUIT;).

   b) Legal statement.

   c) Illegal statement — AND is legal only in branching conditions (see answer a).

   d) Legal statement.

   e) Illegal statement — FIXR results in a type double result, the expression is therefore mixed.

   f) Illegal statement — NOT is illegal in arithmetic expressions.

   g) Illegal statement — # is invalid operator.

   h) Legal statement.

   i) Illegal statement — RESULT := C LOR (A := B); is legal however.

   j) Illegal statement — mixed expression.

   k) Legal statement.

6. a) Valid expression.

   b) Valid expression.

   c) Invalid expression — C MOD LOGICAL (D) + LOGICAL (F).

   *NOTE: an alternative may be INTEGER (C) MOD D + F. However in this case
   an integer rather than a logical divide is used.*

   d) Valid expression.

   *NOTE: The solutions presented as answers to these exercises are not necessarily
   unique nor are they always the optimum one. They are only examples
   of correct solutions.*

7. a) IF AGE <= 30 AND HT = 66 AND SEX AND JOB = 0 AND LANG = 2 THEN
      <<STATEMENT>>

   b) IF 30 <= ID <= 70 AND HT = 62 AND 90 <= WT <= 110 AND EYES = 2
      AND NOT SEX AND (JOB = 3 OR JOB = 7 OR JOB = 9) AND LANG = 7 THEN
      <<STATEMENT>>

   c) IF ID > 50 AND (21 <= AGE <= 24 OR 37 <= AGE <= 50) AND HT = 79
      AND WT > 190 AND (EYES = 1 XOR SEX) AND (JOB = 1 OR JOB = 8) AND
      (LANG = 4 OR LANG = 9) THEN <<STATEMENT>>

   d) IF 45 <= AGE <= 50 AND ((EYES = 1 AND SEX) OR (EYES = 2 AND NOT SEX))
      AND ((JOB = 4 AND LANG = 0) OR (JOB = 5 AND LANG = 1)) THEN
      <<STATEMENT>>

8. a) PTS := 0;
   IF AGE <= 30 THEN PTS := PTS + 1;
   IF HT = 66 THEN PTS := PTS + 1;
   IF SEX THEN PTS := PTS + 1;
   IF JOB = 0 THEN PTS := PTS + 1;
   IF LANG = 2 THEN PTS := PTS + 1;
   <<END OF TEST>>

   b) PTS := 0;
   IF 30 <= ID <= 70 THEN PTS := PTS + 1;
   IF HT = 62 THEN PTS := PTS + 1;
   IF 90 <= WT <= 110 THEN PTS := PTS + 1;
   IF EYES = 2 THEN PTS := PTS + 1;
   IF NOT SEX THEN PTS := PTS + 1;
   IF JOB = 9 OR JOB = 7 OR JOB = 3 THEN PTS := PTS + 1;
   IF LANG = 7 THEN PTS := PTS + 1;
   <<END OF TEST>>

   c) PTS := 0;
   IF ID > 50 THEN PTS := PTS + 1;
   IF 21 <= AGE <= 24 OR 37 <= AGE <= 50 THEN PTS := PTS + 1;
   IF HT = 79 THEN PTS := PTS + 1;
   IF WT > 190 THEN PTS := PTS + 1;
   IF EYES = 1 XOR SEX THEN PTS := PTS + 1;
   IF JOB = 1 OR JOB = 8 THEN PTS := PTS + 1;
   IF LANG = 4 OR LANG = 9 THEN PTS := PTS + 1;
   <<END OF TEST>>

   d) PTS := 0;
   IF 45 <= AGE <= 50 THEN PTS := PTS + 1;
   IF (EYES = 1 AND SEX) OR (EYES = 2 AND NOT SEX) THEN PTS := PTS + 1;
   IF (JOB = 4 AND LANG = 0) OR (JOB = 5 AND LANG = 1) THEN PTS := PTS + 1;
   <<END OF TEST>>

