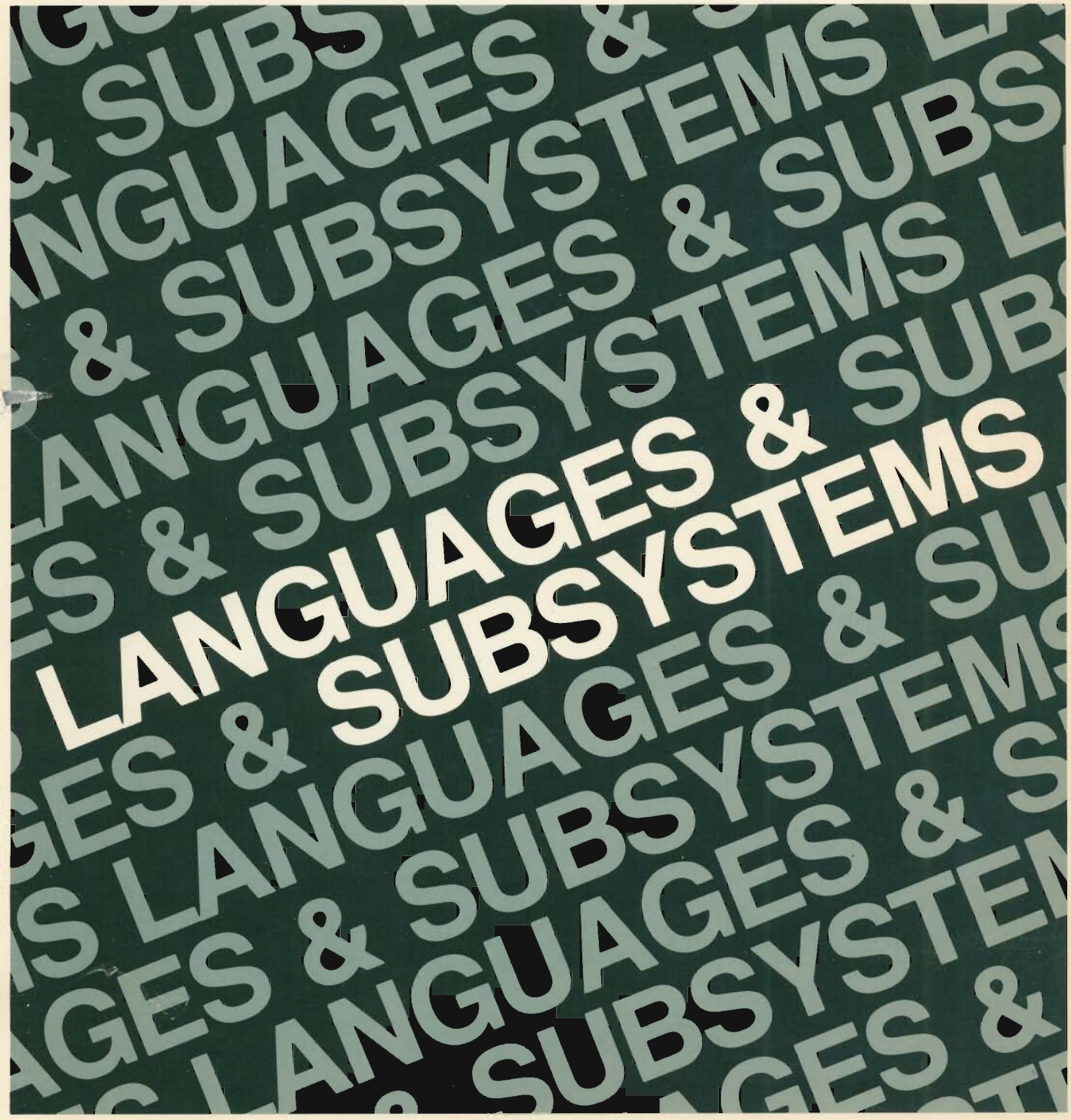
HP 3000 Computer Systems

HEWLETT PACKARD

Systems Programming Language
reference manual



HP Computer Museum
www.hpmuseum.net

For research and education purposes only.

HP 3000 Computer System

Systems Programming Language

Reference Manual

HEWLETT PACKARD

19420 Homestead Road, Cupertino, California 95014

Part No. 30000-90024
Product No. 32100A

Printed in U.S.A. 2/84

NOTICE

The information contained in this document is subject to change without notice.

HEWLETT-PACKARD MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Hewlett-Packard shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance or use of this material.

Hewlett-Packard assumes no responsibility for the use or reliability of its software on equipment that is not furnished by Hewlett-Packard.

LIST OF EFFECTIVE PAGES

The List of Effective Pages gives the date of the current edition and of any pages changed in updates to that edition. Within the manual, any page changed since the last edition is indicated by printing the date the changes were made on the bottom of the page. Changes are marked with a vertical bar in the margin. If an update is incorporated when an edition is reprinted, these bars are removed but the dates remain. No information is incorporated into a reprinting unless it appears as a prior update.

First Edition	Jun 1976
Second Edition	Sep 1976
Third Edition	Feb 1984

PRINTING HISTORY

New editions are complete revisions of the manual. 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 of the manual changes only when a new edition is published. When an edition is reprinted, all the prior updates to the edition are incorporated. No information is incorporated into a reprinting unless it appears as a prior update. The edition does not change.

The software product part number printed alongside the date indicates the version and update level of the software product at the time the manual edition or update was issued. Many product updates and fixes do not require manual changes, and conversely, manual corrections may be done without accompanying product changes. Therefore, do not expect a one to one correspondence between product updates and manual updates.

First Edition	Jun 1976	
Second Edition	Sep 1976	
Update #1 Incorporated	Dec 1976	
Update #2	Feb 1977	
Update #2 Incorporated	Dec 1977	
Third Edition	Feb 1984	32100A.08.04

PREFACE

This publication is the reference manual for the HP 3000 Computer System Systems Programming Language (SPL).

This publication contains the following sections:

Section I — is an introduction to SPL source format and the HP 3000 Computer System.

Section II — describes SPL data storage formats, SPL constants, identifiers, arrays, and pointers.

Section III — describes the global declarations.

Section IV — describes arithmetic and logical expressions, assignment, MOVE, and SCAN statements.

Section V — describes the various program control statements including GO TO,DO, WHILE, FOR, IF, CASE, procedure call, subroutine call, and RETURN statements.

Section VI — describes the machine level constructs including the ASSEMBLE statement (to use any machine instruction), the DELETE statement, the PUSH statement (for saving registers), and the SET statement (for setting registers).

Section VII — describes the subprogram units (procedures, intrinsics, and subroutines) and the local declarations.

Section VIII — discusses some of the more common MPE intrinsics for performing input/output.

Section IX — discusses the various compiler commands.

Section X — discusses the MPE commands used to compile, prepare, and execute an SPL source program together with some introductory material on using the Segmenter.

Appendix A — lists the ASCII character set.

Appendix B — lists the reserved words in SPL.

Appendix C — describes how to build your own intrinsic file.

Appendix D — lists the MPE Operating System intrinsic procedures.

Appendix E — lists the diagnostic messages which can be generated by the SPL compiler.

Appendix F — explains how to call SPL from other languages.

Other publications which should be available for reference when using this manual are:

Systems Programming Language Textbook (30000-90025)

MPE Commands Reference Manual (30000-90009)

MPE Intrinsics Reference Manual (30000-90010)

MPE Segmenter Reference Manual (30000-90011)

Machine Instruction Set Reference Manual (30000-90022)

System Reference Manual (30000-90020)

Compiler Library Reference Manual (30000-90028)

EDIT/3000 Reference Manual (03000-90012)

CONVENTIONS USED IN THIS MANUAL

NOTATION	DESCRIPTION
[ ]	An element inside brackets is *optional*. Several elements stacked inside a pair of brackets means the user may select any one or none of these elements. Example: [A / B] user may select A or B or neither
{ }	When several elements are stacked within braces the user **must** select one of these elements. Example: {A / B / C} user must select A or B or C.
italics	Lowercase italics denote a parameter which must be replaced by a user-supplied variable. Example: CALL *name* *name* one to 15 alphanumeric characters.
underlining	Dialogue: Where it is necessary to distinguish user input from computer output, the input is underlined. Example: NEW NAME? ALPHA1
superscript C	Control characters are indicated by a superscript C Example: Y^C
return	*return* in italics indicates a carriage return
linefeed	*linefeed* in italics indicates a linefeed
. . .	A horizontal ellipsis indicates that a previous bracketed element may be repeated, or that elements have been omitted.

CONTENTS

CONTENTS (continued)

CONTENTS (continued)

ILLUSTRATIONS

TABLES

SPL STRUCTURE

SECTION I

1-1. INTRODUCTION TO SPL

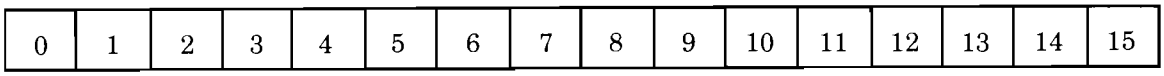
SPL (Systems Programming Language for the HP 3000 Computer System) is a high-level, machine dependent programming language that is particularly well suited for the development of compilers, operating systems, subsystems, monitors, supervisors, etc.

SPL has many features normally found only in high-level languages such as PL/I or ALGOL: free-form structure, arithmetic and logical expressions, high-level statements (IF, FOR, GOTO, CASE, DO-UNTIL, WHILE-DO, MOVE, SCAN, procedure call, assignment, and compound statements), recursive procedures and subroutines, and variables and arrays of six data types (byte, integer, logical, double integer, real, and long real). In addition, IF, FOR, CASE, DO-UNTIL, and WHILE-DO statements can be indefinitely nested within each other and themselves. These features significantly reduce the time required to write programs and make them much easier to read and update.

In addition, SPL provides machine-level constructs that insure the programmer has complete control of the machine when he needs it. These constructs include direct register references; branches based on actual hardware conditions; bit extracts, deposits, and shifts; delete statements; register push/set statements; and an ASSEMBLE statement to generate any sequence of machine instructions.

1-2. CONVENTIONS

In the HP 3000, the bits of a word are numbered from left to right starting with bit 0. Thus, the sign, or most significant, bit of a single word is bit 0 and the least significant bit is bit 15.



1-3. SOURCE PROGRAM FORMAT

An SPL source program can contain both program text and compiler commands in 80 column records. Program text is entered in free format in columns 1-72. A statement is terminated with a semicolon (;) and may continue to successive lines without an explicit continuation indicator. Statement labels are identifiers followed by a colon (:) preceding the statement. For example,

```
START: SCAN BUF WHILE TEST;
```

Any compilation is bracketed by BEGIN and END statements. A period is required after the final END. For example,

```
BEGIN
  INTEGER I;
  I:= 2*373+ 275;
END.
```

Compiler commands are denoted by a $ in column 1 and may be interspersed with program text lines. However, unlike program text lines, compiler commands which are to be continued must contain an ampersand (&) as the last non-blank character of the line. If using EDIT/3000 to enter text, you must explicitly enter a space following the ampersand and before pressing return. In addition, the continuation lines must contain a $ in column 1. For example,

```
$CONTROL LIST,SOURCE,WARN,MAP,&
$CODE,LINES=36
```

A compiler command line must never be separated from its continuation line by a program text line. Refer to section IX for a discussion of all the SPL compiler commands.

1-4. DELIMITERS

Blanks are always recognized as delimiters in SPL, except within character strings (see paragraph 2-17 for the format of string constants). Therefore, blanks cannot be embedded in the following items:

Reserved words (see Appendix B).
Identifiers

:=	assignment
<<	start of a comment
>>	end of a comment

Special characters can also act as delimiters:

Punctuation : ; , . "
Relational Operators = < >
Parentheses ()
Operators + − * /^
Brackets [ ]

1-5. COMMENTS

A comment is used to document a program but has no effect upon the functioning of the program itself;that is, a comment does not generate any code.

Comments may take either of the following forms in SPL:

Format 1: COMMENT [*comment*];

Format 2: <<[*comment*]>>

EXAMPLES:

```
<<comment>>
COMMENT CONTROL: MESSAGE;
<<This is a comment>>
!This is a comment
COMMENT
    THIS
    IS
    A
    COMMENT
    ;
```

where

comment
is any sequence of ASCII characters except a semicolon in Format 1 and >> in Format 2. The ASCII character set is listed in Appendix A.

Format 1 is equivalent to a null statement and can be used anywhere a statement or declaration is expected. Format 2 can be used anywhere in a program except in an identifier.

The characters within a comment are ignored by the compiler; they are not upshifted (changed to uppercase) if lowercase.

When the special character '!' is encountered outside a comment, define, or string, the rest of the source line following the exclamation point will be regarded as a comment.

1-6. PROGRAM STRUCTURE

SPL is a block structured language which takes advantage of the virtual memory scheme of the HP 3000 to provide program segmentation as a user option. Thus, by using procedures and segmentation, the programmer can organize his program such that the entire program does not have to reside in memory at the same time. The system automatically gets procedure segments from auxiliary memory and loads them into main memory when necessary.

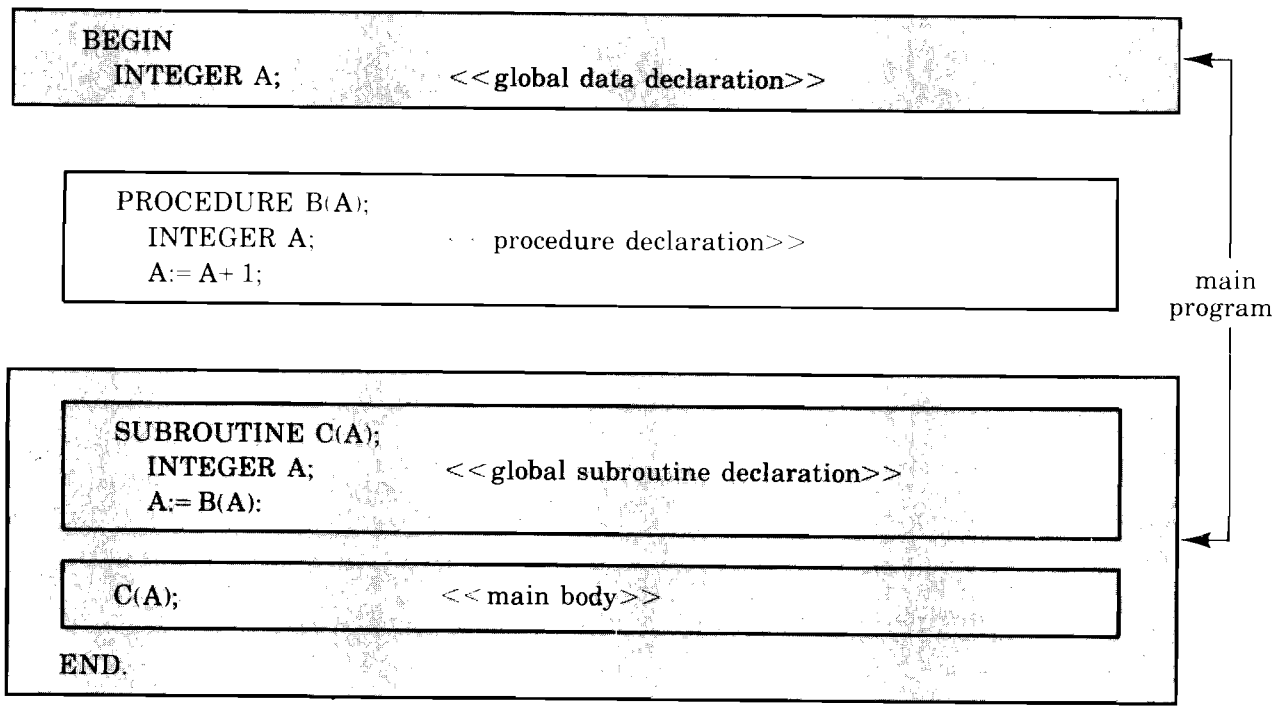
Additionally, SPL uses the stack architecture of the HP 3000 to handle both global and local variables. Global variables may be referenced anywhere in the program except in procedures where a local variable has the same identifier. Local variables are allocated memory locations upon entering a procedure and can only be referenced within the procedure in which they are declared. The memory locations assigned to local variables are released when the procedure is exited. When one procedure calls another procedure, the local variables of the calling procedure are not available to the called procedure unless they are passed as parameters; however, their memory locations are saved so that upon returning to the original procedure, the local variables contain the same values as before the procedure call.

Similarly, both global and local subroutines are allowed in SPL. However, unlike global variables, global subroutines can only be called within the main program and not within a procedure. Local subroutines may be called only within the procedure in which they are declared.

The SPL compiler accepts either complete programs or subprograms as source input. A program consists of both declarations and a main body of executable statements. The declaration portion may contain variable, procedure, intrinsic, and/or global subroutine declarations.

A subprogram consists of only the declaration portion and does not contain a main body. In a subprogram compilation, global declarations (that is, declarations for variables which can be referenced throughout the entire program) do not allocate any space and global subroutines are ignored if present. A subprogram compilation generates code for procedures and local subroutines only and must be linked to a separately compiled main program before being executed.

For example,



1-7. PROGRAM

A program is an organized collection of declarations and statements designed to solve a specific problem. A main program consists of global data declarations and subroutines and a main body.

The form for a program is:

BEGIN
[*global data declarations*]
[*procedures/intrinsics*]
[*global-subroutines*]
[*main-body*]
END.

where

global data declarations
are statements defining the attributes of the global identifiers used in the program (see section III).

procedures/intrinsics
are statements which define all the procedures and intrinsics used in the program (see section VII). A procedure definition includes data declarations for parameters and local variables followed by the executable statements of the procedure.

global-subroutines
are the subroutines used by the main program.

main-body
is a sequence of statements separated by semicolons

statement [;...;*statement*]

statement
is an executable statement.

The program elements must be in the order shown above.

For example,

```
BEGIN
    INTEGER A:=0,B,C:=1;              <<global data declaration>>
    PROCEDURE N(X,Y,Z);               <<procedure>>
        INTEGER X,Y,Z;                <<local data declaration>>
        X:=X*(Y+Z);
    FOR B:=1 UNTIL 20 DO              <<main program>>
        N(A,B,C);
END.
```

1-8. SUBPROGRAM

A subprogram is a portion of a program which can be compiled by itself but must be linked to a main program for execution. A $CONTROL SUBPROGRAM compiler command is used before the subprogram text to put the compiler in subprogram mode. See section IX for the compiler commands used to link a subprogram to a main program for execution.

The form of a subprogram is the same as a program except that a subprogram does not have a main body.

The form for a subprogram is:

BEGIN
[*global data declarations*]
[*procedures/intrinsics*]
[*global-subroutines*]
END.

where

global data declarations
are statements defining the attributes of the global identifiers used in the program (see section III).

procedures/intrinsics
are statements which define all the procedures and intrinsics used in the program (see section VII). A procedure definition includes data declarations for the parameters and local variables followed by the executable statements of the procedure.

global-subroutines
are the subroutines used by the main program. The *global-subroutines* can be omitted since the compiler ignores them in subprogram compilations.

For example,

```
$CONTROL SUBPROGRAM
BEGIN
    INTEGER N,M,O; <<does not allocate space>>
    EQUATE A:= 101, B:= 202;
    PROCEDURE C;
        BEGIN
          .
          .
          .
        END;
    PROCEDURE D;
        BEGIN
          .
          .
          .
        END;
END.
```

1-9. INTRODUCTION TO HP 3000 HARDWARE CONCEPTS

A 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 program several times, each execution is a distinct process. A process consists of a code domain (the machine instructions of the program) and a data area called a "stack." The code and data in the HP 3000 are always separated logically. The code may always be shared, but the data stack cannot. The MPE Operating System schedules and dispatches a process for execution. See the *MPE General Information Manual* for a further discussion of processes and the stack.

1-10. CODE SEGMENTS

All machine instructions within the HP 3000 are organized into variable length segments accessed through a hardware-known table called the Code Segment Table (CST). 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. Segments are brought from disc into main memory as needed. A process can execute *only one* code segment at a time. The process "escapes" from its current code segment by executing a Procedure Call (PCAL) instruction. A PCAL can reference procedures in different code segments from the current one and cause control to be transferred to a different code segment. A PCAL instruction is generated by either a function designator (see paragraph 4-6) or a procedure call statement (see paragraph 5-8).

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

1. PB — Program Base register. Contains the absolute address of the starting location of the segment in main memory.
2. PL — Program Limit register. Contains the absolute address of the last location of the code segment.
3. 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.

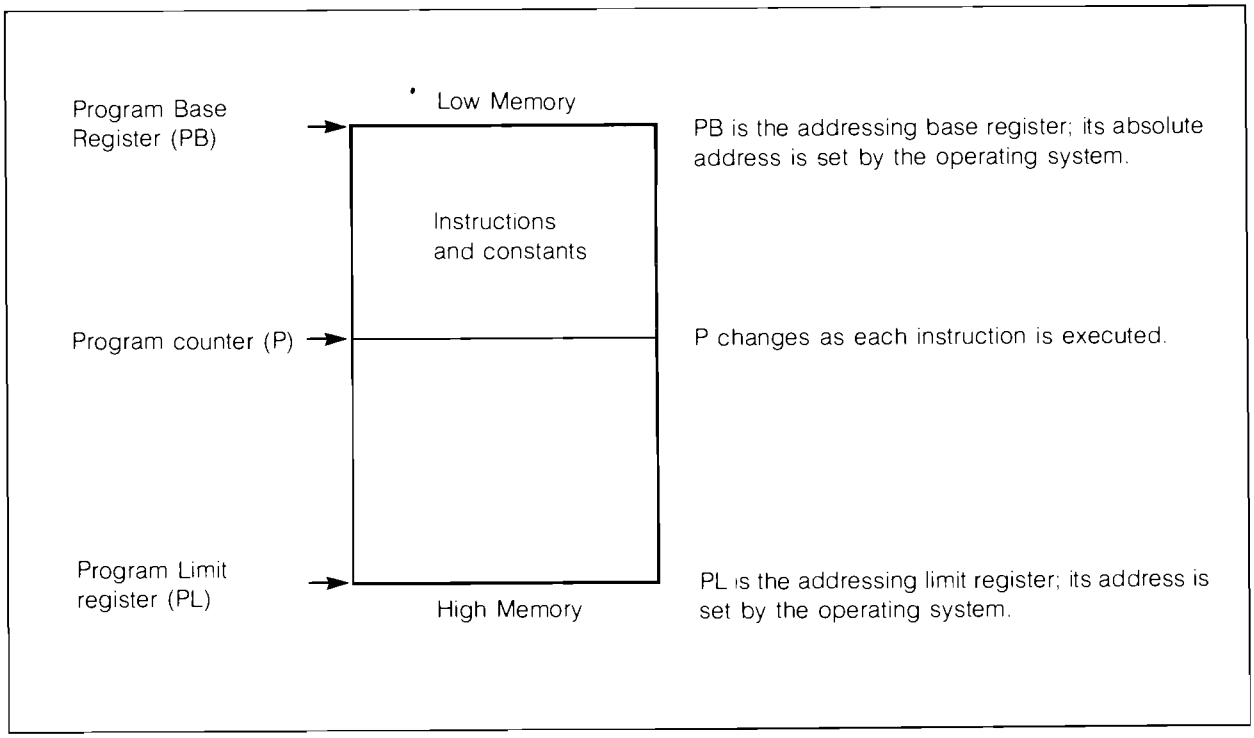


Figure 1-1. Code Segment Registers

Code segmentation is controlled by using the SEGMENT parameter on $CONTROL commands (see section IX). The segment name stays in effect until another segment name is specified. For procedures, the $CONTROL SEGMENT command must precede the procedure declaration of the first procedure in the segment. If a new segment is to be specified for the main program, the $CONTROL SEGMENT command follows the procedure and intrinsic declarations and precedes the global subroutines and main body. Global subroutines must be in the same segment as the main body. See figure 1-2 for a sample SPL program which has two procedures in one segment and a global subroutine with the main body in another.

```
00000 0    $CONTROL USLINIT,MAIN=MAINLINE
00000 0    BEGIN
00000 1      INTEGER LENGTH,TIME;
00000 1      ARRAY BUFFER(0:35);
00000 1      INTRINSIC PRINT,READ;
00000 1
00000 1    $CONTROL SEGMENT=PROC'A'SEG
00000 1    PROCEDURE PROC'A(LEN);
00000 1    VALUE LEN;
00000 1    INTEGER LEN;
00000 1    PRINT (BUFFER,-LEN,0);
00000 1
00000 1    PROCEDURE PROC'B(LEN);
00000 1    VALUE LEN;
00000 1    INTEGER LEN;
00000 1    PRINT(BUFFER,-LEN,%320);
00000 1
00000 1    $CONTROL SEGMENT=MAINLINESEG
00000 1
00000 1    SUBROUTINE READ'A'LINE;
00000 1    LENGTH:=READ(BUFFER,-72);
00006 1
00006 1    <<     START OF MAINLINE    >>
00006 1
00006 1    LOOP:
00006 1
00006 1    READ'A'LINE;
00010 1    IF LENGTH <> 0 THEN
00013 1       BEGIN
00013 2       IF ((TIME:=TIME+1) MOD 2)=0 THEN PROC'A(LENGTH)
00022 2                                   ELSE PROC'B(LENGTH);
00026 2       GO TO LOOP;
00027 2       END;
00027 1    END.


      MAINLINESEG        0
         NAME             STT  CODE ENTRY SEG
         MAINLINE           1     0     6
         READ               2                 ?
         PROC'A             3                 1
         PROC'B             4                 1
         TERMINATE'         5                 ?
         SEGMENT LENGTH          40
      PROC'A'SEG         1
         NAME             STT  CODE ENTRY SEG
         PROC'B             1     0     0
         PRINT              3                 ?
         PROC'A             2     6     6
         SEGMENT LENGTH          20
```

Figure 1-2. Sample Segmented Program

1-11. DATA SEGMENTS

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:

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

2. 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:

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

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

3. 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 words may be kept in hardware registers.

The relationship of the five data addressing registers is shown in figure 1-3. Each process is also described by a status register that contains 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.

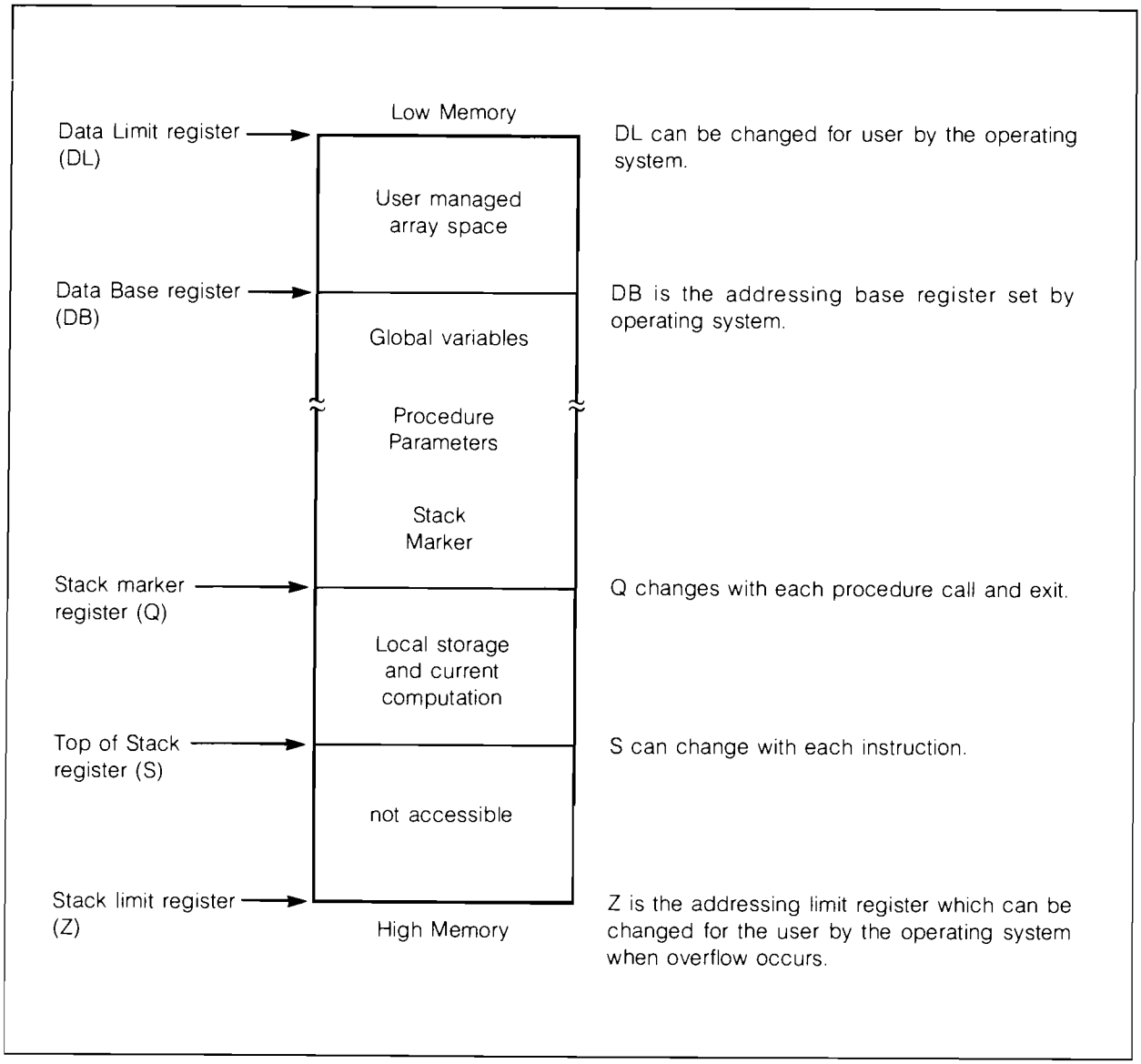


Figure 1-3. Data Stack Registers

Instructions are provided to access all regions indicated in this diagram except S to Z. The four top-of-stack registers are not shown.

In the HP 3000, memory reference instructions specify an address relative to one of the hardware registers. Each register has its own addressing range as indicated below:

	+	−
P register	255	255
DB register	255	*****
Q register	127	63
S register	*****	63

Note that the DB register cannot be directly addressed with a negative range and that the S register cannot be addressed with a positive range. The DB negative area can be accessed through indirect addressing and indexing. The S positive area is undefined since S points to the top of the stack.

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).

The basic addressing mode in the HP 3000 is word addressing (one word = 16 bits); however, there are also instructions to load and store bytes (half words — 8 bits) and doublewords (32 bits).

Many HP 3000 instructions use the top of the stack (the absolute address in the S register) as an implicit operand. For example, the ADD instruction always uses the values in S−0 and S−1 for its operands. 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.

1-12. PROCEDURES

A procedure is a self-contained section of code which is called to perform a function. Some of the features of procedures are:

- Procedures can be passed parameters (either call-by-value or call-by-reference).
- Procedures can declare local variables and reference global variables.
- Procedures can return a value.
- Procedures can call themselves.
- Procedures can be called from either procedures or the main body.
- Procedures can have local subroutines (sections of code which can only be called from within the procedure).

Procedure declarations precede the main body of the program and contain the local declarations and the procedure body.

For example, a procedure to compute N factorial is

```
INTEGER PROCEDURE FACT(N); VALUE N; INTEGER N;
BEGIN
    FACT:= IF N=0 THEN 1
                  ELSE N*FACT(N-1);
END;
```

For a complete explanation of procedure declarations, see section VII.

1-13. SUBROUTINES

An SPL subroutine is a simpler and less powerful section of code than the procedure. Subroutines can have parameters, can be typed functions and can be called recursively. A subroutine is called with an

SCAL instruction instead of a PCAL instruction. SCAL does not provide a 4-word stack marker to save the environment; therefore,

- Values in the Q and index registers remain unchanged.
- A PB-relative return address is placed on the top of the stack.
- Subroutines cannot have local variables.
- Subroutines must be located in the same segment as the caller since the SCAL and SXIT instructions 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.
- Subroutines can be declared within procedures and can reference procedure-local variables.

Global subroutines can be called only within the main body. Global subroutine declarations must appear after the procedure and intrinsic declarations.

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 executable statements of the procedure body. For a complete description of subroutine declarations, see section VII.

1-14. INTRINSICS

An intrinsic is a procedure which has previously been defined, either as part of the MPE Operating System or in a user's own intrinsic file. The advantage of using intrinsics is that you do not have to include the complete procedure in your program, but merely declare the name of the intrinsic in an intrinsic declaration.

MPE intrinsics are available to:

- Access and alter files.
- Manage program libraries.
- Obtain date, time, and accounting information.
- Determine job status.
- Determine device status.
- Obtain device file information.
- Transmit messages.
- Insert comments in command stream.
- Perform ASCII/binary number conversion.
- Perform input/output on job/session standard devices.
- Obtain system timer information.
- Obtain the user's access mode and attributes.
- Search arrays and format parameters.
- Execute MPE commands programmatically.

Intrinsics must be declared with an intrinsic declaration (See section VII). Appendix C shows how to build your own intrinsic file. Appendix D contains a list of the MPE intrinsics. Refer to the *MPE Intrinsics Reference Manual* for a complete description of the system intrinsics.

1-15. COMPOUND STATEMENTS

BEGIN and END are used as a delimiting pair and are matched much like parentheses. Within the body of a main program or a procedure, a BEGIN-END pair can be used to combine several statements into one compound statement. Compound statements are useful in IF, FOR, CASE, DO-UNTIL, and WHILE-DO statements.

The form of a compound statement is:

```
BEGIN
[statement;...;statement]
END
```

where

statement
is any SPL executable statement (including compound statements).

For example,

```
IF A<B THEN
    BEGIN
        A:=B;
        B:=D;
        E:=F
    END;
```

Note that a semicolon is not required before the END statement. If it is included, it is a null statement.

1-16. ENTRY POINTS

Both main programs and procedures can have multiple entry points. The first executable statement of a main program or procedure is an implicit entry point. Alternate entry points are labeled statements whose labels are declared in an entry declaration (see paragraph 3-7 for the format of an entry declaration). An entry point cannot be the object of a GO TO statement.

A program may be started at an alternate entry point with a parameter on the :RUN or :PREPRUN command. An alternate entry point for a procedure is equivalent to another name for the procedure that can be called with the same formal parameters. Local variables are set up and initialized regardless of which entry point is used. For example, assume the following program has been compiled and prepared (:SPLPREP) and the program file is $OLDPASS.

```
BEGIN
      ENTRY P1,P2,P3;
    .
    .
    .
P1: A:= 100;
    .
    .
    .
P2: A:= 200;
    .
    .
    .
P3: A:= 300;
    .
    .
    .
END.
```

To start execution at P2, use the command

```
:RUN $OLDPASS,P2
```

BASIC ELEMENTS

SECTION II

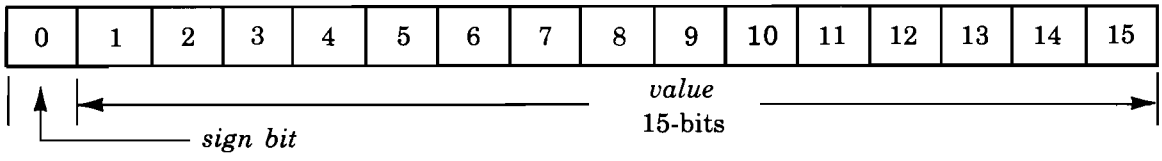
2-1. DATA STORAGE FORMATS

SPL processes six types of data: integer, double integer, real, long (extended precision real), byte, and logical. Each data type has its own representation in memory. The following paragraphs describe the data types and discuss the manner in which they are stored in memory.

2-2. INTEGER FORMAT

Integers are whole numbers containing no fractional part. Integer values are stored in one 16-bit computer word. The leftmost bit (bit 0) represents the arithmetic sign of the number (1= negative, 0= positive). The remaining 15 bits represent the binary value of the number. Integer numbers are represented in two's complement form and range from −32768 to +32767.

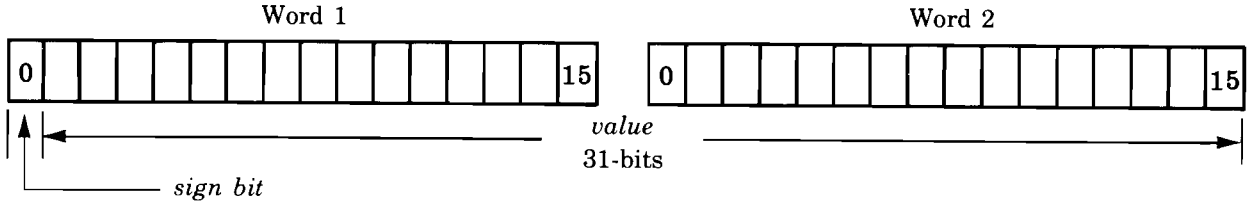
Decimal Value	Two's Complement
+32767	%077777
.	.
.	.
.	.
+ 1	%000001
0	%000000
− 1	%177777
− 2	%177776
.	.
.	.
.	.
−32768	%100000



2-3. DOUBLE INTEGER FORMAT

When you wish to use integer values with magnitudes greater than the integer format allows, you may use double integers. Double integers use 2 computer words for a total of 32 bits. The leftmost bit of the

first word (bit 0) is the sign bit (1= negative, 0= positive). The remaining 31 bits represent the binary value of the number. Double integer numbers are represented in two's complement form and range from −2,147,483,648 to +2,147,483,647.



2-4. REAL FORMAT

Real numbers are represented in memory by 32 bits (two consecutive 16-bit words) with three fields. The fields are the sign, the exponent, and the mantissa. The format is that known as excess 256 — exponents are biased by +256. Thus, a real number consists of:

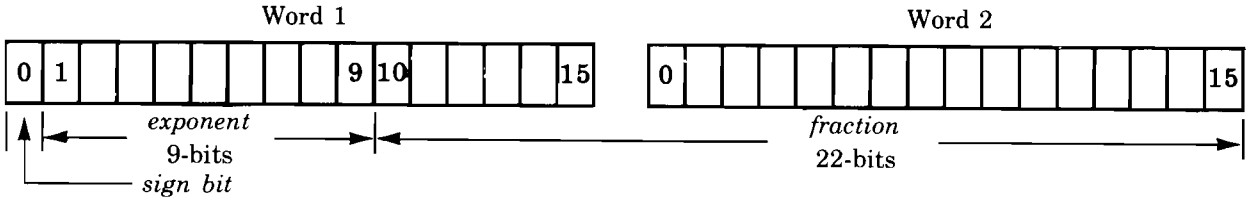
Sign(S)
Bit 0 of the first word (positive= 0, negative= 1). A value X and its negative, −X, differ only in the sign bit.

Exponent(E)
Bits 1 through 9 of the first word. The exponent ranges from 0 to 777 octal (511 decimal). This number represents a binary exponent, biased by 400 octal (256 decimal). The true exponent is E−256; it ranges from −256 to +255.

Fraction(F)
A binary number of the form 1.*xxx*, where *xxx* is represented by 22 bits, stored in bits 10 through 15 of the first word and all of the second word. Note that the 1. is not actually stored, there is an assumed 1. to the left of the binary point. Floating-point zero is the only exception — it is represented by all 32 bits being zero.

The range of the magnitude of non-zero real values is from $8.63617^{*}\ 10^{-78}$ to $1.157921\ *\ 10^{-77}$. Real numbers are accurate to 6.9 decimal places.

The internal representation for real numbers is:



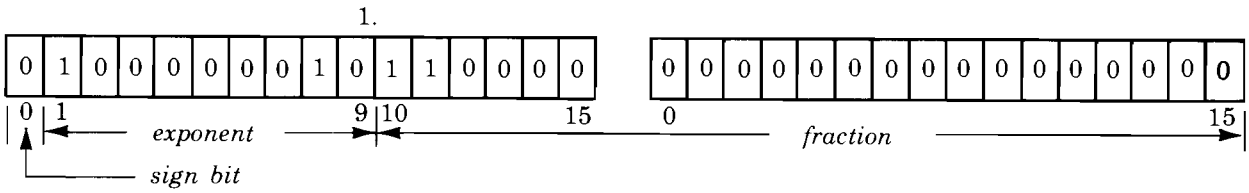
The formula for computing the decimal value of a floating-point representation is:

Decimal value = $(-1)^{S} * F * 2^{(E-256)}$

which is equivalent to:

$$\text{Decimal value} = (-1)^S * (1.0 + (xxx * 2^{-22})) * 2^{(E-256)}$$

For example, 7.0 is represented as



Sign (S) = 0 (positive)

Exponent (E) = 402 (octal) = 258 (decimal)

Fraction (F) = 1.11 (binary) = $(1 \times 2^0) + (1 \times 2^{-1}) + (1 \times 2^{-2})$
= 1 + 1/2 + 1/4
= 1.75 (decimal)

So, the decimal value of the real value is:

$(-1)^0 \times 1.75 \times 2^{(258 - 256)} = 1 \times 1.75 \times 2^2$
$= 1.75 \times 4$
$= 7.0$

2-5. LONG FORMAT *

Long numbers are represented in memory by 64 bits (four consecutive 16-bit words) with three fields. The fields are the sign, the exponent, and the mantissa. The format is that known as excess 256 — exponents are biased by +256. Thus, a long number consists of:

Sign(S)
Bit 0 of the first word (positive= 0, negative= 1). A value X and its negative, – X, differ only in the sign bit.

Exponent(E)
Bits 1 through 9 of the first word. The exponent ranges from 0 to 777 octal (511 decimal). This number represents a binary exponent, biased by 400 octal (256 decimal). The true exponent is E– 256; it ranges from – 256 to + 255.

Fraction(F)
A binary number of the form 1.*xxx*, where *xxx* is represented by 54 bits, stored in bits 10 through 15 of the first word and all of the second, third, and fourth words. Note that the 1. is not actually stored, there is an assumed 1. to the left of the binary point. Floating-point zero is the only exception — it is represented by all 64 bits being zero.