9. a) PTS := IF LANG = 0
```
   ┌ THEN IF SEX
   │     ┌ THEN IF 69 <= HT <= 75
   │     │    ┌ THEN IF 160 <= WT <= 210
   │     │    │    ┌ THEN IF 21 <= AGE <= 35
   │     │    │    │    ┌ THEN IF EYES = 2
   │     │    │    │    │    ┌ THEN IF JOB = 5
   │     │    │    │    │    │    ┌ THEN 7
   │     │    │    │    │    │    └ ELSE 6
   │     │    │    │    │    └ ELSE 5
   │     │    │    │    └ ELSE 4
   │     │    │    └ ELSE 3
   │     │    └ ELSE 2
   │     └ ELSE 1
   └ ELSE 0;
```
   b) PTS := IF NOT SEX
```
   ┌ THEN IF 21 <= AGE <= 25
   │    ┌ THEN IF 58 <= HT <= 65
   │    │   ┌ THEN IF 95 <= WT <= 120
   │    │   │   ┌ THEN IF EYES = 4
   │    │   │   │   ┌ THEN IF JOB = 3
   │    │   │   │   │   ┌ THEN IF LANG = 4
   │    │   │   │   │   │   ┌ THEN 7
   │    │   │   │   │   │   └ ELSE 6
   │    │   │   │   │   └ ELSE 5
   │    │   │   │   └ ELSE 4
   │    │   │   └ ELSE 3
   │    │   └ ELSE 2
   │    └ ELSE 1
   └ ELSE 0;
```

10.

CASE I

```
X := 0;
IF L1 ┌ THEN BEGIN
      │       IF L2
      │           THEN X := 1
      │       END
      └ ELSE X := 2;
```

CASE II

```
X := 0;
IF L1 THEN
        IF L2
            ┌─────── THEN X := 1
            └ ELSE X := 2;
```

The innermost THEN is paired with the closest following ELSE and pairing proceeds outward. This default association can be overridden by using BEGIN-END pairs to separate statements.

Table I

L_1	L_2	X
T	T	1
T	F	0
F	T	2
F	F	2

Table II

L_1	L_2	X
T	T	1
T	F	2
F	T	0
F	F	0

ANSWERS FOR SECTION IV EXERCISES

1. a) Valid

   b) Illegal real number

   c) Illegal operation

   d) Valid

   e) Illegal — strings not allowed.

   f) Valid

   g) Illegal — absolute value not allowed.

   h) Illegal — strings not allowed.

   i) Illegal — X and Y undefined when Z is declared.

2. a) Invalid — , in bounds.

   b) Valid

   c) Valid

   d) Invalid — no name.

   e) Invalid — only last array can be initialized.

   f) Invalid — equals instead of replacement (:=) and no quotes around string.

   g) Invalid — real constant cannot be used to initiate integer.

   h) Invalid — nested repeat groups not allowed.

   i) Invalid — reserved word cannot be used as identifier.

   j) Invalid — too many elements.

3. a) False

   b) False

   c) True

   d) False

   e) True

4. a) Yes.

   b) The second solution is more efficient. Use of an explicit subscript (even 0) always specifies use of the Index Register. Since the subscript is 0 in this case it makes no sense to cause the Index Register to be loaded with 0.

5. ARRAY DATA (–23 : 127);

6. EQUATE   BLANK = %40,

              RETURN = %15,

              LINEFEED = %12,

              NULL = 0;

7. EQUATE S = 7, L = 3;

8. a) IF 5 <= DATA <= 20 OR X <= DATA <= Y THEN X := X + 1;

   b) ARRAY ONE (0:7) := 1, 3, 5, 7, 9, 11, 13;
      ARRAY TWO (0:7) := 1, 3, 5, 7, 9, 11, 13;

   c) IF LINK.(0:2) = 1 AND LINK.(2:6) < 4 THEN LINK.(8:8) := 0;

9. DEFINE EQ = EQUATE #, INT = INTEGER #, ARY = ARRAY #, DEF = DEFINE #,
     LOG = LOGICAL #, ASSIGN = THEN #, OTHERWISE = ELSE #;

10. The address of the zero element is computed by subtracting the lower subscript from the first address assigned to the array.

$$\text{Address of zero element} = (\text{first word address of array}) - (\text{lower subscript value})$$
$$= (DB + 256) - 100$$
$$= DB + 156$$

To address arrays the computer hardware uses the address of the zero element to which is added the value of the index register.

For example:

$$\text{Effective address} = \text{Address of zero element} + \text{Index register}$$
$$= 156 + 100$$
$$= 256$$

11. EQUATE N = 99;
   INTEGER ARRAY AGE (0:N);
   INTEGER YOUNG, OLD, AVERAGE, SUM := 0, I;

   YOUNG := OLD := AGE; <<INITIALIZE>>
   FOR I := 0 UNTIL N DO
      BEGIN
         IF YOUNG > AGE (I) THEN YOUNG := AGE (I);
         IF OLD < AGE (I) THEN OLD := AGE (I);
         SUM := SUM + AGE (I);
      END;
   AVERAGE := SUM/(N + 1);

12. a) (N + 1)

   b) Yes.

13. a) 4 times

   b) 2 times (I = 9, which is odd or *true.*)

   c) Infinite loop. N + 1 <= 10 is true (−1) or false (0).
      In either case the value assigned to N will always be less than 10.

   d) 10 times

14. BEGIN
    INTEGER JUMPS := 1;
    REAL DISTANCE := 500.;

    WHILE .025 $\leq$= (DISTANCE := DISTANCE/2.) DO JUMPS := JUMPS + 1;

    END.

15. a) INDEX = 0 label is FIRST
    b) INDEX = 3 label is NEXT (index out of bounds)
    c) INDEX = –1 label is NEXT (index out of bounds)
    d) INDEX = 2 label is THIRD

16. a) True   c) True   e) False
    b) False   d) True   f) True

17. CASE TITLE OF
        BEGIN
            PREFIX := "MR ^ ^";
            PREFIX := "MRS ^";
            PREFIX := "MISS";
            <<NULL>>;
            PREFIX := "DR ^ ^"
        END;

ANSWERS TO SECTION V EXERCISES

1. a) Valid
   b) Valid
   c) Invalid — too many characters.
   d) Invalid — only one array can be initialized.
   e) Invalid — real number not allowed.
   f) Valid
   g) Valid
   h) Valid
   i) Warning — too many characters.
   j) Valid

2.

WORD	
—	A

ARRAY NULL	
A	—
—	—

TOS	
—	A

3. WORD.(0:8) := A;   or   TOS.(0:8) := A;
   WORD.(8:8) := B;   TOS.(8:8) := B;
                           WORD := TOS;

   There are many other solutions.

4. None — INDEX is equivalenced to the hardware Index Register.

5.

	Byte Address	Word Address	Final Byte Address
Address 1	%1023	%411	%1022
Address 2	%1460	%630	%1460
Address 3	%2577	%1277	%2576

6. a) 1 word
   b) 1 word
   c) 1 word
   d) 1 word
   e) Invalid — No words.

7. a) no
   b) no
   c) yes
   d) no
   e) yes

8. TOTAL := (37 −1 + 144) * 3;
   TOTAL := 540;
                 Result is not truncated since integer arithmetic is used for bytes.