*NOTE: Throughout this discussion the following changes apply to Pre-Series II Systems: Long numbers are 48 bits (three words) accurate to 11.7 decimal places. The decimal value of a floating point representation of a long value is $(-1)^S * (1.0 + (xxx * 2^{-38})) * 2^{(E-256)}$

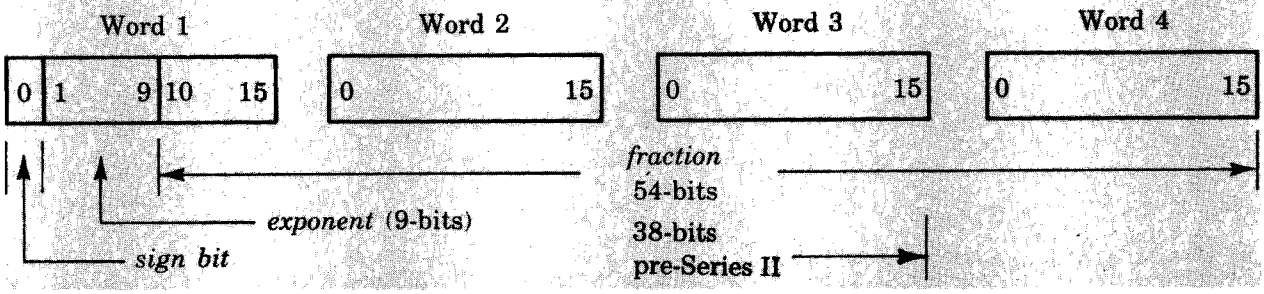
The range of the magnitude of non-zero long values is from $8.636168555094445 * 10^{-78}$ to $1.157920892373162 * 10^{77}$. Long numbers are accurate to 16.5 decimal places. The formula for computing the decimal value of a floating-point representation is:

$$\text{Decimal value} = (-1)^S * F * 2^{(E-256)}$$

which, for long values, is equivalent to:

$$\text{Decimal value} = (-1)^S * (1.0 + (xxx * 2^{-54})) * 2^{(E-256)}$$

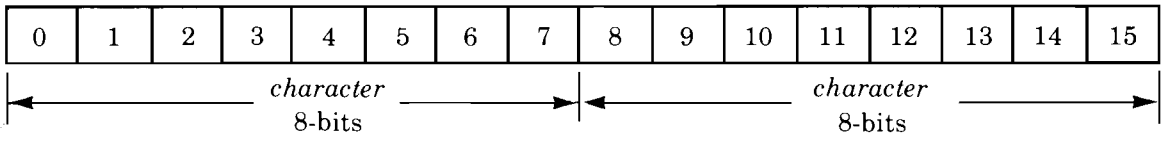
The internal representation for long numbers is:



2-6. BYTE FORMAT

Character strings are stored using byte format. Character values are represented by 8-bit ASCII codes, two characters packed in one 16-bit computer word. The number of words used to represent a character value depends on the actual number of characters in the string. Appendix A shows the ASCII characters and their octal codes.

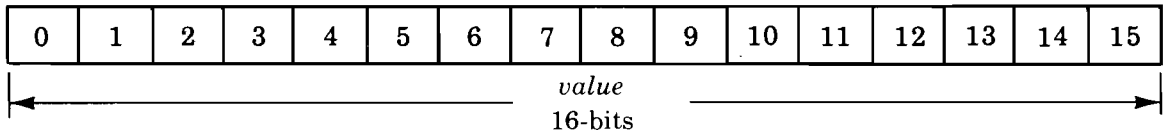
The internal representation of byte values is:



2-7. LOGICAL FORMAT

Logical values are stored in one 16-bit computer word. They are treated as unsigned integer values ranging from 0 to 65,535. A value is considered true if it is odd and false if it is even (i.e., only bit 15 is checked). When a value is set to TRUE, a word of all ones is used (% 177777). A value set to FALSE is all zeros.

The internal representation of a logical value is:



2-8. CONSTANT TYPES

Constants are literal values that stand for themselves. There are two basic types of constants in SPL: numeric constants and string constants.

Numeric constants are broken down into five types:

1. Integer (16 bits — includes 1 sign bit)
2. Double integer (32 bits — includes 1 sign bit)
3. Real (32 bit floating point)
4. Long (64 bit floating point)
5. Logical (16 bits — no sign bit)

String constants are made up of ASCII characters which are packed two 8-bit characters to a word.

In SPL, constants are merely bit patterns that occupy a given number of bits. A given 16-bit pattern can have many constant interpretations (two characters, an integer, a logical value, etc.). Note that hardware instructions provide arithmetic capability for all of the constant types mentioned here.

2-9. INTEGER CONSTANTS

Integers are signed whole numbers containing no fractional part. Decimal integer constants use the decimal digits 0 through 9. They can contain a leading plus (+) or minus (−) sign. A number without a leading sign is positive. The range of an integer constant is from −32768 to +32767.

The form of a decimal integer constant is,

[*sign*] integer

where

sign
is + or −.

integer
is a string of the digits 0 through 9.

For example,

0
12345
−31766
+12384

2-10. DOUBLE INTEGER CONSTANTS

Double integers are signed whole numbers containing no fractional part. Decimal double integer constants use the decimal digits 0 through 9 followed by a D. They can contain a leading plus (+) or

minus (−) sign. A number without a leading sign is positive. The range of a double integer constant is from −2,147,483,648 to +2,147,483,647. The form of a decimal double integer constant is:

[*sign*] integer D

where

sign
is + or −

integer
is a string of the digits 0 through 9.

For example,

```
−123456D
+99999999D
  312735D
 0 D
```

2-11. BASED CONSTANTS

SPL allows you to use any base from 2 (binary) through 16 (hexadecimal) in constants. A based constant can contain a leading sign and/or a trailing type designator. A leading per cent sign (%) denotes a based constant. The base is enclosed in parentheses following the per cent sign. If a base is not specified, the constant is octal (base 8). The letters A,B,C,D,E, and F represent the values 10,11,12,13,14, and 15 respectively in bases greater than 10. If a type designator is used with a base greater than 10, a space must precede the type designator.

The form of a based constant is:

[*sign*] %[(*base*)] *integer* [*type-designator*]

where

sign
is + or −.

base
is any integer between 2 and 16. If the % is used without a base being specified, base 8 (octal) is assumed.

integer
is a string of digits, where digit is between 0 and *base*− 1.

type-designator
is D,E, or L for DOUBLE, REAL, or LONG respectively. If a *type-designator* is not specified, the constant will be a single-word constant which can be used as type INTEGER, LOGICAL, or BYTE.

For REAL and LONG based constants, the bit pattern of the based integer is used directly as a right justified real number — it is not converted to floating point form. A leading minus sign will generate

the two's complement form of single-word and type DOUBLE based constants, but will only reverse the sign bit for REAL and LONG based constants.

For example,

```
+%777
-%(2)10101010
%(16)ABC D              <<type DOUBLE>>
%(16)ABCD               <<single-word>>
```

2-12. COMPOSITE CONSTANTS

Composite constants are a convenient way of representing specific bit patterns for tables and special numbers such as the lowest possible real number. A composite constant consists of a series of bit fields separated by commas which is enclosed in brackets ([]). Each bit field contains a field length and an unsigned integer value separated by a slash. The integer value may be an unsigned composite integer; thus, composite integers may be nested within a composite constant. Composite constants may contain a leading sign and/or a trailing type designator.

The form of a composite constant is:

[*sign*] *composite-integer* [*type-designator*]

where

sign
is + or −.

composite-integer
is of the form:

[*length/value,...,length/value*]

NOTE

The brackets [ ] in this case are literal symbols which are part of the syntax for composite integers — they do not represent the symbols used to denote optional items in this manual.

length
is an unsigned non-zero decimal, based, composite, or equated integer constant. The sum of the lengths for a composite constant cannot exceed the number of bits used to represent the constant type. If the sum of the lengths is greater than 16, a *type-designator* is required.

value
is any unsigned decimal, based, composite, or equated integer constant. *Type-designators* are not allowed.

type-designator
is D,E, or L for DOUBLE, REAL, or LONG respectively. If a *type-designator* is not specified, the constant will be a single-word constant which can be used as type INTEGER, LOGICAL, or BYTE.

Composite constants are formed by left-to-right concatenation of binary bit fields. Within each bit field, unspecified leading bits are set to zero and bits exceeding the field size are truncated on the left. The resulting composite integer is right justified with leading bits set to zero. If a minus sign is used with a single-word or a type DOUBLE composite constant, the two's complement will be generated. If a minus sign is used with a REAL or LONG composite constant, the sign bit will be reversed and the other bits will be unchanged — no conversion to floating point form occurs with composite constants.

For example,

```
 [32/1]D                                    = %00000000001
 [32/1]E                                    = %00000000001
-[32/1]D                                    = %37777777777
-[32/1]E                                    = %10000000001
 [3/2,12/%5252]                             = %25252
 [2/211,15/[3/%(2)101,12/0],10/123] D       = %720000173
-[3/2,12/%5252]                             = %152526
```

2-13. EQUATED INTEGERS

Equated integers are used to assign an integer value to an identifier for compile-time only. An equated integer 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 Equate declarations 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. An equated integer reference may be preceded by a plus (+) or minus (−) sign. The value assigned to an identifier in an EQUATE declaration must be a single-word value; however a D may be used after the identifier to convert the single-word value to a double-word value whose first word is all zeros. If a D is used, a space must separate the identifier from the D.

The form of an equated integer constant is

[*sign*] identifier [D]

where

sign
is + or −.

identifier
is a legal SPL identifier which has been declared in an EQUATE declaration (see paragraph 3-9).

2-14. REAL CONSTANTS

Real constants are represented by an integer part, a decimal point, and a decimal fraction. Either the integer part or the decimal fraction may be omitted (but not both) to indicate a zero value for that part only. A leading plus (+) or minus (−) sign may be used. A number without a sign is positive. The constant can contain a scale factor to indicate a power of ten by which the value is multiplied.

The forms of a real constant are

Format 1: [*sign*] *based/composite-integer* E

Format 2: [*sign*] *decimal-number* [E [*sign*] *power*]

Format 3: [*sign*] *decimal-integer* E [*sign*] *power*

where

sign
is either + or −.

based/composite-integer
is any unsigned based or composite integer constant.

decimal-number
is of one of the following three forms:

n.n
n.
.n

(*n* being an unsigned decimal integer).

power
is an unsigned decimal integer constant.

decimal-integer
is an unsigned decimal integer constant.

Real numbers are accurate to 6.9 decimal digits of magnitude (0 can be represented exactly). The absolute value of non-zero real numbers can range from 8.63617 x 10^{-78} to 1.157921 x 10^{77} . The E construct is used to indicate the scaling factor, if any. For example, 2.5E−2 means 2.5 x 10^{-2}.

Note that when a composite or based integer is used, there is no power after the E, and that the E is required to indicate a real value. The bit pattern created for the integer is used directly as a right-justified real number; it is not converted to floating-point form. This construct is useful for creating special floating-point constants such as the smallest positive number. When the base is greater than 10, a space must precede the E.

For example,

```
+1.234
−.2024
−1.105E−21
10E−20
%(4)321000E
%(2)1111011110111E
[3/5,5/273,20/%(16)102AB39]E
```

Some examples of invalid real constants are

+10.E	<<missing power>>
E-21	<<missing decimal-number>>
2E−	<<missing power>>

2-15. LONG CONSTANTS

Long constants are represented by an integer part, a decimal point, and a decimal fraction. Either the integer part or the decimal fraction may be omitted (but not both) to indicate a zero value for that part only. A leading plus (+) or minus (−) sign may be used. A number without a sign is positive. The constant can contain a scale factor to indicate a power of ten by which the value is multiplied.

The forms of a long constant are

Format 1: [*sign*] *based/composite-integer* L

Format 2: [*sign*] *decimal-number* [L [*sign*] power]

Format 3: [*sign*] *decimal-integer* L [*sign*] *power*

where

sign
is either + or −.

based/composite-integer
is any unsigned based or composite integer constant.

decimal-number
is of one of the following three forms:
n.n
n.
.n
(*n* being an unsigned decimal integer).

power
is an unsigned decimal integer constant.

decimal-integer
is an unsigned decimal integer constant.

Long numbers are accurate to 16.5*decimal digits of magnitude (0 can be represented exactly). The absolute value of non-zero long numbers can range from $8.636168555094445 \times 10^{-78}$ to $1.157920892373162 \times 10^{77}$. The L construct is used to indicate the scaling factor, if any. For example, 2.5L−2 means 2.5×10^{-2}.

Note that when a composite or based integer is used, there is no power after the L, and that the L is required to indicate a long value. The bit pattern created for the integer is used directly as a right-justified long number; it is not converted to floating-point form. This construct is useful for creating special floating-point constants such as the smallest positive number. When the base is greater than 10, a space must precede the L.

For example,

```
9321.678975L72
-.111015L-27
%(8)3777777777L
```

*11.7 with pre-Series II Systems

2-16. LOGICAL CONSTANTS

Logical constants are 16-bit positive integers. Hardware operations on logical values are defined for addition, subtraction, multiplication, division, and comparison.

Logical values can be represented by any of the following:

1. TRUE
2. FALSE
3. integer

where

TRUE and FALSE
are SPL Reserved words.

integer
is any (single word) decimal, based, composite, or equated integer.

A logical value is considered true if its value is odd, false if its value is even (i.e., only bit 15 is checked). When the reserved words TRUE and FALSE are used, they are equivalent to the integer values −1 (all ones) and 0 (all zeros) respectively. Since logical values are always assumed to be positive, they range from 0 to +65,535. When negative integers are used as logical values, they are interpreted as large positive numbers (e.g., −1 equals %177777).

2-17. STRING CONSTANTS

A string constant is a sequence of one or more ASCII characters bounded by quote marks ("). Each character is converted to its 8-bit representation and the characters are packed two per word.

The form of a string constant is

"*character-string*"

where

character-string
is a sequence of ASCII characters (see Appendix A).

A character string can contain from 1 to 127 ASCII characters. A quote (") is represented within a character string by a pair of quotes ("") to avoid ambiguity with the string terminator.

For example,

"THE CHARACTER "" IS A QUOTE MARK."
"A NORMAL STRING WOULD LOOK LIKE THIS"
"lowercase letters are not UPSHIFTED in strings"

2-18. IDENTIFIERS

Identifiers are symbols used to name data and code constructs in an SPL program. They consist of uppercase letters and numbers, and are assigned uses by declarations. There is no implicit typing for identifiers.

The form of an identifier is

letter [*letter'digit-string*]

where

letter
is a letter of the alphabet (A-Z).

letter'digit-string
is a string of letters (A-Z), digits (0-9), and apostrophes (').

An *identifier* always starts with a letter and may contain from 1 to 15 characters (letters, digits, and apostrophes). *Identifiers* larger than 15 characters are truncated on the right (A123456789012345 = A12345678901234). Lowercase letters are allowed, but are always converted to uppercase form (Aabc = AABC). If the listing device has upper and lowercase characters, a lowercase identifier is printed in lowercase, but SPL does not differentiate it from an uppercase identifier with the same characters. The attributes of an *identifier* are determined by a declaration, not by the form of the identifier.

Reserved words are combinations of characters that cannot be used as identifiers, since they have implied meanings in the language. (See Appendix B for a list of SPL reserved words).

For example,

```
MATRIX
A""B
AN'IDENTIFIER
MAT123
X
```

2-19. 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. Array elements are sometimes called subscripted variables.

SPL allows one-dimensional arrays (only one subscript is permitted) in all data types (integer, logical, real, byte, long, and double). Subscripting automatically uses the index register to indicate the element number. Bounds checking is not done at either compile-time or run-time. Arrays can be initialized but do not have a default initialization value. 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. Values in P-relative arrays are constants which cannot be changed at run-time.

2-20. 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. A pointer can be changed dynamically to point to different data items during program execution. Pointers are declared in a pointer declaration (see paragraph 3-4 for global pointer declarations and paragraph 7-24 for local pointer declarations).

There are four contexts in which pointers can be used:

1. Anywhere that the object of the pointer could be used; this generates an automatic indirect reference to the object of the pointer.

2. On the left side of an assignment statement to change the value of the object of the pointer.

3. 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.

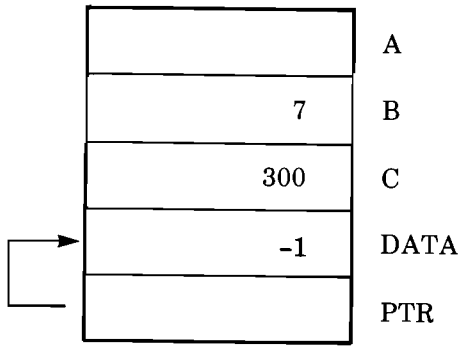
4. A pointer can implicitly reference the LST and SST instructions. (Privileged mode only.) The pointer reference must always be subscripted and cannot be preceded by '@'. MAP indicates this addressing scheme by ST+*number* as shown in the example below. Refer to the *Machine Instructions Set* manual for more detailed information.

```
00000100   00000 0   $CONTROL INNERLIST,MAP,ADR
00001000   00000 0   BEGIN
00002000   00000 1   INTEGER POINTER SYSGLOB=0;
00002100   00000 1   INTEGER CONSOLE;
                          DB+000
00003000   00000 1   CONSOLE:=SYSGLOB(%74);
                     00000    LDXI,074              021474  01.05
                     00001    LDI ,000              021000  01.05
                     00002    LST ,000              030000  02.45
                     00003    STOR  DB 000          051000  03.15
00004000   00004 1   END.
                     00004    PCAL,052              000000  14.90


IDENTIFIER          CLASS          TYPE          ADDRESS

  CONSOLE          SIMP. VAR.     INTEGER        DB+000
  SYSGLOB          POINTER        INTEGER        ST+000
  TERMINATE        PROCEDURE
```

For example, assume the following data declarations

```
INTEGER A,B:=7,C:=300,DATA:=-1;
INTEGER POINTER PTR:=@DATA;
```

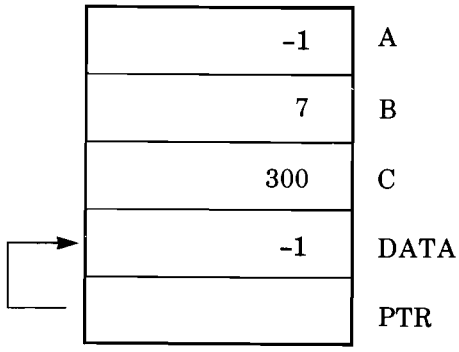
These declarations initialize the variables B, C, and DATA and set up PTR as a pointer to DATA as shown below.



Now, consider the statement

A:= PTR;

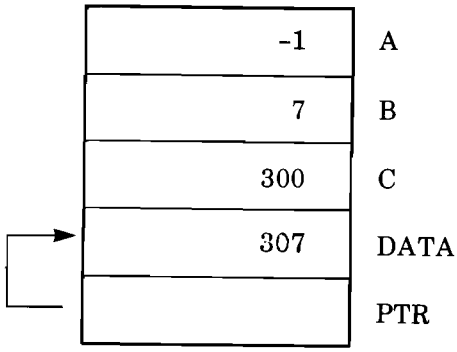
This statement assigns the object of the pointer PTR (i.e., DATA) to A.



Using the pointer on the left side of an assignment statement can change the value of the object of the pointer.

PTR:= B+ C;

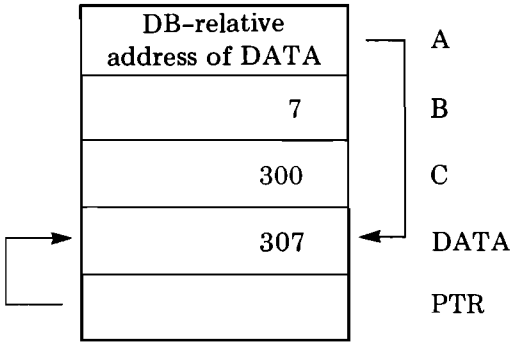
The object of the pointer PTR (i.e., DATA) is assigned the value of B+ C.



Preceding the pointer variable with an @ references the address contained in the pointer instead of the value of the object of the pointer. Using this construct on the right side of an assignment statement assigns the DB-relative address of the object of the pointer to a variable. For example,

```
A:=@PTR;
```

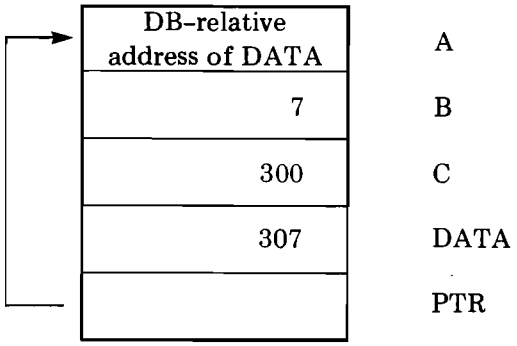
A is assigned the address contained in PTR (that is, the address of DATA).



To change the pointer to point to a different data item, use the @ construct on the left side of an assignment statement as shown below.

```
@PTR:=@A;
```

This statement changes PTR to point to A instead of DATA.



2-21. LABELS

Labels are used to identify statements for transfer of control and for documentation purposes. A label must always be followed by a colon (:) to separate it from the statement that it identifies. For consistency and documentation, labels may be declared with a label declaration; however, it is not necessary to do so since labels declare themselves automatically when they are used. A label can be used to identify only one statement within the scope of the identifier; that is, the same label can be used to identify two different statements as long as the statements are not both in the main body or both in the same procedure.

2-22. SWITCHES

The purpose of a switch is to transfer control to one of several labeled statements within a program. A switch is first declared with a switch declaration (see paragraph 3-6 for the format of a switch declaration). 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. A switch of program control is accomplished by using a GO TO statement with the switch identifier and an index. The index is evaluated to an integer value and control is transferred to the switch label specified by that number. Bounds checking on the index to insure that the value has a corresponding labeled statement is optional. See paragraph 5-2 for the form of the GO TO statement.

For example,

```
BEGIN
        INTEGER INDX;
        REAL A,B;
        SWITCH SW:= L1,L2,L3,L4;
            .
            .
            .
        INDX:= - 1;
LOOP:   INDX:= INDX+ 1;
        GO TO SW(INDX);
L1:     A:= B;
        GO TO LOOP;
L2:     B:= A;
        GO TO LOOP;
L3:     A:= A+ B;
        GO TO LOOP;
L4:     B:= A+ B;
            .
            .
            .
END.
```

GLOBAL DATA DECLARATIONS

SECTION III

3-1. TYPES OF DECLARATIONS

A declaration defines the attributes of an identifier before it is used in a program or procedure. All identifiers in SPL programs (with the exception of labels) must be explicitly declared once only within a single program or procedure. There are two possible levels of declarations in SPL:

Global (in a main program)
Local (in procedures)

Globally declared identifiers can be accessed throughout a program (even within procedures) and their declarations are grouped together at the beginning of the program. Locally declared identifiers can be accessed only within the procedure where declared and their declarations are grouped together at the beginning of the procedure body. This section covers global data declarations only; refer to section VII for local declarations.

Global data declarations immediately follow the opening BEGIN as shown below.

```
      BEGIN
----> [global-data-declarations]<----
      [procedures/intrinsics]
      [global-subroutines]
      [main-body]
      END.
```

Global data declarations are composed of the following types of declarations (which are described individually later in this section):

- global simple variable declarations
- global array declarations
- global pointer declarations
- label declarations
- switch declarations
- entry declarations
- define declarations
- equate declarations

Global data identifiers (simple variables, arrays, and pointers) are either allocated space in the stack or use space in the stack allocated to another identifier. Normally, the next available DB-relative location is allocated for the identifier. However, a register-relative or identifier-relative location may be specified in the declaration to override the default allocation. In this case, the referenced location is used without being allocated. When using identifier or register references, the compiler only checks that the resulting address is within the direct address range of the register being used. You must insure that this location does not exceed the bounds of your data stack when the identifier is referenced

at execution time. Additionally, when using a reference identifier, you must declare it before using it as a reference identifier. For example, the declarations:

```
INTEGER A,B,C;
LOGICAL D= A+ 2;
```

indicate that D is a LOGICAL simple variable using the same location as the INTEGER variable C. The syntax for register and identifier references is described in the appropriate paragraphs for the type of indentifier (simple variable, array, or pointer) in this section. Data identifiers which are register or identifier referenced cannot be initialized.

3-2. SIMPLE VARIABLE DECLARATIONS

A simple variable declaration specifies the type, addressing mode, storage allocation, and initialization value for identifiers to be used as single data items. The type assigned to a variable determines the amount of space allocated to the variable and the set of HP 3000 instructions which can operate on the variable.

Two methods can be used to link global variables to variables in separately compiled procedures. The first method is to use the GLOBAL attribute in the global variable declaration and the EXTERNAL attribute in the local variable declaration (see paragraph 7-19). The identifiers in both declarations must be the same and the MPE Segmenter is responsible for making the correct linkages. (See the *MPE Segmenter Subsystem Reference Manual* for a discussion of the Segmenter.) The second method is to include dummy global declarations at the beginning of subprogram compilations. All global declarations must be included, even for identifiers not referenced in the subprogram, and they must be in the same order as in the main program. It is possible, although not recommended, to use different identifiers for the same variable, but you are responsible for keeping them straight. The second method is faster and requires less space in the USL (User Subprogram Library) files, but does not protect you against improper linkages.

The form of a global simple variable declaration is:

[GLOBAL] ***type variable-declaration*[,...,*variable-declaration*];**

EXAMPLES:

```
INTEGER I,J:= 1245;
DOUBLE II:= -1234579 D;
REAL A,B,C:= 1.321E-21,Z= DB+ 3;
LOGICAL INDX= X,LI= I,JI= J;
GLOBAL BYTE DOLLAR:= "$";
```

where

type
specifies the data type of the variables in the declaration. The *type* may be INTEGER, LOGICAL, BYTE, DOUBLE, REAL, or LONG.

variable-declaration
can be any of the following forms:

variable [:= *initial-value*]
variable = *register* [*sign offset*]
variable = *reference-identifier* [*sign offset*]

variable
is a legal SPL *identifier*.

initial-value
is an SPL constant to be used as the value of the *variable* when program execution begins.

register
specifies the register to be used in a register reference. The *register* may be DB, Q, S, or X.

sign
is + or −.

offset
is an unsigned decimal, based, composite, or equated integer constant.

reference-identifier
is any legal SPL *identifier* which has been declared as a data item except DB,PB,Q,S, or X.

Form 1 of the variable declaration allocates the next available DB-relative location(s) for the *variable*. The amount of space allocated depends on the variable *type*. If an *initial value* is specified, the *variable* is initialized when execution starts. If the constant used for the initial-value is too large, it is truncated on the left, except string constants which are truncated on the right. If no *initial-value* is specified, the variable is not initialized.

Form 2 of the variable declaration equivalences a *variable* either to the index register (X) or to a location relative to the contents of one of the base registers (DB, Q, or S). Since the index register is 16 bits, only variables of type INTEGER, LOGICAL, and BYTE may be equivalenced to this register.

Form 3 of the variable declaration equivalences a *variable* to a location relative to another variable. The *reference-identifier* must be declared first. For example, the declarations

```
LOGICAL A;
INTEGER B= A+ 5;
```

equivalence B to the location 5 words past the location of A. Simple variables which are address referenced to arrays use either the location of the zero element of the array (if direct), or the location of the pointer to the zero element of the array (if indirect). Note that if the *reference-identifier* is an array, only the zero element may be used in a variable reference of a simple variable declaration. In any case, the final address must be within the direct address range.

DB, PB, Q, S, and X cannot be used as the *identifier* on the right side of an equals sign in a variable declaration, because they are interpreted as register references instead of variable references. For example, consider the declaration

```
INTEGER A,B,C,DB,D= DB+ 2;
```

The variable D is equivalenced to the location 2 cells past the cell to which the DB register points — not 2 cells past the location assigned to the variable DB.

The legal combinations of registers, signs, and offsets are shown below

Register	Sign	Offset
DB	+	0 to 255
Q	+	0 to 127
Q	–	0 to 63
S	–	0 to 63
X	none	none

3-3. ARRAY DECLARATION

An array declaration specifies one or more identifiers to represent arrays of subscripted variables. An array is a block of contiguous storage which is treated as an ordered sequence of "variables" having the same data type. Each "variable" or element of the array is denoted by a unique subscript (SPL provides one-dimensional arrays only). An array declaration defines the following attributes of an array:

- The bounds specification (if any) which determines the size of the array and the legitimate range of indexing.
- The data type of the array elements.
- The storage allocation method.
- The initial values, if desired.
- The access mode (direct or indirect).

There are two types of access modes used for arrays: indirect and direct. An indirect array uses a pointer to the zero element of the array. Addressing an indirect array element uses both indirect addressing and indexing. If the array is a BYTE array, the pointer contains a DB-relative byte address. For all other data types, the pointer contains a DB-relative word address. A direct array uses a location within the direct address range of one of the registers (DB, Q, or S) as the zero element of the array and then uses indexing to address a specific array element. Figure 3-1 illustrates the differences between direct and indirect arrays.

The area in the stack between DB and the initial value of Q is divided into two areas: Primary DB Storage and Secondary DB Storage. The Primary DB area is used for global storage of simple variables, direct arrays, and pointers to indirect global arrays. The Secondary DB area is used for global storage of indirect arrays. The Primary DB area cannot normally extend past DB+ 255. The only exception is when the last global data declaration is for a DB-relative direct array whose zero

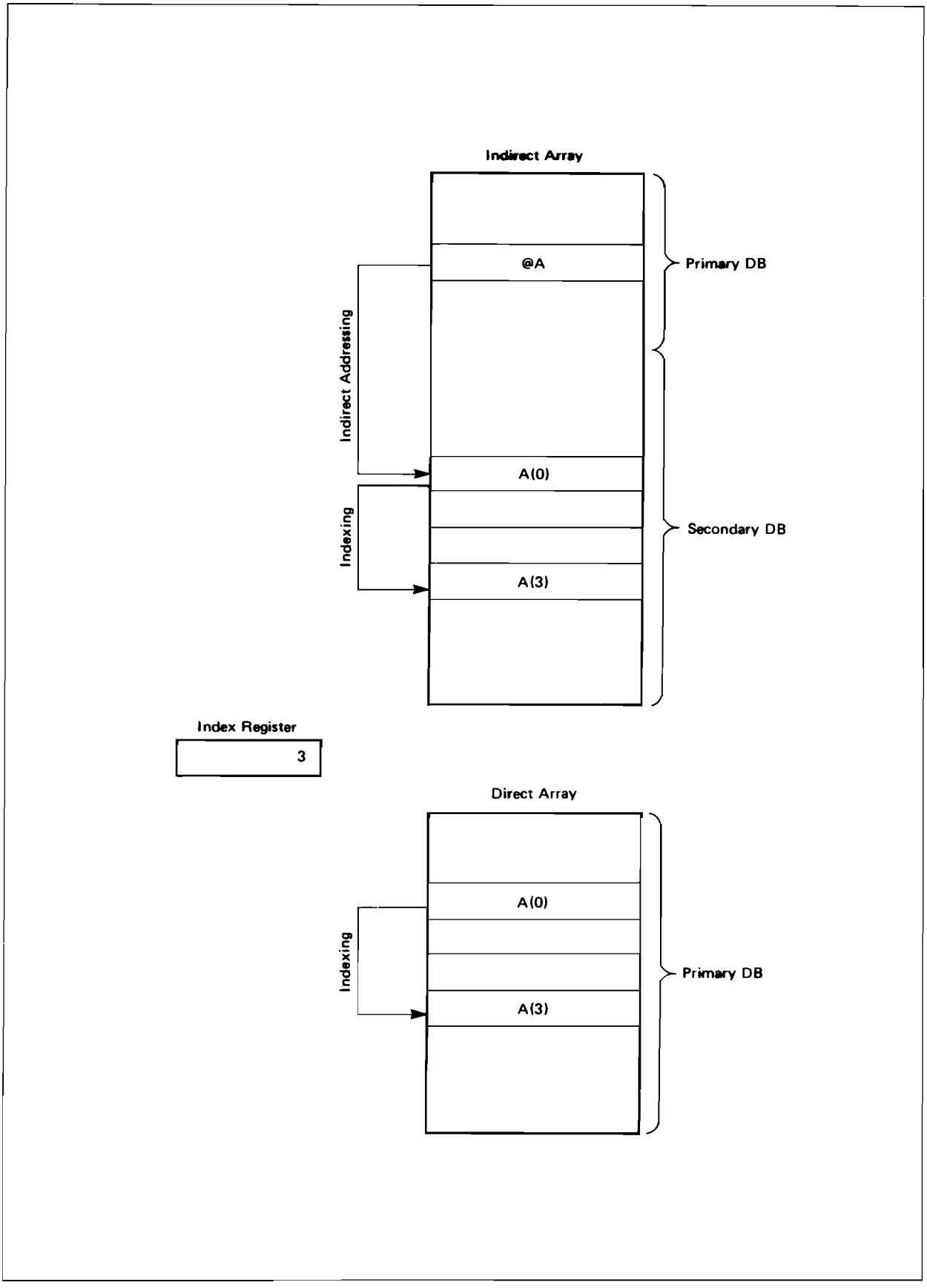


Figure 3-1. Accessing Array Elements

element falls between DB+ 0 and DB+ 255. Since the index register is used to address array elements, the array may extend past DB+ 255. The Secondary DB area immediately follows the Primary DB area regardless of the size of the Primary DB area.

There are two methods which can be used to link global arrays to arrays in separately compiled procedures. The first method is to use the GLOBAL attribute in the global array declaration and the EXTERNAL attribute in the local array declaration (see paragraph 7-23). The identifiers in both declarations must be the same and the Segmenter is responsible for making the correct linkages. The second method is to include dummy global declarations at the beginning of subprogram compilations. All global declarations must be included, even for identifiers not referenced in the subprogram, and they must be in the same order as in the main program. It is possible, although not recommended, to use different identifiers for the same array, but you are responsible for keeping them straight. The second method is faster and requires less space in the USL (User Subprogram Library) files, but does not protect you against improper linkages.

The form of a global array declaration is:

[GLOBAL] [*type*] ARRAY [*global-array-dec,...,global-array-dec,*]

{ *global-array-dec* / *initialized-global-array-dec* } ;

where

GLOBAL
is used for arrays which are referenced in procedures compiled separately.

type
specifies the data type of the array. The *type* can be INTEGER, LOGICAL, BYTE, DOUBLE, REAL, or LONG. If not specified, the array is type LOGICAL.

global-array-dec
is one of the following forms:

1. *array-name(lower:upper)* [=DB]

   This form is used for an uninitialized array with defined bounds. If =DB is not specified, the array is indirect and the next available DB Primary location is allocated for the pointer to the zero element of the array. Storage for the array itself is allocated in the Secondary DB area. If =DB is specified, the array is direct and the next available *n* cells in the DB Primary area are allocated for the array (where *n* is the number of locations required to store the array). The zero element of the array must be within the direct address range whether or not it is actually an element of the array. For example, consider the declaration:

   INTEGER ARRAY A(−20:−10)=DB;

   The next available DB primary location is allocated to A(−20), but all indexing is done relative to A(0) even though it is not an actual element of the array. The address which A(0) would have if it were in the array must be between DB+ 0 and DB+ 255.

2. *array-name(@)*= DB [+ *offset*]

This form is used for an indirect array with undefined bounds. If no *offset* is specified, the next available Primary DB location is used, without being allocated, as the pointer to the zero element of the array. If an *offset* is specified, then that DB-relative cell is used, without being allocated, as the pointer to the zero element. In either case, space is not allocated for the array in the Secondary DB area nor is initialization allowed.

3. *array-name(*)*= DB [+ *offset*]

This form is used for a direct array with undefined bounds. If no *offset* is specified, the next available Primary DB location is used, without being allocated, as the zero element of the array. If an offset is specified, then that DB-relative location is used, without being allocated, as the zero element of the array. In either case, space is not allocated for the array nor is initialization allowed.

*4. *array-name(@)* [=*register sign offset*]

This form is used for an indirect array with undefined bounds whose pointer is Q or S-relative. If a base-register reference is not specified, the next available DB cell is allocated for the pointer to the zero element of the array. If a base-register reference is specified, then that Q-relative or S-relative cell is used, without being allocated, as the pointer to the zero element of the array. Space is not allocated for the array nor is initialization allowed.

*5. *array-name(*)*

This form can be used for an indirect array with undefined bounds. The next available DB cell is allocated for the pointer to the zero element of the array. Space is not allocated for the array nor is initialization allowed. This form is equivalent to *array-name(@)* without a base-register reference.

*6. *array-name(*)* = *register sign offset*

This form is used for direct arrays with undefined bounds which are Q-relative or S-relative. The specified cell is used as the zero element of the array; however, space for the array is not actually allocated and the array cannot be initialized.

*7. *array-name(*)* = *reference-identifier* [*sign offset*]

This form is used for an indirect array with undefined bounds whose pointer is Q- or S-relative. If a base-register reference is not specified, the next available DB cell is allocated for the pointer to the zero element of the array. If a base-register reference is specified, then that Q-relative or S-relative cell is used, without being allocated, as the pointer to the zero element of the array. Space is not allocated for the array nor is initialization allowed.

```
INTEGER B(*)= A+ 10;
```

would not be allowed because the direct address range for the DB register is 0 to 255. If the array is direct, the referenced location is used as the zero element of the array. If the array is indirect, the referenced location is used as the pointer to the zero element except when either the array or the *reference-identifier* (but not both) is type BYTE, in which case the next available DB-cell is allocated for the pointer to the zero element. Space is not allocated for the

array nor can the array be initialized. DB, PB, Q, S, and X cannot be used as the *reference-identifer* because they are interpreted as register references instead.

*8. *array-name(*) = reference-identifier (index)*

This form is used for equivalencing one array to another array. The *reference-identifier* may be either an array or a pointer variable and must be declared first. If the *reference-identifier* is a direct array, the array is a direct array whose zero element is the location of the referenced array element. If the *reference-identifier* is an indirect array or a pointer variable, the array is indirect. In this case, the next available DB cell is allocated for the pointer to the zero element of the array if a non-zero index is specified or if either the array or the *reference-identifier* (but not both) is type BYTE; otherwise, both use the same location for the pointer to the zero element. In any case, space is not allocated for the equivalenced array nor can the equivalenced array be initialized. DB, PB, Q, S, and X cannot be used as the *reference-identifier* because they are interpreted as register references instead.

*Forms 4 through 8 are not allowed if the word GLOBAL is included in the declaration.

array-name
is a legal SPL *identifier.*

reference-identifier
is any legal SPL *identifier* except DB,PB,Q,S, or X which has been declared as a data item.

register
specifies the base register in a register reference. The *register* may be either Q or S.

sign
is + or −.

offset
is an unsigned decimal, based, composite, or equated integer constant within the direct address range as shown below:

Register	Sign	Offset
DB	+	0 to 255
Q	+	0 to 127
Q	−	0 to 63
S	−	0 to 63

initialized-global-array
is of the form:

array-name(lower:upper) [=DB] := *value-group*[,...,value-group]

lower
specifies the lower bound of the array. It can be any decimal, based, composite, or equated single-word integer constant or constant expression.

upper
specifies the upper bound of the array. It can be any decimal, based, composite, or equated single-word integer constant or constant expression.

index
indicates the element of the referenced array to be used as the reference location. The *index* can be any decimal, based, composite, or equated single-word integer constant.

value-group
is either of the following:

initial-value
repetition-factor (*initial-value* [,...,*initial-value*])

initial-value
is any SPL numeric or string constant.

repetition-factor
specifies the number of times the initial value list will be used to initialize the array elements. The *repetition-factor* can be any unsigned non-zero decimal, based, composite, or equated single-word integer constant.

Global arrays with defined bounds can be initialized. Initialization consists of a := followed by a list of numerical constants or strings. A group of constants can be surrounded by parentheses and preceded by a repetition factor (*n*) to specify that the constants in parentheses are to be used *n* times in initializing the array before going on to the next item in the list. These repeat groups cannot be nested. Elements are initialized starting with the lowest subscript and continuing up until the constant list is exhausted. The initialization list cannot contain more values than there are elements in the array. If the constant used for the initial value is too large, it is truncated on the left except string constants which are truncated on the right. If no initial value is specified, the variable is not initialized. Only the last array in a declaration list can be initialized.

Table 3-1 summarizes the syntax and meanings for the various forms of global array declarations. Figure 3-2 shows a series of array declarations with the locations assigned to the identifiers.

Table 3-1. Global Array Declaration Summary

FORM	OFFSET RANGE	ADDRESSING MODE	POINTER LOCATION	ZERO ELEMENT LOCATION
id(*low*:*up*)		Indirect	next DB (A)	Sec. DB (A)
id(*low*:*up*)=DB		Direct		Primary DB (A)
id(@)=DB		Indirect	next DB	C(next DB)
id(@)=DB+*offset*	0-255	Indirect	DB+*offset*	C(DB+*offset*)
id(*)=DB		Direct		Primary DB
id(*)=DB+*offset*	0-255	Direct		DB+*offset*
id(@)		Indirect	next DB (A)	C(next DB)
id(@)=Q+*offset*	0-127	Indirect	Q+*offset*	C(Q+*offset*)
id(@)=Q−*offset*	0-63	Indirect	Q-*offset*	C(Q-*offset*)
id(@)=S−*offset*	0-63	Indirect	S-*offset*	C(S-*offset*)
id(*)		Indirect	next DB (A)	C(next DB)
id(*)=*id*		Note 1	Note 2	Note 3
id(*)=*id*+*offset*	Note 4	Direct		*id*+*offset*
id(*)=*id*−*offset*	Note 4	Direct		*id*−*offset*
id(*)=*id*(*index*)		Note 5	Note 6	*id*(*index*)
id(*)=Q+*offset*	0-127	Direct		Q+*offset*
id(*)=Q−*offset*	0-63	Direct		Q-*offset*
id(*)=S−*offset*	0-63	Direct		S-*offset*

Legend

(A) — Storage is allocated for the designated pointer or array.

C( ) — The contents of the location in parentheses is the address of the zero element of the array.

id – *identifier*

low — lower bound

up — upper bound

NOTES

1. If the right side *id* is a direct array or a simple variable, the addressing mode is direct. If the right side *id* is an indirect array or a pointer variable, the addressing mode is indirect.

2. If the addressing mode is indirect, both identifiers use the same pointer location unless one *id* is type BYTE and the other is not, in which case, the next available DB-cell is allocated for the pointer.

3. The zero element is in the same location as the right side *id* (or its zero element if the right side *id* is an array).

4. The offset must result in an effective address within the direct address range of the base register which the right side *id* uses.

5. If the right side *id* is a direct array, the left side *id* is direct; if the right side *id* is a pointer variable or an indirect array, the left side *id* will be indirect.

6. If the addressing mode is indirect, the next available DB-cell is allocated for the pointer if:

   a. a non-zero index is specified.
      and/or
   b. one of the two identifiers is type BYTE and the other is not.

Otherwise, both identifiers use the same pointer location. If the addressing mode is direct, there is no pointer.

3-4. POINTER DECLARATION

A pointer declaration defines an identifier as a "pointer" — a single word quantity used to contain the DB-relative address of another data item — the object of the pointer. A pointer declaration defines the following attributes of a pointer:

- The data type.
- The storage allocation method.
- The initial address to be stored in the pointer (optional).

When the pointer is accessed, the object is accessed indirectly through the pointer address. The object is assumed to be, or is treated as if it were, the type of the pointer.