9. INTEGER POINTER IPNTR := @SAM(1);

10. a) The contents of NUMBER(0) are incremented by one to a value of 2.

    b) The contents of NUMBER(2), NUMBER(3), NUMBER(4), and NUMBER(5) are added together and the sum (18) replaces the previous contents of NUMBER(1).

    c) The contents of the pointer location NPNTR are increased by a value of 7. NPNTR now points to NUMBER(7);

    d) The contents of NPNTR are restored to the address of NUMBER(0);

    e) The address of NUMBER(9) is stored in the location NUMBER(0);

11. BUFF := %125252;
    MOVE BUFF(1) := BUFF, (99);

12. MOVE OUTBUF := INBUF, (128), 1;
    @ INPNTR := TOS;
    @ OUTPNTR := TOS;

13. SOURCEBUF (80) := "*";
    MOVE DESTBUF := SOURCEBUF WHILE A;

14. SOURCEBUF (80) := "*";
    MOVE DESTBUF := SOURCEBUF WHILE A, 1;
    NUMBER := IF > THEN TRUE ELSE FALSE;
    COUNT := TOS; <<SAVE LAST DESTINATION ADDRESS>>
    COUNT := COUNT − @ DESTBUF;

15.

S − 1	%1004	(DB relative address)
S − 0	.E	

16. IF DATA = "END OF INPUT" THEN FLAG := TRUE;

ANSWERS TO SECTION VI EXERCISES

1.

(40200)	PREV Q →		PREVIOUS STACK MARKER
			LOCAL VARIABLES
(40206)	PREV S →		PARAMETERS (3 WORDS)
		1	INDEX
		124	REL. P (P–PB + 1)
		4076	STATUS REG.
(40212)	Q, S →	12	

2. DL = 30000   PB = 002000
   DB = 30000 P = 002140
   Q = 31000   PL = 002500
   S = 31004 X = 000001   (31000) Q →   PREVIOUS STACK MARKER
   Z = 32000   STATUS = 002037

   (31004) S →   TWO WORDS LEFT ON STACK BY EXIT 1.

3. a) INTEGER X, Y, Z;
      X was not specified.

   b) PROCEDURE FIXIT (A, B, X);
      VALUE A, B;
      REAL A, B, X;
      OPTION FORWARD;
      Option part must not precede specification part.

   c) VALUE X, Y; REAL X, Y;
      Semi-colon was missing between Y and REAL; note that BEGIN-END are not required
      in a one statement procedure.

D-15

4. PROCEDURE MOVEWORDS (FROM, DEST, COUNT);
    VALUE COUNT; INTEGER COUNT;
    LOGICAL ARRAY FROM, DEST;
    BEGIN
        MOVE DEST := FROM, (COUNT);
    END; <<MOVEWORDS>>

5. a)

VARIABLE	MAIN	PROCEDURE ALPHA	PROCEDURE BETA	TYPE
X	✓	✓	✓	GLOBAL
Y	✓	✓	✓	GLOBAL
A		✓		LOCAL
B		✓		LOCAL
I			✓	LOCAL
J			✓	LOCAL
K			✓	LOCAL

b)

VARIABLE	MAIN	PROCEDURE THING	SUBROUTINE SAM	SUBROUTINE FIXIT	TYPE
A	✓	✓	✓	✓	GLOBAL
B	✓	✓	✓	✓	GLOBAL
I		✓	✓		LOCAL
J		✓	✓		LOCAL

6. SUBROUTINE SCL (VREAL, SCALE);
   VALUE VREAL, SCALE;
   REAL VREAL, SCALE;

      X := INTEGER (FIXR(VREAL * SCALE));

7. REAL PROCEDURE RAD (DEGREE);
    VALUE DEGREE;
    REAL DEGREE;

    RAD := DEGREE * .017453;

8. a) REAL SUBROUTINE ANS (A, B);
      VALUE A, B; REAL A, B;
      BEGIN
          (INTEGER X = X;)
          IF X > 0 THEN ANS := A + B
              ELSE ANS := A - B;
      END;
  No local declarations are allowed in a subroutine.

  b) SUBROUTINE EVALUATE (DATA, GOODRANGE);
     REAL DATA, GOODRANGE;
     (OPTIONAL EXTERNAL;)

  Option External is not valid in a subroutine. (Subroutines may not be compiled without a main program or procedure to contain them.)

c) SUBROUTINE MOVER (SOURCE, DEST, COUNT);
   BEGIN
   ARRAY SOURCE, DEST;
   INTEGER COUNT;
         MOVE DEST := SOURCE, (COUNT);
   END;

   The subroutine parameter specifications are incorrectly placed. They should be located before the BEGIN.