There are two methods which can be used to link global pointers to pointers in separately compiled procedures. The first method is to use the GLOBAL attribute in the global pointer declaration and the EXTERNAL attribute in the local pointer declaration (see paragraph 7-27). The identifiers in both declarations must be the same and the Segmenter is responsible for making the correct linkages. The second method is to include dummy global declarations at the beginning of subprogram compilations.

```
00001000  00000 0   $CONTROL ADR
00002000  00000 0   BEGIN
00004000  00000 1   ARRAY A(0:10),A0(0:10):=11(%17);
                            DB+000
                            DB+001
00005000  00001 1   REAL ARRAY A1(0:10);
                            DB+002
00006000  00001 1   REAL ARRAY A2(0:10)=DB;
                            DB+003
00007000  00001 1   REAL ARRAY A3(@)=DB;
                            DB+031
00008000  00001 1   REAL ARRAY A4(@)=DB+5;
                            DB+005
00009000  00001 1   REAL ARRAY A5(*)=DB;
                            DB+031
00010000  00001 1   REAL ARRAY A6(*)=DB+6;
                            DB+006
00011000  00001 1   REAL ARRAY A7(@);
                            DB+031
00012000  00001 1   REAL ARRAY A8(@)=Q+3;
                            Q +003
00013000  00001 1   REAL ARRAY A9(@)=Q-3;
                            Q -003
00014000  00001 1   REAL ARRAY A10(@)=S-2;
                            S -002
00015000  00001 1   REAL ARRAY A11(*);
                            DB+032
00016000  00001 1   REAL ARRAY A12(*)=A1;
                            DB+002
00017000  00001 1   REAL ARRAY A13(*)=A1+4;
                            DB+006
00018000  00001 1   REAL ARRAY A14(*)=A2-1;
                            DB+002
00019000  00001 1   REAL ARRAY A15(*)=A1(5);
                            DB+033
00020000  00001 1   REAL ARRAY A16(*)=Q+3;
                            Q +003
00021000  00001 1   REAL ARRAY A17(*)=Q-3;
                            Q -003
00022000  00001 1   REAL ARRAY A18(*)=S-2;
                            S -002
00023000  00001 1   BYTE ARRAY A19(*)=A0;
                            DB+034
00061000  00001 1   END.
 PRIMARY DB STORAGE=%035;    SECONDARY DB STORAGE=%00054
 NO. ERRORS=000;             NO. WARNINGS=000
 PROCESSOR TIME=0:00:02;     ELAPSED TIME=0:00:08
```

Figure 3-2. Sample Global Array Declarations

All global declarations must be included, even for identifiers not referenced in the subprogram, and they must be in the same order as in the main program. It is possible, although not recommended, to use different identifiers for the same pointer, but you are responsible for keeping them straight. The second method is faster and requires less space in the USL (User Subprogram Library) files, but does not protect you against improper linkages.

The form of a global pointer declaration is:

[GLOBAL] [*type*] POINTER *pointer-dec* [,...,*pointer-dec*];

EXAMPLES:

```
INTEGER A; LOGICAL B;
BYTE POINTER P:=@A;
INTEGER ARRAY N(0:10);
INTEGER POINTER PN:=@N(5);
POINTER P3=DB+ 2,P4,P5:=@A, P6=B;
INTEGER POINTER PCB = 3;
```

where

GLOBAL
is used for pointers referenced in procedures compiled separately.

pointer-dec
is one of the following:

1. *pointer-name* [:= @*reference-identifier* [(*index*)]]

   This form allocates the next available DB cell for the pointer variable. If the :=@*reference-identifier* is used, the pointer is initialized to the address of the *reference-identifier* or array-element if an *index* is included. The *reference-identifer* must be declared first.

   NOTE

   Global pointers can only be initialized to point to identifiers which have been declared to be DB-relative, either explicitly or implicitly. They cannot be initialized to point to identifiers which have been register referenced to the Q, S, or X registers. Thus, the following is not allowed:

   INTEGER A=Q+1; POINTER B:=@A;

   However, you can use an assignment statement (see paragraph 4-20) to dynamically set the pointer to such a variable unless it was equivalenced to the index register.

2. *pointer-name* = *reference-identifier* [*sign offset*]

   This form is used to equivalence a pointer variable to a location relative to another identifier.

Space is not allocated for the pointer nor can the pointer be initialized. The resulting address for the pointer variable must be within the direct address range of the base register which the *reference-identifier* uses.

3. *pointer-name = register [sign offset]*

   This form is used to equivalence a pointer variable to a location relative to a base-register. Space is not allocated for the pointer nor can the pointer be intitialized. The resulting address for the pointer variable must be within the direct address range of the specified base-register.

4. *pointer-name = offset*

   This form is used only in privileged mode. It is the offset in System DB. The pointer reference must always be subscripted and cannot be preceded by '@'.

type
specifies the data type of the pointer variables in the declaration. The *type* can be INTEGER, LOGICAL, BYTE, DOUBLE, REAL, or LONG.

pointer-name
is a legal SPL *identifier*.

reference-identifier
is any legal SPL *identifier* which has been declared as a data item except DB,PB,Q,S, or X.

register
specifies the base register in a register reference. The *register* can be DB, Q, or S.

sign
is + or −.

offset
is an unsigned decimal, based, composite, or equated integer within the direct address range as shown below.

Register	Sign	Offset
DB	+	0 to 255
Q	+	0 to 127
Q	−	0 to 63
S	−	0 to 63
ST (system table)	+	> = 0

index
indicates the array element whose address the pointer will be initialized to contain. The *index* can be any decimal, based, composite, or equated single-word integer constant.

Pointers are initialized with addresses of other variables or constants. The method is to follow the pointer with :=@ and a data reference (simple variable, pointer element, or array element) or := constant. The address of the specified data item, adjusted to the address type of the pointer, is stored in the cell allocated for the pointer. BYTE pointers contain DB-relative byte addresses, whereas all other types of pointers contain DB-relative word addresses.

See "Pointers" (paragraph 2-20) for methods of referring to and through pointers. Pointers can be indexed like arrays and can contain word or byte addresses.

Pointers can be declared with all data types; if no type is specified, type LOGICAL is assumed. The type determines what data type the object of the pointer is assumed to have. This allows objects declared with one type to be accessed as another data type by accessing them through pointers.

Pointers which are not address referenced are allocated the next available DB-relative location and can be initialized. Pointers which are referenced use the address of the referenced item or the specified register relative location and cannot be initialized.

3-5. LABEL DECLARATION

A label declaration specifies that an identifier will be used in the program as a label to identify a statement. Labels are referenced when it is necessary to transfer control to a specific statement; they need not be declared explicitly unless the programmer wishes.

The form of a label declaration is:

LABEL ***label*** **[,...,*label*];**

EXAMPLES:

```
LABEL L1,L2,L3;
LABEL L;
```

where

label
is a legal SPL *identifier*.

Labels are used to identify statements as follows:

```
LABEL L1;
  .
  .
  .
L1:A:= B;
```

The syntax for labeled statements is given in paragraph 1-3. In SPL, a *label* implicitly declares itself when it is used to identify a statement, as the object of a GO TO statement, or in a switch declaration. It need not be explicitly declared in a label declaration except as desired for documentation purposes. See "GO TO Statement" (paragraph 5-2) and "Switch Declaration" below for use of labels.

3-6. SWITCH DECLARATION

A switch declaration relates an identifier to an ordered set of labels. The switch is accessed as a computed (or indexed) GO TO statement. The purpose of a switch is to allow selective transfer of control to any of the statements identified by the labels in the switch declaration.

The form of a switch declaration is:

```
SWITCH switch-name := label [,...,label];
```

EXAMPLES:

```
SWITCH SW:=L1,L2,L3,L4,L5,L6,L7,L8,L9;
SWITCH ERROR'SELECT:=ERR1,ERR2,ERR3,ERR4,ERR5,ERR6;
```

where

switch-name
is a legal SPL *identifier*.

label
identifies the statement to which control is transfered when the switch is invoked.

Only one *switch-name* can be declared in each switch declaration. Associated with each *label* in the label list from left-to-right is an ordinal integer from 0 to $n-1$, where n is the number of labels in the list. This integer indicates the position of the *label* in the list. Each position in the list must contain a *label;* null elements are not allowed. When the *switch-name* is referenced (see "GO TO Statement" in paragraph 5-2), the value of an integer subscript determines which label is selected from the list. Bounds checking in this selection is optional. Entry points are not allowed in switch declarations Switch labels may not occur in subroutines.

3-7. ENTRY DECLARATION

The purpose of a global entry declaration is to specify multiple entry points to a main program beyond the implicit entry point which is the first statement of the program. Each entry identifier must occur somewhere in the body as a statement label, but cannot be the object of a GO TO.

The form of an entry declaration is:

```
ENTRY label [,...,label];
```

EXAMPLES:

```
ENTRY P1,P2,P3;
ENTRY P1;
```

where

label
identifies the statement to be used as an alternate entry point.

By specifying the entry point to the operating system, the program can be started at other than its natural beginning. See "Entry Points" in paragraph 1-16.

For example, here is a sample entry declaration:

```
ENTRY P1,P2,P3;
```

3-8. DEFINE DECLARATION AND REFERENCE

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. This provides a convenient abbreviation mechanism to avoid repeating long constructs that are used many times throughout a program.

The form of a define declaration is:

```
DEFINE identifier = text# [,...,identifier = text#] ;
```

EXAMPLES:

```
DEFINE AS= ASSEMBLE(#,LA= LONG ARRAY#;
DEFINE DA= DOUBLE ARRAY#;
```

where

identifier
is a legal SPL *identifier*.

text
specifies the block of text to be substituted when the define is invoked. The *text* can be any sequence of ASCII characters; however, # can be used only within a string.

A define identifier can be referenced anywhere except within an identifier, string, or constant. The text should make sense when inserted where the define is referenced.

At declaration time, a define has no effect on the compilation of the program. It has effect only in the context where it is referenced. For this reason, undeclared identifiers can appear in defines; they need to have been declared only when the define is referenced. Similarly, the define text is checked for syntax errors in the context where it is referenced, not where it is declared.

Define declarations can be nested (define identifiers can be used in other definitions), but they cannot be recursive (a define identifier appearing within its own text), since this leads to infinite nesting when the define is referenced.

The number sign (#) terminates a define text only if it is not contained in a string. For example, the string "ABCD#"# is valid text terminated by the second #. Incomplete comments cannot appear in DEFINEs.

Only one block of text can be assigned to a particular identifier.

For example, here are some sample define declarations and references:

```
DEFINE I= ARRAY B(0:1)#;
INTEGER I; <<INTEGER ARRAY B(0:1);>>
```

```
DEFINE SUM= A+ B+ C+ D+ E#;
J:= SUM; <<J:= A+ B+ C+ D+ E;>>
```

3-9. EQUATE DECLARATION AND REFERENCE

An equate declaration assigns an integer value (determined by an expression of integer constants and other equates) to an identifier. The equate mechanism is only a documentation and maintenance convenience; it does not allocate any run-time storage, but merely provides a form of consistent identification for constants. When an equate identifier is used, the appropriate constant is substituted in its place. When equates are used instead of actual constants, programs can be updated easily; instead of replacing every occurrence of a constant, only the equate declaration is changed.

The form of an equate declaration is:

EQUATE ***identifier*** **=** ***equate-expression*** **[,..., *identifier* = *equate-expression*];**

EXAMPLES:

```
EQUATE BELL= 7,CR=%15;
EQUATE N= 100,M= N+ 50;
```

where

identifier
is a legal SPL *identifier*.

equate-expression
can be either one of or a combination of two forms:

[*sign*] *unsigned-integer* [*operator unsigned-equate-expr*]

(equate-expression)

sign
is + or −.

unsigned-integer
is an unsigned decimal, based, composite, or equated single-word integer constant.

operator
is +,−,*, or /.

unsigned-equate-expr
is an unsigned *equate-expression*.

The value to be assigned to an equate *identifier* is determined by an equate expression. Equate expressions consist of operators (*,/,+,−), unsigned integers (including previously defined equated integers), and parentheses. Evaluation of the expression proceeds from left to right, except that multiplication and division (*,/) are done before addition and subtraction (+,−) and expressions in

parentheses are done before the operators that surround them. The value of an equate expression must fit in a single-word or it will be truncated on the left. Since equate identifiers can be used in equate expressions, a series of related equate declarations can be set up such that changing only the first changes all the rest.

Equate identifiers can be used anywhere in the program that an integer or unsigned integer constant is allowed.

For example, here are some sample equate declarations and references:

```
EQUATE M=1,N=M+1,P=N+1;
EQUATE T=20*P/(20-P+M);
J:=136*T;
  <<M=1, N=2, P=3, T=3, J=408>>
```

3-10. DATASEG DECLARATION

The DATASEG declaration is intended for privileged users requiring an extra data segment (defined as split-stack mode, section 8-1). It ensures the reliability of the generated split-stack code by limiting the declared variables to explicit DB-relative offsets. Only simple variables, arrays, and pointers are permitted as DATASEG declarations; no GLOBAL, EXTERNAL, or OWN declarations are allowed. A variable declaration without an offset will be assigned the next available offset.

The variables defined within the DATASEG declaration are used in conjunction with the MOVEX instruction and the WITH statement, as detailed in section 4-21A and 6-5 respectively.

The form of a DATASEG declaration is:

DATASEG *dataseg-name = dataseg#*

```
    BEGIN
    type dataseg-variable [= dataseg offset]

    END;
```

EXAMPLES:

```
    DATASEG X=77
      BEGIN
      INTEGER I;          <<OFFSET = 0>>
      REAL R=X+5;         <<OFFSET = 5>>
      LONG L=R+2;         <<OFFSET = 7>>
      ARRAY A(0:5);       <<OFFSET = 1>>
      END;
```

where

dataseg-name
is an SPL identifier,

dataseg#
is an integer constant or integer constant expression,

type
may be INTEGER, LOGICAL, BYTE, DOUBLE, REAL, or LONG,

dataseg-variable
is a legal SPL identifier,

dataseg-offset
is the dataseg-variable followed by a sign (+,−)
and an integer constant.

EXPRESSIONS, ASSIGNMENT, AND SCAN STATEMENTS

SECTION IV

4-1. EXPRESSION TYPES

An expression is a sequence of operations upon constants, variables, and indexed items which results in a single value of a specified data type. If the data type is logical, the expression is a logical expression and logical operators are allowed within it. If the data type is numeric (i.e., byte, integer, double, real, or long), the expression is an arithmetic expression and arithmetic operators are used within it. An IF expression allows a choice to be made between two expressions of the same word size based on hardware or software conditions.

Within SPL expressions, only variables of the same data type can appear on either side of an operator. That is, an integer can be multiplied by an integer, but not by a real. The only exception to this rule is the exponentiate operator (^) in arithmetic expressions; real and long data items can be exponentiated to integer powers. In all other cases, the combination of differing data types can only be accomplished through type transfer functions. For example, the function FIXR converts an expression of type real into one of type double and rounds the result to the closest integer:

FIXR(*real-expression*)

A corresponding function, FIXT, converts real to double and truncates the result:

FIXT(*real-expression*)

Type transfer functions are not available for all possible transformations. The following table shows which transfers are provided and which functions should be used in each case. In some cases, it may be necessary to specify nested type transfer functions (e.g., to convert from real to integer, either INTEGER(FIXR(*real-expression*)) or INTEGER(FIXT(*real-expression*))).

FROM	TO					
	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	------

4-2. VARIABLES

A variable is one of the items which can occur in expressions. Each variable, whether it is a simple variable, an array element, a pointer reference, or the top of the stack, is associated with one data item of a specific type. The address of any data item can be used as an integer variable since it is a 16-bit signed quantity.

The form of a variable in an expression is one of the following:

data-item **[(*index*)]**
TOS
@*identifier* **[(*index*)]**
ABSOLUTE(*index*)

The form of a variable on the left of an assignment operator (:=) is one of the following:

data-item **[(*index*)]**
TOS
@*pointer-name*
ABSOLUTE(*index*)

where

data-item
is a *simple-variable*, *array-name*, or *pointer-name*.

index
specifies an offset. The *index* is either an expression or an assignment statement of type integer, logical, or byte. If an *index* is not specified with an *array-name*, a *pointer-name*, or ABSOLUTE, then zero is assumed.

TOS
is the Top Of Stack

identifier
is a *simple-variable, array-name, pointer-name, label,* or *procedure-name* whose DB- or PB-relative address is used as an integer value.

ABSOLUTE
is used to denote an absolute memory location. To use this construct, you must have privileged mode (PM) capability.

The three most common types of variables occurring in all data types are the simple variable, the array reference, and the pointer reference. Array and pointer references specify an element by means of a subscript or index; the index must always be a one-word value (byte, integer, or logical). The index value specifies an element index, not a word index. It is loaded into the index register and used in an indexed memory reference instruction. Note that this may change the value of the condition code. If no index is specified, the reference is to the zero element, which is more efficient than explicitly specifying 0 as the index since the index register is not used.

4-3. TOS

TOS is a reserved symbol that always refers to the top of the stack; it can be used anywhere a variable can be used. When TOS is used on the left side of an assignment statement (TOS:=*expression*), the normal store operation is omitted and the result is left on the top of the stack. If TOS occurs in an expression, the contents of the top of the stack are used as the next operand. TOS must be used carefully, since the compiler does not keep track of the number of elements pushed onto the stack prior to encountering TOS. The data type of TOS is determined by context; it takes the type of the expression or other operand. Thus, in one context TOS might refer to the top word, in another the top four words. Note that TOS does not refer to the same memory location from one statement to the next, since S is constantly changing. The default type for TOS is integer. A general rule for determining the effect of TOS is to assume that TOS is a variable and then delete all LOAD and STOR operations for TOS. For example,

```
TOS:= 7;          <<LOAD 7>
A:= TOS+ 6;       <<A:= 13>>
```

4-4. ADDRESSES (@) AND POINTERS

When @ precedes a simple variable, it specifies that the DB-relative address of the simple variable is desired. All addresses are signed, one-word integers and are treated as such in expressions. When @ precedes an array identifier, it refers to the DB- or PB-relative address of the zero element of the array (whether direct or indirect). When @ precedes an array reference (*identifier(index)*), it refers to the DB- or PB-relative address of the array element. When @ precedes a pointer identifier, it refers to the address contained within a pointer cell; when an index is specified, @ refers to the address of the data element relative to the zero element pointed at by the pointer. For example,

```
BEGIN
    INTEGER A;
    INTEGER ARRAY B(0:10);
    POINTER P:=@B(5);
        A:=@A;  <<A assigned address of A>>
        A:=@P;  <<A assigned address of B(5)>>
        A:=@B;  <<A assigned address of B(0)>>
END.
```

If the @ construct is used on the left of an assignment operator, it must be used with either a *pointer-name* or an *array-name* of an indirect array and an *index* cannot be specified. This usage changes the address which the pointer contains. For arrays, this means that there is a new zero element. For example,

```
@ A:= @ A(1);
```

would make A(1) the new A(0). For pointer variables, the statement:

```
@ P:= @ B;
```

changes P to point to the location assigned to B. The various combinations using the @ construct and pointers are summarized in figure 4-1.

```
POINTER P1,P2;
LOGICAL VAR;

P1:=P2;          <<The object of P2 is stored into the object of P1>>
P1:=@P2;         <<The address in P2 is stored into the object of P1>>
@P1:=@P2;        <<The address in P2 is stored into P1>>
@P1:=P2;         <<The object of P2 is stored into P1>>
P1:=VAR;         <<The value of VAR is stored into the object of P1>>
P1:=@VAR;        <<The address of VAR is stored into the object of P1>>
@P1:=@VAR;       <<The address of VAR is stored into P1>>
@P1:=VAR;        <<The value of VAR is stored into P1>>
VAR:=P1;         <<The object of P1 is stored into VAR>>
VAR:=@P1;        <<The address in P1 is stored into VAR>>
```

Figure 4-1. Pointers and Addresses

4-5. ABSOLUTE ADDRESSES

The ABSOLUTE construct can only be executed in privileged mode. It provides access to the contents of an absolute memory location. The address (*index*) is loaded into the index register. If ABSOLUTE appears on the left side of an assignment statement (ABSOLUTE(*index*):=*expression*), a PSTA (privileged store) instruction is generated which stores the top of the stack (*expression*) in the absolute memory location specified by the index register. If ABSOLUTE appears within an expression, a PLDA (privileged LOAD) instruction is generated which loads onto the stack the contents of the absolute location specified by the index register. For example,

```
LOGICAL L1,L2,L3;
INTEGER A1,A2,A3=X;
  .
  .
  .
L1:=ABSOLUTE(A1*A2);
ABSOLUTE(L2):=A1+5;
ABSOLUTE(A3):=A1+5; <<A3 is the index register>>
L1:=ABSOLUTE(ABSOLUTE(3));
L1:=ABSOLUTE(A3);
```

4-6. FUNCTION DESIGNATOR

Function designators are another of the possible components of an expression. A function designator specifies a function (a typed procedure or subroutine) to be executed and a list of actual parameters (values or addresses) to be passed to the function. The function returns a value of the appropriate data type to the place in the expression where it was called.

The form of a function-designator is:

name [([*actual-parameter*] [,...,[*actual-parameter*]])]

NOTE

An *actual-parameter* can be omitted only if OPTION VARIABLE is specified in the procedure declaration.

EXAMPLES:

```
F(*,A,B(2))
G(C+ 3,I:= I+ 1,D< E)
```

where

name
is the name of the function procedure or subroutine to be executed.

An *actual-parameter* is one of the following:

identifier [(*index*)]
arithmetic-expression
logical-expression
assignment-statement
*

identifier
is a *simple-variable, array-name, pointer-name, procedure-name,* or *label.* The DB- or PB-relative address is passed to the function. PB-relative arrays cannot be passed as parameters. An *identifier* must be used if the formal parameter is not used in a VALUE statement within the procedure or subroutine.

index
specifies an array or pointer element. The *index* is an expression or an assignment statement of type INTEGER, LOGICAL, or BYTE. If an *index* is not specified for an array or pointer, then zero is assumed.

arithmetic-expression logical-expression and *assignment-statement*
are evaluated and the result is passed as a call-by-value parameter. The forms for these items are described fully later in this section.