9. Point (A) RESULT is 6.

   Point (B) RESULT is 60.

10. No value was stored in the function name (VERIFY). The default return is 0 or logical FALSE.

11. PROCEDURE COMPUTE (X, Y);
    VALUE X;
    INTEGER X, Y;
        Y := X MOD 8;

    The parameter specifications were incorrectly placed following BEGIN. Since this is a single statement procedure, BEGIN and END are optional.

ANSWERS TO SECTION VII EXERCISES

1. (base)

    a)   2 %(2)1111111011000101

    b)   4 %(4)33323011

    c)   8 %177305

    d) 16 %(16)FEC5

2. a) 0 000 000 101 101 111

    b) 0 000 011 110 010 111

    c) 0 000 111 111 100 010

    d) 1 010 110 001 101 111

3. a) %102770
        %115372
        %062377

    b) [3/4, 5/5, 4/%17, 4/%10]
        [5/%23, 8/%137, 3/2]
        [8/100, 8/255]

4. a) [8/%12, 8/%15]

    b) [8/7, 8/3]

5. DATA := IF (DATA LAND [2/3, 3/0, 1/1, 3/0, 3/7, 4/0]) = [2/1, 3/0, 1/0, 3/0, 3/5, 4/0]
              THEN TRUE ELSE FALSE;

6.   *TO*

FROM	LONG	REAL	DOUBLE	INTEGER	LOGICAL	BYTE
LONG	—	REAL				
REAL	LONG	—	FIXR FIXT			
DOUBLE	LONG	REAL	—	INTEGER	LOGICAL	
INTEGER		REAL	DOUBLE	—	LOGICAL	BYTE
LOGICAL		REAL	DOUBLE	INTEGER	—	BYTE
BYTE		REAL	DOUBLE	INTEGER	LOGICAL	—

7. SHORT := INTEGER (FIXR (REAL (DATA)));

8. REAL PI := 3.14159;
    LONG EULERS'CONSTANT := .57721566490 L;
    INTEGER RADIUS := 3959;
    DOUBLE GALLONS := 18627235 D;
    LONG WINN'SCONSTANT := −4.537269537L23;
    REAL PLANCK := 6.62559E−27;

9.

TYPE	# OF BITS	LARGEST_{10}	SMALLEST_{10}
BYTE	8	255	0
LOGICAL	16	65535	0
INTEGER	16	+32767	-32768
REAL	32	$(6.9 \text{ DIGITS})+10^{77}$	$(6.9 \text{ DIGITS})-10^{77}$
DOUBLE	32	2147483647	-2147483648
TRIPLE	ILLEGAL TYPE	ILLEGAL TYPE	ILLEGAL TYPE
LONG	48	$(11.7 \text{ DIGITS})+10^{77}$	$(11.7 \text{ DIGITS})+10^{77}$

10. PROCEDURE AVERAGE (NUMBRS, COUNT);
    VALUE COUNT;
    INTEGER ARRAY NUMBRS;
    INTEGER COUNT;
    BEGIN
    INTEGER INDX = 0,
        X = X;
    X := -1;
    WHILE (X := X + 1) $\leq$ COUNT DO INDX := INDX + NUMBRS (X);
    INDX := INDX/COUNT;
    END;

11.