The function procedure or subroutine must have been previously declared (see "Procedure Declaration" and "Subroutine Declaration" in section VII). The actual parameters must match the formal parameters one-to-one as specified in the declaration; correspondence is checked left-to-right. An actual parameter may be omitted only if OPTION VARIABLE has been specified in the procedure declaration.

A stacked parameter is specified by an asterisk (*) to indicate that you have already loaded the necessary address or value onto the stack. Labels cannot be stacked. If any parameter is stacked, all parameters to its left must also be stacked. In addition, functions require that a 1-, 2-, or 4-word zero (depending on the function type) be pushed onto the stack before the function parameters to reserve space for the return value. Normally, the compiler provides this zero automatically; however, if stacked parameters are used, you must arrange for this zero. For example,

```
INTEGER PROCEDURE COMPUTE(N);VALUE N;...;
  ASSEMBLE (ZERO);
  TOS:= A;
  B:= COMPUTE(*)+ 1000;
```

For more details on calling procedures and subroutines, see "Procedure Call Statement" and "Subroutine Call Statement" in paragraphs 5-8 through 5-13.

Procedure calls use the PCAL instruction and subroutine calls use the SCAL instruction.

4-7. BIT OPERATIONS

Bit operations can be used in any type of expression. Bit extraction is the extraction of a contiguous bit field starting at a particular bit position. Bit concatenation consists of extracting a bit field from a specified position in one quantity and depositing it at a specified position in another quantity. Bit shifts allow values to be shifted left or right, arithmetically, circularly, or logically. All bit operations are performed on copies of the specified quantities so that the original variables remain unchanged.

A simple-variable of type BYTE is stored in bits 0-7. However, before performing a bit operation, the value is loaded onto the stack into bits 8-15. Therefore, bit operations using BYTE simple-variables should use bits 8 through 15 instead of 0 through 7.

Bit extraction and concatenation are defined for one-word quantities only. Bit shifts are provided for one-, two-, three-, and four-word quantities. See "Assignment Statement" later in this section for bit deposit.

4-8. BIT EXTRACTION

The purpose of bit extraction is to isolate a contiguous bit field from the 16 bits of a one-word value. The result is a right justified value with leading bits set to zero. The maximum field that can be extracted in a single operation is 15 bits. Bit extraction uses the EXF (extract field) instruction. Extraction starts with the bit of the source specified by left-source-bit and continues to the right for the number of bits indicated by length, wrapping around to bit 0, if necessary.

The form of a bit extraction is:

source **.** **(*left-source-bit* : *length*)**

EXAMPLES:

```
A.(8:3)
A(I).(15:1)
```

where

source
is a single-word integer, logical, or byte primary from which the bits are extracted. Refer to paragraphs 4-11 and 4-14 for the definition of primary.

left-source-bit
specifies the bit of the source word at which the extraction begins. The *left-source-bit* is any unsigned decimal, based, composite, or equated integer constant from 0 to 15 inclusive.

length
specifies the number of bits to be extracted. The *length* is any unsigned decimal, based, composite, or equated integer constant from 1 to 15 inclusive.

See figure 4-2 for a sample bit extraction.

4-9. Bit Concatenation (Merging)

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 *left-dest-bit* indicates in which bit position of the destination primary to deposit the field extracted from the source primary. The *left-source-bit* indicates at which position in the source primary to begin extracting the bit field. The length indicates how many contiguous bits to extract and subsequently deposit. Bit concatenation uses both the EXF (extract field) and DPF (deposit field) instructions which are described in the *Instruction Set Reference Manual*.

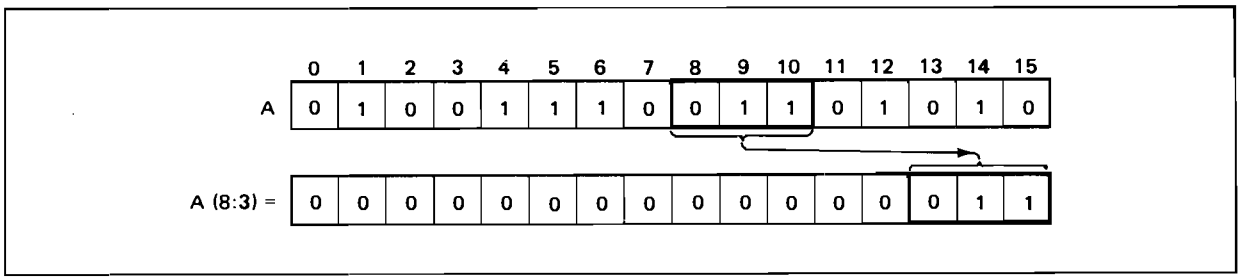


Figure 4-2. Bit Extraction

The form of a bit concatenation is:

***destination* CAT *source* (*left-dest-bit* : *left-source-bit* : *length*)**

EXAMPLES:

A CAT B(8:4:2)
A CAT %23(6:11:5)
%(16)69A2 CAT %(16)ABCD (8:4:4)

where

source
specifies the item from which bits are extracted. The *source* is a single-word integer, logical, or byte primary (defined under "Arithmetic Expressions" and "Logical Expressions" later in this section).

destination
specifies the value into which bits are deposited. The *destination* is a single-word integer, logical, or byte primary (defined under "Arithmetic Expressions" and "Logical Expressions" later in this section).

left-source-bit
specifies the starting bit position of the bit extraction. It is an unsigned decimal, based, composite, or equated integer constant whose value is between 0 and 15 inclusive.

left-dest-bit
specifies the starting bit position of the bit deposit. It is an unsigned decimal, based, composite, or equated integer constant whose value is between 0 and 15 inclusive.

length
specifies the number of bits to be copied. The *length* is an unsigned decimal, based, composite, or equated integer constant whose value is between 1 and 15 inclusive.

See figure 4-3 for a sample bit concatenation.

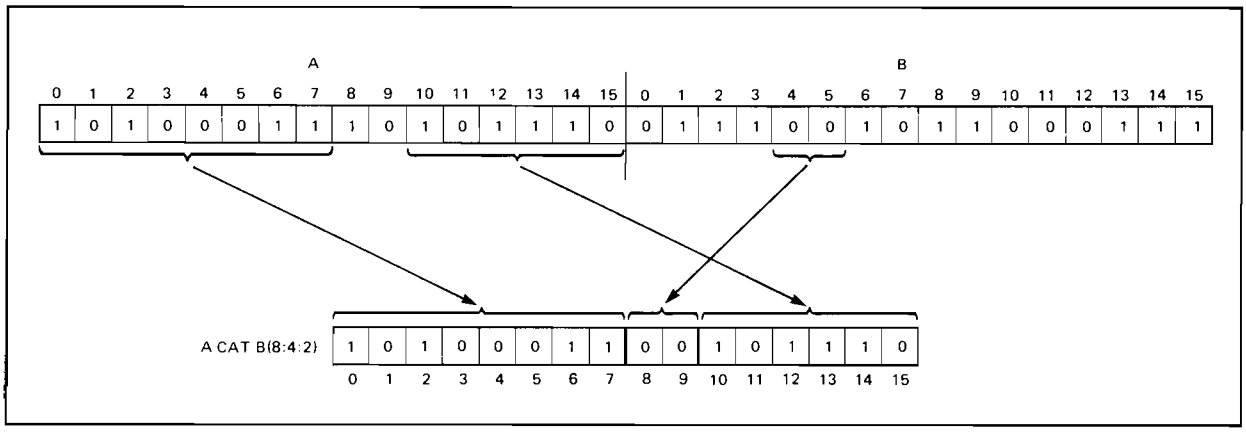


Figure 4-3. Bit Concatenation

4-10. BIT SHIFTS

In the bit shifts, the *shift-op* is a mnemonic for a hardware shift operation. Consult the hardware documentation for complete details. In general, 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 propagate the sign bit on a right shift (in other words, fill with the sign bit); and circular shifts do not have a fill bit (that is, bits shifted off one end are shifted in at the other end). SPL does not perform type or word size tests. If a multiple-word shift is specified, you are responsible for ensuring that the proper number of words (2, 3, or 4) is on the stack. Note that if the shift count is not a constant less than 64, the index register is used.

The form of a bit shift is:

operand **&** *shift-op* **(***shift-count***)**

EXAMPLES:

```
(A:= A+ 1) & LSR(3)
VAR & DASL(6)
%1234D & DCSL(SHIFT)
```

where

operand
is an arithmetic or logical primary of any SPL type (see "Arithmetic Expressions" and "Logical Expressions" later in this section).

shift-op
specifies the shift operation to be performed. The *shift-op* is one of the following: LSL, LSR, ASL, ASR, CSL, CSR, DASL, DASR, DLSL, DLSR, DCSL, DCSR, TASL, TASR, TNSL, QASR, or QASL.

shift-count
specifies the number of bits to be shifted. The *shift-count* is an integer expression (described in "Arithmetic Expressions" later is this section).

The meanings of the *shift-op* mnemonics are shown below:

LSL	Logical Shift Left
LSR	Logical Shift Right
ASL	Arithmetic Shift Left
ASR	Arithmetic Shift Right
CSL	Circular Shift Left
CSR	Circular Shift Right
DASL	Double Arithmetic Shift Left
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
TASL	Triple Arithmetic Shift Left
TASR	Triple Arithmetic Shift Right
TNSL	Triple Normalizing Shift Left
QASR	Quadruple Arithmetic Shift Right
QASL	Quadruple Arithmetic Shift Left

See figure 4-4 for some sample bit shift operations.

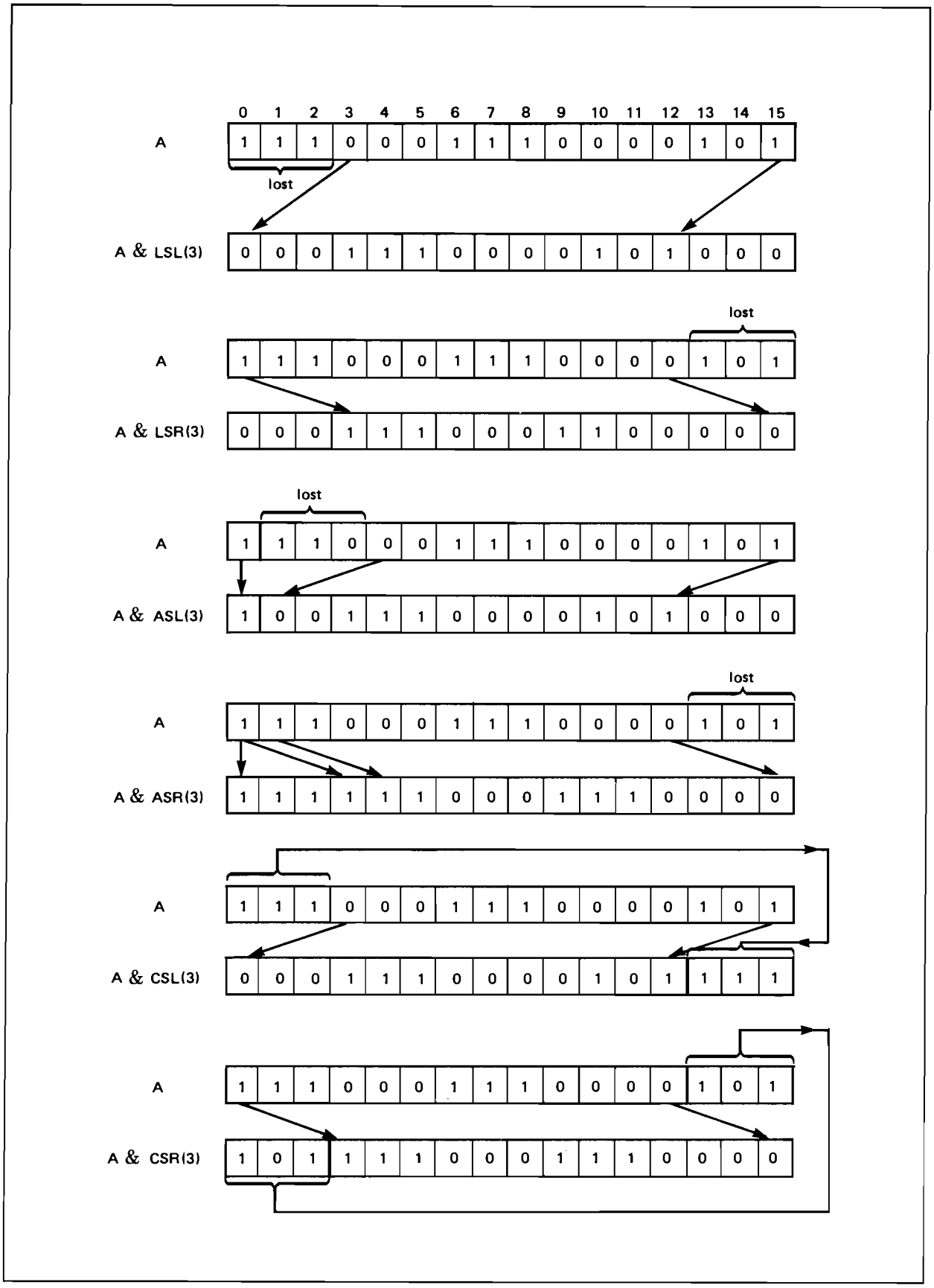


Figure 4-4. Bit Shift Operations

4-11. ARITHMETIC EXPRESSIONS

An arithmetic expression is a sequence of operations upon numeric data which results in a single-value of a specific data type. Execution of operators occurs left-to-right unless higher precedence operators or parentheses are encountered. Type mixing of operands across operators is not allowed, but type transfer functions are provided. Primaries, the basic components of an arithmetic expression, can be constants, variables, bit expressions, arithmetic expressions in parentheses or backward slashes (absolute value), function designators, or assignment statements in parentheses.

The form of an *arithmetic-expression* is:

[*sign*] *primary* [*operator primary ... operator primary*]

EXAMPLES:

```
A+(B*C)/2.0
-A^2+F(B)
\I+3\
(I:=I+1)+(J:=J+1)-2
A(10:2)+B CAT C (8:4:4)
I
```

where

sign
is + or −.

operator
is +, −, *, /, ^, or MOD.

primary
is one of the following:

variable
constant
bit operation
(arithmetic expression)
\arithmetic expression
function-designator
(assignment statement)

NOTE

Allowable exponentiation combinations are:

integer ^ integer
real ^ real
real ^ integer
long ^ long
long ^ integer

variable
designates an item whose value is determined at execution time and can be dynamically changed. The form of a *variable* is described earlier in this section.

constant
designates a value which is established at compile-time and cannot change during execution. The various constant types and their forms are described in section II.

bit-operation
is a *bit-extraction, bit-concatenation,* or *bit-shift* as described earlier in this section. The value used in the expression is the result obtained after performing the *bit-operation.*

function-designator
specifies a call to a procedure which returns a value. The form of a *function-designator* is described earlier in this section.

assignment-statement
specifies that an expression is to be evaluated and the result assigned to a variable or variables before being used in the evaluation of the outer expression. The form of the *assignment-statement* is described later in this section.

4-12. SEQUENCE OF OPERATIONS

Arithmetic operations are ranked in order of precedence to determine the relative order in which operations are executed. Higher precedence operations are performed first. When operations are of the same rank, execution proceeds from left to right. The ranks, from highest to lowest, are:

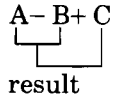
1. Bit operations
   Expressions in parentheses
   Expressions in backward slashes (absolute value)
   Function designators
   Assignment statements in parentheses
   (value assigned to variable and left on the stack)

2. Exponentiation (^, circumflex character)
   (defined for integer, real, and long data, plus real to integer power and long to integer power)

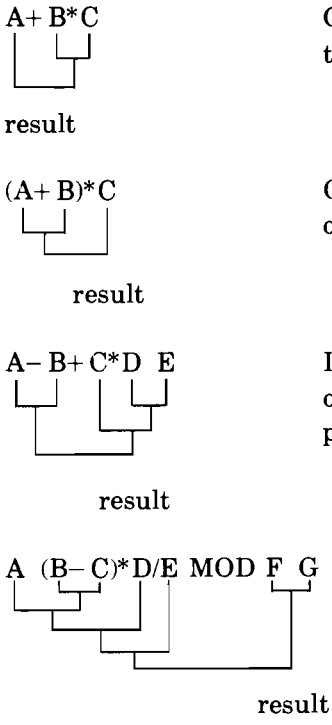
3. Multiply (*) and divide(/) for integer, real, byte, double, and long data.
   Modulo (MOD) or remainder for integer, byte, and double data.

4. Addition (+) and subtraction (−) for integer, real, byte, double, and long data.

The order in which operations are performed is determined by this rank. For example,



Operators of the same rank are performed from left to right.



Operators of different rank are performed according to their position in the hierarchy of operators (highest rank first).

Operators enclosed in parentheses take precedence over operators outside of parentheses, even those of higher rank.

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.

4-13. TYPE MIXING

Mixing of data types across operands is not allowed in SPL, except that real and long values can be exponentiated to integer powers. Type transfer functions are available to handle conflicts (see "Expression Types" earlier in this section).

The type of operands determines the type of both the operation result and the operator used. Integer operations are used when the operands are of type byte.

4-14. LOGICAL EXPRESSIONS

Logical expressions are evaluated in the same manner as arithmetic expressions. However, logical expressions use more and different operators; allow only data of type LOGICAL and provide special constructs, such as byte comparisons. The result of a logical expression is a logical value which can be interpreted as a 16-bit unsigned integer or as true (odd) or false (even). The truth value of a logical expression can be used to make decisions (see "IF Statement" in paragraph 5-6). Logical primaries can be logical constants, variables, bit expressions, expressions in parentheses, functions, or assignment statements in parentheses, or the complement of any logical primary. The operators LAND (Logical AND) and LOR (Logical OR) should not be confused with AND and OR as used in the IF Statement.

The form of a logical-expression can be either of the following:

1. *logical-element* [*operator logical-element*]

2. *lower-value* <= *test-value* <= *upper-value*

EXAMPLES:

```
L
L + NOT L1 LAND L2
1<= N <= 100
L < L1
L XOR L1 MOD L2
```

where

logical-element
is one of the following:

logical-expression

logical-primary [*relational-operator logical-primary*]

arithmetic-expression relational-operator arithmetic-expression

logical-primary logical-operator logical-primary

byte-compare

operator
is LOR (Logical OR), XOR (Logical Exclusive OR), or LAND (Logical AND).

relational-operator
is >,<,=,<>,>=, or <=.

logical-primary
is any of the following:

logical *variable*
logical or integer *constant*
string *constant*
logical *bit-operation*
(*logical-expression*)
logical *function-designator*
(logical *assignment-statement*)
NOT *logical-primary*

logical-operator
is +,–,/,MOD,**,//, or MODD.

byte-compare
is a comparison of a byte array with another byte array, a string constant or constants, or a test of the character type of a byte variable. See paragraph 4-17.

lower-value
is the lower bound of a range comparison. The *lower-value* is an integer expression.

test-value
is the value which is tested for being within the range of the lower and upper values. The *test-value* is an integer expression.

upper-value
is the upper bound of a range comparison. The *upper-value* is an integer expression.

The *relational-operators* have the following meanings:

Operator	Meaning
<	Less than
<=	Less than or equal to
=	Equal to
<>	Not equal to
>	Greater than
>=	Greater than or equal to

The purpose ot a logical expression is to evaluate certain conditions and relations to produce a value which can be interpreted either arithmetically (as a 16-bit positive number) or logically (as either TRUE or FALSE). A logical expression is not a statement of fact, but an assertion that may be true or false at any given time.

Logical quantities in SPL are 16-bit positive integers (see paragraph 2-7). A logical value is true if its integer value is odd, false if its value is even (that is, only bit 15 is checked). The reserved words TRUE and FALSE are equivalent to the numeric values −1 and 0 (%177777 and %000000) respectively.

In general, the result of a logical expression is left as a full word operand on the top of the stack. This result is either a −1 or 0 when a relational operator is encountered. However, when the result of a relational operator is used in a condition clause to make a decision (see IF Statement), the result is not left on the stack but the condition code in the status register is set.

4-15. SEQUENCE OF OPERATIONS

Logical operations are ranked in order of precedence to determine the order in which the operations are performed. Higher precedence operations are performed first. When operations are of the same precedence, execution proceeds from left-to-right. All operands and results are type LOGICAL, unless otherwise noted. There are seven ranks of operations as shown below:

1. Logical bit operation
   Logical-expression in parentheses
   Logical function-designator
   Logical assignment statement in parentheses
   NOT (unary one's complement)

2. `*` (Logical multiply, one-word result)
   `/` (Logical divide, one-word dividend)
   MOD (Logical modulo or remainder, one-word dividend)
   `**` (Logical multiply, result is type double)
   `//` (Logical divide, dividend is type double)
   MODD (Logical modulo or remainder, dividend is type double)

   Note: The MOD and MODD operations divide the dividend by the divisor, discarding the quotient and yielding the remainder as the result. See example with the assignment statement, paragraph 4-20.

3. `+` (Logical addition)
   `-` (Logical subtraction)

4. Algebraic and logical comparisons (=,<>,<,>,<=,>=)
   Byte comparisons and tests

5. LAND (Logical and)

6. XOR (Logical exclusive or)

7. LOR (Logical inclusive or)
   Integer range test (such as, I <= J <= K)

4-16. TYPE MIXING

You cannot mix data types across operands in SPL; however, type transfer functions are available to handle conflicts. In logical expressions, logical operands are used except when the both operands are arithmetic and the result is logical (compares, byte tests, and range tests). See paragraph 4-1 for the type transfer functions.

4-17. COMPARING BYTE STRINGS

Logical expressions provide a mechanism for comparing byte strings to determine whether a particular relation between them is true or false. The test is made using the CMPB (compare bytes) instruction. The byte strings are compared, byte by byte, using their numeric values until the compared bytes are unequal or until a specified number of comparisons has been made. If the specified relation (<,>,=,<=,>=, or <>) holds, the result is TRUE (−1); otherwise, it is FALSE (0).

The form of a byte-compare is one of the following:

byte-reference relational-operator byte-reference ,(count) [*,stack-decrement*]

byte-reference relational-operator *PB,*(count)* [*,stack-decrement*]

byte-reference relational-operator string-constant [*,stack-decrement*]

byte-reference relational-operator (value-group,..., value-group) [*,stack-decrement*]

byte-variable { = / <> } [ALPHA / NUMERIC / SPECIAL]

EXAMPLES:

```
A<B,(5),2
B(5) >= *PB,(5)
* <= "ABC"
A <> NUMERIC
```

where

byte-reference
is one of the following:

1. *array-name* [*(index)*]
2. *pointer-name* [*(index)*]
3. *

array-name
is an *identifier* declared in an array declaration.

pointer-name
is an *identifier* declared in a pointer declaration.

index
is either an expression or an assignment statement of type integer, logical, or byte. If an *index* is not specified, then zero is assumed.

count
is the number of bytes to compare. The *count* is an integer expression. A positive *count* specifies left-to-right comparison and a negative *count* specifies right-to-left.

stack-decrement
indicates how many words to delete from the stack after the compare. The *stack-decrement* is an unsigned integer constant between 0 and 3 inclusive. If not specified, a *stack-decrement* of 3 is used.

value-group
is either of the following:

constant
repetition-factor (constant [,...,constant])

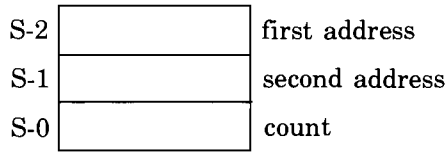
repetition-factor
specifies the number of times the constant list is used before going to the next *value-group*. The *repetition-factor* is an unsigned decimal, based, composite, or equated single-word integer constant.

The string to the left of the relational operator can be specified by a byte pointer or array reference (DB-relative only) or a stacked DB byte address (*). The asterisk specifies that you have already loaded the byte address onto the stack.

The string to the right of the relational operator can be specified by a byte pointer or array reference (DB- or PB-relative), a stacked DB address (*), a stacked PB address (*PB), a string constant, or a list of constants in parentheses.

The absolute value of the count specifies how many bytes to compare. A positive count specifies left-to-right comparison while a negative count specifies right-to-left comparison.

The *stack-decrement* specifies how many values to delete from the stack at the end of the compare operation. If a *stack-decrement* is not specified, all three values are deleted. The contents of the stack during the comparison are shown below:



Byte comparisons can be passed by-value as parameters to procedures and subroutines; however, some extra requirements apply:

1. If a *stack-decrement* is allowed but not specified and the *byte-comparison* is not the last actual parameter, the *byte-comparison* must be enclosed in parentheses. For example,

   P(A,(B<C,(3)),2);

2. Byte comparisons which use stacked values must be enclosed in parentheses and all parameters to the left must be stacked prior to stacking the values to the *byte-comparison*. For example,

   P(*,(*=*,(5)));

4-18. CONDITION CLAUSES

Condition clauses are used in IF expressions, IF statements, DO statements, and WHILE statements. Two types of operands are used in condition clauses: logical-expressions and hardware branch words. Both types of operands result in a value of true or false. These operands can be combined using AND and OR. If two items are combined with OR, the result is true if either item is true or if both items are true. If two items are combined with AND, the result is true only if both items are true. AND has higher precedence than OR, but you can use parentheses around OR'ed expressions to override this precedence. Parentheses cannot be used around items combined with AND.

The form of a *condition-clause* is:

condition-term [{ AND / OR } *condition-term*... { AND / OR } *condition-term*]

EXAMPLES:

```
(A<B OR A<C) AND (A1<B1 OR A1<C1)
CARRY AND A<>B OR A<>C
L1 LAND L2 < L1 LAND L3 OR I<=J
<
OVERFLOW
```

where

condition-term
is either of the following:
condition-primary
(condition-primary [OR *condition-primary*]...OR *condition-primary)*

condition-primary
is either true or false. The *condition-primary* is one of the following:
branch-word
logical-expression

branch-word
is one of the following: CARRY, NOCARRY, OVERFLOW, NOVERFLOW, IABZ, DABZ, IXBZ, DXBZ,=,<>,<,>,<=, or >=.

The hardware branch words test the Status Register, the Index Register, or the Top of Stack as shown below:

BRANCH WORD	TRUE CONDITION
CARRY	Carry bit on (Status Register)
NOCARRY	Carry bit off (Status Register)
OVERFLOW	Overflow bit on (Status Register)
NOVERFLOW	Overflow bit off (Status Register)
IABZ	Increment TOS. True if TOS is then 0.
DABZ	Decrement TOS. True if TOS is then 0.
IXBZ	Increment Index Register (X). True if X is then 0.
DXBZ	Decrement Index Register (X). True if X is then 0.
<	Condition Code equals 1 (Status Register).
=	Condition Code equals 2 (Status Register).
<=	Condition Code equals 1 or 2 (Status Register).
>	Condition Code equals 0 (Status Register).
<>	Condition Code equals 0 or 1 (Status Register).
<=	Condition Code equals 0 or 2 (Status Register).

OR and AND generate branch instructions instead of arithmetic ANDs and ORs. All parts of a condition are not always executed since OR and AND branch out of the condition as soon as the truth value of the condition is determined. For example, if a series of items is joined by ANDs and the first item is false, the whole condition is false so the remaining items are not checked.

NOTE

The CARRY and OVERFLOW bits are cleared after being tested. The Condition Code, Index Register, and TOS are unaffected by being tested.

Extreme care must be taken when using the SPL condition clause to check condition codes returned from intrinsics. The IF>, IF< ..... constructs are only correct if no machine instruction that sets condition code is executed between the setting and checking the condition code. The LDX, XCH, STAX instructions, for example, are all used when SPL indexes into arrays. All of these modify the condition code.

```
a(275)  := fopen();
                    00021    LOAD P+000
                    00022    ZERO, NOP
                    00023    ADDS, 016
                    00024    LDI, 000
                    00025    PCAL, 000
                    00026    XCH, STAX
                    00027    STOR PB 001,1,X
if<> then quit(0);
```

The IF statement in the above example does not test the condition code for the FOPEN procedure. It reflects the condition code set by the XCH, STAX instruction.

4-19. IF EXPRESSIONS

Expressions are used to determine values to be used in statements. The IF expression consists of a *condition-clause* and two alternative expressions. The *condition-clause* is a combination of logical expressions and hardware branch words which results in a true or false value. The two expressions must be of the same word size (byte is treated as one word). If the *condition-clause* is true, the value of the IF expression is the value of the expression after the THEN; if the *condition-clause* if false, the value of the IF expression is the value of the expression after the ELSE. The definition of *condition-clause* is given earlier in this section.

The form of an IF expression is:

IF *condition-clause* THEN *true-value* ELSE *false-value*

EXAMPLES:

```
IF A<B THEN 5 ELSE 6*B
IF < THEN 1 ELSE 0
FACT:=IF N=0 OR N=1 THEN 1 ELSE N*FACT(N-1);
```

where

condition-clause
determines which value to use as the value of the expression. The form of a *condition-clause* is described earlier in this section.

true-value
is the value of the expression if the *condition-clause* is true.

false-value
is the value of the expression if the *condition-clause* is false.

4-20. ASSIGNMENT STATEMENT

The assignment statement stores the result of an expression evaluation into a variable of the same size. Multiple assignments allow the same result to be stored in several variables. Bit deposits allow a one-word result to be stored into a variable starting at a specific bit position.

The form of an assignment statement is:

variable[.(*left-deposit-bit:length*)] :=
[*variable:=...variable:=*] *expression*

EXAMPLES:

```
I:=K*L;
I(5:6):=J:=L;
I(0:8):=B1;
R1:=R1:=R1+(R2*REAL(I));
D:=R1;
A(I:=I+1):=I*2;
```

where

variable
designates the item(s) to which the value of the expression is assigned. The form of a *variable* is described earlier in this section.

left-deposit-bit
specifies the starting bit position of a bit deposit. The *left-deposit-bit* is an unsigned decimal, based, composite, or equated integer constant between 0 and 15 inclusive.

length
specifies the number of bits to be stored. The *length* is an unsigned decimal, based, composite, or equated integer constant between 1 and 15 inclusive.

expression
is evaluated to determine the value to store into the variable(s) on the left of the assignment operator. The *expression* is an arithmetic or logical-expression whose result is the same word size, although not necessarily the same data type, as the variable(s).

The result of the expression evaluation is stored in the variable(s) specified on the left side of the assignment operator (:=) or (__). Blanks cannot be embedded between the colon and the equals sign of an assignment operator. The result must be the same word size, but not necessarily the same data type, as the assignment variable. Type BYTE is treated as a one-word quantity.

When a deposit field is specified, the expression result must be a one-word quantity. The rightmost *n* bits of the result, where *n* is the deposit field length, are stored in the variable starting with the bit position specified. Note that only the leftmost assignment can be a deposit field.

An assignment statement can be used as a term in an expression. In this case, the result of the expression in the assignment statement is first stored into the variable(s) and then used as the value of the term in the outer expression. For example, the statement:

```
J:= K+ (I:= I+ 1)- M;
```

is equivalent to the sequence of statements:

```
I:= I+ 1;
J:= K+ I- M;
```

Note that a semicolon is not used to terminate an assignment statement used within an expression.

Assignment statements can also be used as array or pointer subscripts and as call-by-value parameters to procedures and subroutines. Array subscripts on the left side of an assignment statement can be evaluated either before or after the expression on the right side of the assignment statement depending on the complexity of the subscript. Therefore, you should avoid changing the value of a variable on the right side of an assignment statement if the variable is used as a subscript on the left of the assignment statement. For example,

```
A(I):= B(I:= I+ 1);
```

is not evaluated the same as:

```
A(I+ 0):= B(I:= I+ 1);
```

In the first case, I is incremented and then used as the subscript for both B and A. In the second case, the original value of I is used as the subscript of A. In general, if a subscript which is used on the left side of an assignment statement is evaluated without using the top of the stack, the evaluation of the subscript is done just prior to storing the value in the array element. Subscripts in this category include:

Simple variables	(I)
Increment by one	(I:= I+ 1)
Decrement by one	(I:= I- 1)
Addition of zero	(I:= I+ 0)
Subtraction of zero	(I:= I- 0)

For example,

```
A(I:= I+ 1):= B(I:= I+ 2);
```

is evaluated as:

```
I:= I+ 1;
I:= I+ 2;
A(I):= B(I);
```

Note that if the left-side subscript is itself an assignment statement, it is executed before the right side of the outer assignment statement is evaluated even though the subscript used to determine the element being stored into may not be evaluated until afterwards. However, if the left side subscript

uses the top of the stack, the evaluation of the right side expression does not effect the value of the left side subscript. For example,

```
A(I:= I+ 2):= B(I:= I+ 1);
```

is evaluated the same as:

```
I:= I+ 2;
I:= I+ 1;
A(I- 1):= B(I);
```

If in doubt, you can use the $CONTROL INNERLIST option to check the code which the compiler generates (see paragraph 9-2).

The following examples illustrate the use of assignment statements involving type DOUBLE data and the logical operators**, //, and MODD:

```
LOGICAL L1:= 20000, L2:= 2, L3:= 3;
DOUBLE D1;
D1:= L1**L2 << D1:= 40000D >> ;          Product
L4:= D1//L3 << L4:= 13333 >> ;           Quotient
L5:= D1 MODD L3 << L5:= 1 >> ;           Remainder
```

Care should be taken to ensure that the result of the logical operators // and MODD is a one-word quantity. Any other result causes an integer overflow.

4-21. MOVE STATEMENT

The MOVE statement moves words or bytes from one location to another. The locations can be either DB- or PB-relative. Move operations do not change the contents of the source. There are three types of move operations corresponding to the three types of hardware move instructions:

- Move words (MOVE, MVBL, and MVLB)
- Move bytes (MVB)
- Move bytes while alphabetic and/or numeric with or without upshifting (MVBW)

The MOVE statement can also perform as an arithmetic function by returning the number of bytes or words moved. In this case, it can be used anywhere an integer function is appropriate; however, no stack-decrement is allowed in order to avoid possible corruption of the stack with the use of expressions.

The two forms of a move statement are:

MOVE *destination* := { *source,(count)* / **[PB], (count)* / *string* / *(value-group-list)* } [,*stack-decrement*]

and

MOVE *destination* := { *source* / * } WHILE *condition* [, *stack-decrement*]

EXAMPLES:

```
MOVE OUT:=IN,(10),2;
MOVE OUT:=*PB,(-10);
MOVE OUT:=(10(" "),"STRING",5(" ")),1;
MOVE OUT:=IN WHILE AN;
MOVE OUT:=* WHILE N;
MOVE *:=* WHILE ANS;
```

As an arithmetic function:

```
I := MOVE P:=P1, (<LENGTH>);
IF P(MOVE P:=P1 WHILE ANS) = "xyz" THEN...;
MOVE P :=P1, (SCAN P1(SCAN P1 UNTIL "  ") UNTIL "  ");
```

where

destination
specifies the starting location to be stored into. The *destination* is one of the following:

array-name[(*index*)]
pointer-name[(*index*)]
*

source
specifies the starting location of the item to be copied. The *source* is either of the following:

array-name[(*index*)]
pointer-name[(*index*)]

NOTE

Destination and source addresses are byte addresses for byte moves and word addresses for word moves.

array-name
is an *identifier* declared in an array declaration.

pointer-name
is an *identifier* declared in a pointer declaration.

index
is either an expression or an assignment statement of type integer, logical, or byte. If an *index* is not specified, then zero is assumed.

count
is the number of bytes or words to move. The *count* is an integer expression. A positive *count* specifies left-to-right move and a negative *count* specifies right-to-left.

stack-decrement
indicates how many words to delete from the stack after the move. The *stack-decrement* is an unsigned integer constant between 0 and 3 inclusive for a MOVE and between 0 and 2 inclusive for a MOVE WHILE. If not specified, a *stack-decrement* of 3 is used for a MOVE and 2 for a MOVE WHILE.

value-group-list
is either of the following:

value-group
value-group, value-group-list

value-group
is either of the following:

constant
repetition-factor (constant [,...,*constant*])

repetition-factor
specifies the number of times the constant list is used before going to the next *value-group*. The *repetition-factor* is an unsigned decimal, based, composite, or equated single-word integer constant.

condition
specifies the criteria for continuing the move to the next character. The *condition* is one of the following: A,N,AS, AN, or ANS.

The move statements in SPL are machine dependent because they are based on specific hardware instructions.

The first reference after the MOVE is the *destination*; the item after the assignment operator (:=) is the *source*. INTEGER, REAL, LONG, and DOUBLE arrays use the move words instructions whereas BYTE arrays use the move bytes instructions. When the *source* is a string or a list of constants, the constants are generated in the code stream and moved from there. The syntax for the list of constants is the same as for a list of constants used to initialize an array in an array declaration.

Where * or *PB appears in place of an address, the DB- or PB-relative address must have been previously loaded onto the stack by the user. The *source* can be PB-relative except when the MOVE...WHILE statement is used. The *destination* cannot be PB-relative. If both addresses are stacked, a byte move is assumed.

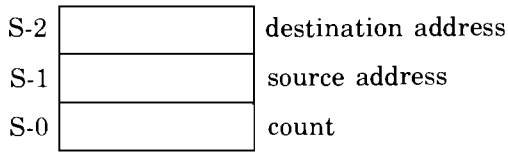
The *count* is an integer expression that specifies the number of words or bytes to move; a positive *count* indicates a left-to-right move and a negative *count* indicates a right-to-left move. At the completion of the move, the *count* equals zero and the addresses have been changed to point to the character following the last character moved.

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 *stack-decrement* operand is then used to delete 0, 1, 2, or all 3 of the parameters from the stack. A blank *stack-decrement* field generates an automatic *stack-decrement* of 3 — delete all three values from the stack. Count always equals 0 and can safely be deleted (sdec = 1). The *stack-decrement* mechanism is used for all move-scan statements.

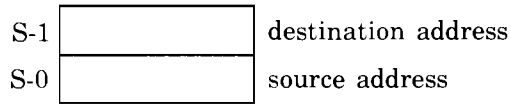
The following code sample illustrates the use of the *stack-decrement* operand to return the number of words or bytes moved.

```
BEGIN

    INTEGER LEN;
    BYTE ARRAY BUFF (0:20);
    MOVE BUFF:="ABCDEFGHIJKLMNO",2; << 2=RETAIN DESTINATION ADDRES<
    LEN:=TOS-LOGICAL(@BUFF);

END
```

The stacked values used by the move words and move bytes instructions are shown below:



The stacked values used for a move bytes while instruction are:



In a MOVE ... WHILE statement, the *condition* specifies the condition for continuing the move to the next character. The *conditions* are shown below:

A	Current character is alphabetic
N	Current character is numeric
AS	Current character is alphabetic; upshift if lower case
AN	Current character is alphabetic or numeric
ANS	Current character is alphabetic or numeric; upshift if lower case

WARNING

The normal checks and limitations that apply to the standard users in MPE are bypassed in privileged mode. It is possible for a privileged mode program to destroy system integrity, including the MPE operating system software itself. Hewlett-Packard cannot be responsible for system integrity when programs written by users operate in privileged mode.

4-21A. MOVEX STATEMENT

The MOVEX instruction is intended specifically for privileged users requiring extra data segments (see section 8-1, split-stack mode). It facilitates the writing of high-level code increasing its reliability. This instruction performs word moves only, not byte moves. Three machine instructions relating to data segments are generated, depending on the move. They are as follows:

MFDS	Move from extra data segment to stack
MTDS	Move to extra data segment from stack
MDS	Move between extra data segments

If the move is confined to a single data segment, a DB-relative MOVE is generated. Please refer to section 3-10 for information about DATASEG declarations.

The form of a MOVEX statement is:

```
MOVEX (destination [,offset]) :=
      (source [,offset]), (length) [,stack-decrement];
```

EXAMPLES:

```
MOVEX (D,9) := (D1, I+J), (K), 6;
MOVEX (99, I+J/2) := (K*M,L), (99);
```

where

destination and *source*
specify the starting location of the words to be moved (source), and the starting location where the words will be stored (destination). Locations must be one of the following:

Either DB-relative pointers (for MFDS and MTDS), DATASEG or DATASEG-relative identifiers (for static XDS moves), or integer expressions (for dynamically calculated XDS numbers). In the latter case, DATASEG-relative identifiers are not permitted in the expression.

offset
(Optional) The beginning offset into the XDS. It can be either a constant or an integer expression that is valid within any containing $SPLIT or WITH. An offset is not permitted when the pointer is DB-relative (as opposed to DATASEG-relative).

length
is the number of words to be moved.

stack-decrement
is an unsigned integer constant indicating how many words to delete from the stack after the move. The default value is 5 for MFDS and MTDS, and 4 for MDS. For any extra data segment move, the maximum value is 7. If a stack-decrement larger than 3 is specified for a DB-relative move, a warning is generated and 3 is used.

4-22. SCAN STATEMENT

The SCAN statement is used to search for either of two specified characters (the test and terminal characters) in a contiguous string of bytes without actually moving any data. When the statement ends, pointers and indicators are left to show what was found and where. The scan statements in SPL are machine-dependent because they are based on specific hardware instructions. There are two scan operations corresponding to the two hardware scan instructions:

- Scan until a test character is found (SCU instruction).
- Scan while a test character is found (SCW instruction).

The SCAN statement can also be used as an arithmetic function to return the number of bytes or words scanned. In this case, it can be used anywhere an integer function is appropriate; however, no stack-decrement is allowed in order to avoid possible corruption of the stack with the use of expressions.

The form of the SCAN statement is:

SCAN *byte-reference* WHILE *testword* [,*stack-decrement*]

SCAN *byte-reference* UNTIL *testword* [,*stack-decrement*]

EXAMPLES:

```
SCAN BUF WHILE TEST;
SCAN BUF(2) WHILE %6440,1;
SCAN * UNTIL ".;";
SCAN BUF UNTIL *,0;
```

As an arithmetic function:

```
I := SCAN P UNTIL "  ";
```

where

byte-reference
is one of the following:

array-name [(*index*)]
pointer-name [(*index*)]
*

array-name
is an *identifier* declared in an array declaration.

pointer-name
is an *identifier* declared in a pointer declaration.

index
is either an expression or an assignment statement of type integer, logical, or byte. If an *index* is not specified, then zero is assumed.

testword
is one of the following:

A decimal, based, composite, or equated single-word integer constant.
A *simple-variable* of type INTEGER or LOGICAL.
"test-character"
"terminal-character test-character"
*

terminal-character
is any ASCII character. Note that " is represented by "".

test-character
is any ASCII character. Note that " is represented by "".

stack-decrement
indicates how many words to delete from the stack after the SCAN. The *stack-decrement* is an unsigned integer constant between 0 and 2 inclusive. If not specified, a *stack-decrement* of 2 is used.

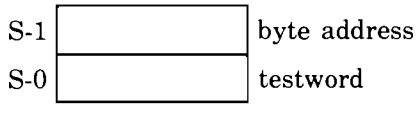
The *byte-reference* which specifies where to start scanning can be a byte array reference, a byte pointer reference, or an asterisk (*) to indicate that the DB-relative address is already on the stack. PB-relative arrays cannot be scanned. If either an array or pointer reference is specified, the address is loaded onto the stack.