	DATA LABELS	DATA STORAGE	
	1	36	HELLO'MESSAGE
39 WORDS TOTAL =	1	0	INBUF
	1	0	SCRATCH

12. a) Valid

    b) Valid

    c) BYTE ARRAY ALPHA (*) = NUMBER;

    d) LOGICAL ARRAY DATA (0:6) = PB := "DATA GOES HERE";

    e) REAL ARRAY BIGNO (0:N) = DB;

    f) Valid

    h) INTEGER ARRAY TOTAL (*) = NUMBER;

    i) BYTE ARRAY STRINGCHAR (0:3) = Q;

    j) ARRAY SAM (1:7) = DB := 0,1,2,3,4,5,6;

13. REAL PROCEDURE SUMIT (UPPER);
        VALUE UPPER;
        INTEGER UPPER;
        BEGIN
            REAL ARRAY TOTAL (0 : UPPER);
            INTEGER I, SUMIT := 0;
            GETDATA (TOTAL, UPPER + 1);
            FOR I := 0 UNTIL UPPER DO
            SUMIT := SUMIT + TOTAL (I);
        END;

14. PROCEDURE TIME'STAMP (TIME);
    VALUE TIME;
    DOUBLE TIME;
    BEGIN
        OWN DOUBLE ARRAY ELAPSED'TIME (0:99);
        OWN DOUBLE START'TIME;
        OWN INTEGER SUB := 0;
        OWN LOGICAL PAIR := FALSE;

        IF PAIR THEN BEGIN <<START & END TIME RECEIVED>>
                    ELAPSED'TIME (SUB) := TIME — START'TIME;
                    SUB := SUB + 1;
                END
            ELSE START'TIME := TIME; <<SAVE START TIME>>
        PAIR := NOT PAIR; <<COMPLEMENT START-END SWITCH>>
        IF SUB = 100 THEN BEGIN
                    WRITETAPE (ELAPSED'TIME); <<output data>>
                    SUB := 0;
                END;
    END <<TIME'STAMP>>;

15. BYTE POINTER   SORC = S – 0,
                   DEST = S – 1;
                   MOVE BUFB := BUFA WHILE A, 0;
                   DEST := SORC;

16. TOS := ZERO; <<PUSH ZERO FOR INTEGER RESULT>>
    TOS := @ STRING; <<PUSH BYTE ARRAY ADDRESS>>
    TOS := BIGWORD; <<PUSH DOUBLE WORD VALUE>>
    TOS := NUMBER; <<PUSH REAL VALUE>>

17. a) 1 word

    b) I = –11
       L = TRUE(177765)
       B = %377

ANSWERS FOR SECTION VIII EXERCISES

1. a)

S – 4	4
S – 3	1000
S – 2	0
S – 1	2
S – 0	4

   b)

S – 3	0
S – 2	1000
S – 1	3
S – 0	4

2. BR DB + 40;
    RSW, DUP;
    MVBW S;
    ORI MASK;

3. RSW
    EXF
    XEQ
    DPF
    PSHR

INDEX

NOTE: Special symbols are listed at the end with cross references to alphabetic entries.

Index of Special Characters

READER COMMENT SHEET

HP 3000 Computer Systems
Systems Programming Language
Textbook

30000-90025   **Jan 1977**

We welcome your evaluation of this manual. Your comments and suggestions help us improve our publications. Please use additional pages if necessary.

Is this manual technically accurate?   Yes ☐   No ☐   (If no, explain under Comments, below.)

Are the concepts and wording easy to understand?   Yes ☐   No ☐   (If no, explain under Comments, below.)

Is the format of this manual convenient in size,   Yes ☐   No ☐   (If no, explain or suggest improvements
arrangement, and readability?   under Comments, below.)

Comments:

_____

FROM:

Name   _____

Company   _____

Address   _____

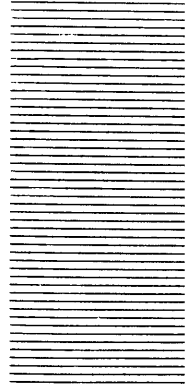   _____

                _____

FOLD   FOLD

BUSINESS REPLY MAIL

No Postage Necessary if Mailed in the United States. Postage will be paid by

Publications Manager
Hewlett-Packard Company
General Systems Division
5303 Stevens Creek Boulevard
Santa Clara, California 95050

FOLD   FOLD

HEWLETT
PACKARD