The *testword* is an integer or logical simple variable, an integer constant, or a one- or two-character string where the first character (bits 0 through 7) specifies the *terminal-character* and the second character (bits 8 through 15) specifies the *test-character.* If no *terminal-character* is specified, bits 0 through 7 are zero-filled. In both cases, each byte in the two-character string is tested against both the test and terminal characters.

In a SCAN UNTIL, the scan continues until either the *test-character* or the *terminal-character* is found. In a SCAN WHILE, the scan continues until a byte is found that matches the *terminal-character* or does not match the *test-character*. The carry bit in the status register is set to 0 after a scan to indicate that the *test-character* was found; it is set to 1 to indicate the *terminal-character* was found. This bit can be tested with the IF statement:

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

The carry bit is cleared after being tested. The *stack-decrement* specifies how many words to delete from the stack after the scan operation. The *stack-decrement* is very important in a scan operation because when the scan terminates, the address of the terminating byte can be left in the stack. The stack for a SCAN UNTIL or a SCAN WHILE appears as shown below:



A *stack-decrement* of 1 deletes the testword but leaves the byte address which can be saved as follows:

```
SCAN'STOP:= TOS;
```

An empty *stack-decrement* field generates a *stack-decrement* of 2 and leaves the stack as it was before the scan statement.

The following code sample illustrates the SCAN UNTIL operation. After the last statement shown, the pointer is pointing to the first "0" character.

```
BYTE POINTER PTR;
BYTE ARRAY CHAR (0:30) := "AAAAAAAAAAAAAA0AAAAAAAAAAAAAAA";
SCAN CHAR UNTIL "Z0" ,1;
@PTR := TOS;
```

In the SCAN WHILE example below, the address of PTR will point to the first non-'A' character.

```
BYTE POINTER PTR;
BYTE ARRAY CHAR (0:30) := "AAAAAAAAAAAAAA0AAAAAAAAAAAAAAA";
SCAN CHAR WHILE "ZA" ,1;
@PTR := TOS;
```

PROGRAM CONTROL STATEMENTS

SECTION V

5-1. PROGRAM CONTROL

Program execution normally proceeds sequentially from statement to statement. By using control statements, you can alter this sequence by transferring control to another statement, by executing a group of statements (a procedure or a subroutine) and then returning to the original flow, or by repeating a pre-determined group of statements. Statements in a program to which control is to be passed are labeled by identifiers preceding the statement. A colon (:) is used to separate the label from the statement. Procedures and subroutines are named by identifiers in declarations (see section VII).

This section covers the following control statements:

- GO TO statement
- DO statement
- WHILE statement
- FOR statement
- IF statement
- CASE statement
- Procedure call statement
- Subroutine call statement
- RETURN statement

5-2. GO TO STATEMENT

The GO TO statement is used to transfer control to a labeled statement. There are two forms of the GO TO statement: the unconditional form and the indexed form. When an unconditional GO TO statement is executed, control is transferred to the statement specified. An indexed GO TO statement is used to invoke a switch to selectively transfer to one of several statements.

The form of a GO TO statement is one of the following:

1. **GO [TO]** ***label***

2. **GO [TO] [*]** ***switch-name (index)***

EXAMPLES:

```
GO TO START;
GO OUT;
GOTO FINIS(A+B-2);
GO *SW(I:=I+1);
```

where

label
identifies the statement to which control is transferred. The *label* is an *identifier* which is used to label a statement other than an entry-point.

switch-name
identifies the switch to be invoked. The *switch-name* is an identifier which has been declared in a switch declaration.

index
indicates which *label* in the switch declaration is to be used. The *index* is an expression or assignment statement whose result is a single-word value.

The three forms GO, GOTO, and GO TO are equivalent. In an indexed GO TO statement, bounds checking is performed on the index value unless an asterisk (*) is used before the *switch-name*.

The object of a GO TO statement in the main-body must be a global *label* or *switch-name* and the object of a GO TO statement in a procedure or subroutine must be a local *label* or *switch-name*. You cannot use a GO TO statement to transfer into a procedure and you can only use a GO TO statement to transfer out of a procedure if the *label* has been passed to the procedure as a parameter. Switches cannot be passed as parameters.

Switches are invoked using an indexed GO TO statement; the *index* is an integer value that specifies the label desired. Labels in a switch declaration are numbered consecutively starting with 0. Normally, if the index value is less than zero or greater than the number of labels minus one, control is transferred to the statement following the GO TO statement. However, if the asterisk option is specified, bounds checking is not performed and invalid indexes cause unpredictable results. When a switch is invoked, the index value is stored in the index register.

NOTE

A switch cannot be invoked within a subroutine nor can any labels assigned to a switch appear in a subroutine.

5-3. DO STATEMENT

The DO statement is used to repeatedly execute a statement until a specified *condtion-clause* becomes true. When the *condition-clause* is true, control is transferred to the next statement after the DO statement.

The form of the DO statement is:

DO *loop-statement* UNTIL *condition-clause*

EXAMPLES:

```
DO A(I:= I+ 1):= I*2 UNTIL I>23;
DO BEGIN
     I:= I+ 1;
     IVAL(I):= I/(X*Y+ 3);
     BVAL(I):= (X*Y+ 3)/I;
   END
UNTIL I>20;
```

where

loop-statement
is the statement which is executed each pass through the loop. The *loop-statement* may be either a simple or compound statement including another DO statement.

condition-clause
determines whether or not to execute the *loop-statement* another time. See paragraph 4-18 for the form of a condition-clause.

Note that a semicolon is not used to separate the loop-statement from the reserved word UNTIL.

After the *loop-statement* is executed, the *condition-clause* is evaluated and tested. If the *condition-clause* is false, the *loop-statement* is executed again; if the *condition-clause* is true, control is transferred to the statement following the DO statement. The *condition-clause* is evaluated and tested after each execution of the *loop-statement* (the *loop-statement* is always executed at least once).

5-4. WHILE STATEMENT

The WHILE statement is used to repeatedly execute a statement as long as a specified *condition-clause* is true. The WHILE statement differs from the DO statement in that the *condition-clause* is tested before executing the *loop-statement* instead of after and the condition-clause must be true for the *loop-statement* to be executed instead of false. When the *condition-clause* is false, control is transferred to the statement following the WHILE statement.

The form of the WHILE statement is:

WHILE *condition-clause* DO *loop-statement*

EXAMPLES:

```
WHILE I<21 DO A(I:=I+1):=2-I;
WHILE 0<=N<=100 LAND NOT Q="/" DO
    BEGIN
      Q:=C5(I);
      I:=I+1;
      N:=N*I;
    END;
```

where

condition-clause
determines whether or not to execute the *loop-statement*. See paragraph 4-18 for the form of a *condition-clause*.

loop-statement
is the statement which is executed each pass through the loop while the *condition-clause* is true. The *loop-statement* may be either a simple or compound statement including another WHILE statement.

The *condition-clause* is always tested before executing the *loop-statement*. Thus, if the *condition-clause* is false on the first pass, the *loop-statement* will not be executed at all. The *condition-clause* consists of logical-expressions and hardware branch words as described in paragraph 4-18. However, the following branch words have different meanings when used in a WHILE statement:

IABZ	Increment TOS. Execute *loop-statement* if TOS is non-zero.
DABZ	Decrement TOS. Execute *loop-statement* if TOS is non-zero.
IXBZ	Increment the index register. Execute *loop-statement* if the index-register is non-zero.
DXBZ	Decrement the index register. Execute *loop-statement* if the index-register is non-zero.

5-5. FOR STATEMENT

The FOR statement is used to repeatedly execute a statement, changing an integer *test-variable* by a specified amount each time, until the test variable exceeds a specified limit. The FOR statement uses hardware loop control instructions which require special stack markers so you should be very careful when performing your own stack manipulation within a FOR statement.

The form of a FOR statement is:

```
FOR [*] test-variable := starting-value [STEP step-value]
UNTIL ending-value DO loop-statement
```

EXAMPLES:

```
FOR I:=3 UNTIL LIM DO A(I):=I*2;
FOR *I:=1 STEP 2 UNTIL LIM DO
    SUM:=SUM+NARN(I);
FOR I:=MAX STEP -RANGE/4 UNTIL MAX-RANGE DO
    BEGIN
      FOFI:=A*I-2+B*I+C;
      SUM:=SUM+FOFI;
    END;
```

where

test-variable
is the variable which is altered by the *step-value* each pass through the loop and is tested for exceeding the *ending-value*. The *test-variable* is an integer *simple-variable*.

starting-value
is the value assigned to the *test-variable* before the first pass through the loop. The *starting-value* is an INTEGER, LOGICAL, or BYTE expression.

step-value
is the amount by which the *test-variable* is changed each time the loop is executed. The *step-value* is an INTEGER expression. If omitted, a *step-value* of 1 is used.

ending-value
is the value against which the *test-variable* is tested each pass through the loop to determine whether or not to execute the *loop-statement* again. The ending-value is an integer expression.

loop-statement
is the statement which is executed each pass through the loop. The *loop-statement* may be either a simple or compound statement including another FOR statement.

The *starting-value, step-value,* and *ending-value* are calculated once upon entry into the FOR statement. The *starting-value* is stored into the *test-variable* and tested before the *loop-statement* is first executed. After each execution of the *loop-statement*, the variable is changed by the *step-value* and compared with the *ending-value*. If the *step-value* is positive and the *test-variable* is less than or equal to the *ending-value*, the *loop-statement* is executed again. If the *test-variable* is greater than the *ending-value*, control is transferred to the statement after the FOR statement. For negative *step-values*, the loop is executed again if the *test-variable* is greater than or equal to the *ending-value*. After the FOR statement is executed, the *test-variable* contains the value which exceeds the *ending-value*.

Thus, the statement:

```
FOR J:=1 UNTIL 10 DO ...;
```

executes the *loop-statement* 10 times and J has a value of 11 when the loop is completed.

You can use an asterisk (*) after FOR to specify that the *loop-statement* is to be executed once without testing the *test-variable* against the *ending-value*. This guarantees that the *loop-statement* is executed at least once even if the *starting-value* is past the *ending-value*.

CAUTIONS in the Use of FOR Statements

If the *test-variable* is equivalenced to the index register, the TBX and MTBX instructions are used for loop-control; otherwise, the TBA and MTBA instructions are used. Since all of these instructions use values placed in the stack, if you alter the stack during the execution of the *loop-statement*, unpredictable results may occur. Additionally, if you exit a FOR statement, for example, with a GO TO or RETURN, from within the *loop-statement*, the *test-variable* address, the *step-value*, and the *ending-value* are left on the stack. If the index register is used as the *test-variable*, any operation within the *loop-statement* which changes the index register, such as array referencing, can destroy the loop control.

Therefore, it would be prudent for the 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.) Executing a CASE statement embedded in a FOR loop will modify the index register.

Table 5-1. Comparison of DO, WHILE, and FOR Statements

COMPARISON OF DO, WHILE, AND FOR STATEMENTS
DO STATEMENT The *condition-clause* is evaluated and tested after the *loop-statement* is executed. The *loop-statement* is repeated if the *condition-clause* is false. The *loop-statement* is always executed at least once.
WHILE STATEMENT The *condition-clause* is evaluated and tested before the *loop-statement* is executed. The *loop-statement* is executed if the *condition-clause* is true. The *loop-statement* is not always executed at least once.
FOR STATEMENT The *test-variable* is checked before the *loop-statement* is executed. The *loop-statement* is executed if the *test-variable* is less than or equal to the *ending-value* (for positive *step-values*) or greater than or equal to the *ending-value* (for negative *step-values*). The *loop-statement* is always executed at least once if an asterisk is specified after the reserved word FOR.

5-6. IF STATEMENT

The IF statement is used either to execute one of two alternative statements or to execute or skip a single statement based on whether a *condition-clause* is true or false.

The form of an IF statement is:

IF *condition-clause* THEN *true-branch* [ELSE *false-branch*]

EXAMPLES:

```
IF A<B THEN MAX:=B ELSE MAX:=A;
IF I>100 THEN GO TO L1;
IF A<B AND A<C THEN
   BEGIN
     MIN:=A;
     GO TO L2;
   END;
```

where

condition-clause
determines whether or not to execute the *true-branch*. The form of a *condition-clause* is described in paragraph 4-18.

true-branch
is the statement which is executed if the *condition-clause* is true. The *true-branch* may be either a simple or a compound statement including another IF statement.

false-branch
is the statement which is executed if the *condition-clause* is false. The *false-branch* may be either a simple or compound statement including another IF statement.

There are two forms of the IF statement: single-branch and double-branch. The single-branch IF statement is used when the two alternatives are to execute a statement or not to execute a statement. If the *condition-clause* is true, the statement is executed and control proceeds to the statement after the IF statement, unless the true-branch has tranferred to another statement with a statement such as a GO TO or RETURN. If the *condition-clause* is false, the *true-branch* statement is not executed and control is transferred to the statement after the IF statement. For example,

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

The double-branch IF statement is used to select one of two alternative statements. If the *condition-clause* is true, the *true-branch* statement is executed. If the *condition-clause* is false, control is

transferred to the *false-branch* statement. When the selected statement has been executed, control is transferred to the statement after the IF statement except when a transfer has been executed from the selected statement with, for example, a GO TO or RETURN statement. Some sample double-branch IF statements are shown below:

```
IF A<B THEN XA:=XA+A
       ELSE XA:=XA+B;
IF TESTVAR THEN Y:=Y+1
            ELSE IF EXTRATEST THEN Y:=Y-1;
IF TEST THEN A:=A+B ELSE A:=A-B;
```

Note that you cannot use a semicolon between the *true-branch* and the reserved word ELSE.

IF statements can be indefinitely nested. The innermost THEN is paired with the closest following ELSE and pairing proceeds outward. For example,

```
IF condition-clause
   THEN
     IF condition-clause
       { THEN
         { IF condition-clause
           { THEN true-branch
           { ELSE false-branch
       { ELSE false-branch;
```

In the above example, the outermost IF statement is a one-branch IF statement.

As noted in paragraph 4-18, logical expressions and/or branch words can be combined using AND and OR to form a *condition-clause*. These connectors should not be confused with the logical connectors LAND and LOR which are used within logical expressions. If two items are combined with OR, the result is true if either item is true or if both items are true. If two items are combined with AND, the result is true only if both items are true. AND has higher precedence than OR, but you can use parentheses around OR'ed expressions to override this precedence. Parentheses cannot be used around items combined with AND.

5-7. CASE STATEMENT

The CASE statement is used to select one of a set of statements for execution by using an index value into a compound statement. The statements of the compound statement are assigned index values consecutively starting with 0 and incrementing by 1. After the selected statement has been executed, control is transferred to the statement after the CASE statement unless a transfer is executed in the selected statement such as a GO TO or RETURN statement.

The form of a CASE statement is:

CASE [*] *index* OF BEGIN *statement* [;...;*statement*] END

EXAMPLE:

```
CASE J OF
  BEGIN
     A:= 100;
     B:= 200;
     BEGIN
       C:= 300;
       IF A<B THEN D:= 100
     END;
     QR:= 500
  END;
```

where

index
determines which statement to execute. The *index* is an INTEGER, LOGICAL, or BYTE expression.

statement
is any simple or compound executable statement including another CASE statement. Null statements are allowed.

Bounds checking on the index value is normally performed to insure that the index is between 0 and n− 1 inclusive (where n is the number of statements in the body of the CASE statement). However, if you do not want bounds checking to be performed, you can specify an * before the *index*. If the asterisk option is specified, an invalid index will cause unpredictable results.

To transfer control immediately to the next statement, use a null statement in the case body. For example,

```
CASE J OF
  BEGIN
    A:= 100;
    ; <<NULL statement; NO ACTION, BUT HOLDS PLACE>>
    C:=200
  END;
```

If J equals 0, statement A:=100 will be executed.
If J equals 1, control is transferred to the statement after the CASE statement.
If J equals 2, the statement C:=200 is executed.
If J ≥3, then the next statement following the CASE statement is executed.

The CASE statement uses the index register to store the index value.

5-8. PROCEDURE CALL STATEMENT

The procedure call statement is used to transfer control to a previously declared procedure and pass a list of actual parameters to it. When a procedure is completed, control normally returns to the statement following the call; however, the procedure can override this return (see "Passing Labels as Parameters", paragraph 5-11).

The form of a procedure call statement is:

procedure-name **[((*actual-parameter*][,...,[actual-parameter]])]**

NOTE

An *actual-parameter* can be omitted only if OPTION VARIABLE is specified in the procedure declaration.

EXAMPLES:

```
COMPUTE (R+ 23.0,L2,PROC5);
COMPUTE (*,,P4);
REVERSE;
```

where

procedure-name
identifies the procedure to which control is transferred. The *procedure-name* is an *identifier* which has been declared either in a procedure-declaration as a *procedure-name* or *entry-point* or in an intrinsic-declaration.

actual-parameter
is one of the following:

identifier[(*index*)]
arithmetic-expression
logical-expression
assignment-statement
*

identifier
identifies a call-by-reference parameter. The following items can be passed: *simple-variables, array-names, pointer-names, procedure-names, entry-points,* and *labels.*

index
denotes an array or pointer element. The *index* is an expression or an assignment statement of type INTEGER, LOGICAL, or BYTE and can only be specified for *array-names* and *pointer-names*. If an *index* is not specified, the zero element is used.

arithmetic-expression, logical-expression, and *assignment-statement*
are evaluated to pass a value as a call-by-value parameter. The forms for these items are described in paragraphs 4-11 through 4-17 and 4-20.

The * is used to indicate that you have already put the parameter onto the stack. See paragraph 7-4 for a discussion of the correspondence between the actual-parameters in a procedure-call and the formal-parameters in a procedure-declaration.

If a function procedure is called using a procedure call statement instead of a function-designator in an expression, the return value is deleted from the stack upon returning to the calling routine unless the procedure overrides the normal return.

Two types of parameter passing are allowed in SPL: by reference and by value. A call-by-reference parameter places an address onto the stack. A data item (simple-variable, array-element, or pointer-element) which is passed by reference can have its value changed in the calling environment by changing its value in the procedure. A call-by-value parameter is passed by evaluating the parameter at the time of the procedure call and placing this value onto the stack. If a parameter is passed by value, changes to the parameter value in the procedure will not alter the value of the parameter in the calling environment.

When a procedure call statement is executed, the actual parameters are loaded onto the stack and a PCAL instruction is executed. The PCAL instruction places a four-word stack marker onto the stack, changes the Q-register to point to the top of this stack marker, and transfers control to the entry-point of the procedure. The stack marker contains the following information:

Q-3	Index Register
Q-2	Return address
Q-1	Status Register
Q-0	delta Q

The return address is P+ 1− PB where P is the value of the P register when the PCAL instruction is executed and PB is the base register for the code segment. The delta Q is the number of words between the new value of Q and the previous value of Q.

Because of the stack architecture, recursive procedures (that is, procedures which call themselves) are allowed.

5-9. STACKING PARAMETERS

Stacked parameters may be either call-by-reference or call-by-value. For call-by-reference parameters, you must put the address of the *actual-parameter* onto the stack. For example,

```
TOS:=@A;
```

For call-by-value parameters, you must put the value of the *actual-parameter* onto the stack. For example,

```
TOS:=I+2;
```

If any parameter is stacked, all parameters to its left must also be stacked. For example,

```
P(*,*,B,C);
```

Labels cannot be stacked. Before stacking parameters for a call to a function procedure, you must push a one-,two-,or four-word zero, depending on the data type of the function, onto the stack for the return value. This zero is generated automatically if no parameters are stacked. For example, assume P is a REAL procedure which has two call-by-reference parameters. The following steps are needed if you want to stack the parameters:

```
TOS:=0D;
TOS:=@A;
TOS:=@B;
P(*,*);
```

5-10. MISSING PARAMETERS IN PROCEDURE CALLS

If the procedure is declared with OPTION VARIABLE, parameters can be omitted from the actual-parameter list by leaving a comma to hold their place or by using a right parenthesis to terminate the list if you want to omit the parameters at the end of the formal-parameter list. For example, consider the procedure declaration:

```
PROCEDURE P(A,B,C,D,E,F);...;OPTION VARIABLE;...
```

To pass only the first parameter, use a procedure call such as

```
P(R);
```

To pass the first and last parameters, use a procedure call such as

```
P(R1,,,,,R2);
```

If you want to omit all parameters, you can use either of the following:

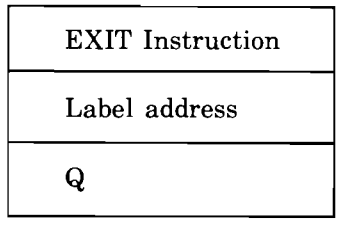
```
P; or P();
```

The called procedure is responsible for checking the existence of actual parameters. See paragraph 7-9 for a discussion of how to perform this checking.

5-11. PASSING LABELS AS PARAMETERS

Labels may be passed to procedures as call-by-reference parameters to allow control to transfer to a place other than the normal return address upon completion. Unlike other call-by-reference parameters, however, a label is passed as a three-word label descriptor. If a label is passed to several levels of procedure calls (such as A calls B which calls C), the label descriptor allows you to transfer to the label without executing an EXIT instruction for each procedure through which the label was passed; only the first procedure which received the label parameter is exited. This technique can be very useful for error processing.

The label descriptor contains the following information:



The first word of the label descriptor is an exit instruction to exit the first procedure to which the label is passed. The second word is the address of the label. The third word is the value of the Q register upon entry to the first procedure to which the label is passed.

When a transfer to a label which was passed as a parameter is executed, the following steps are performed:

1. The label descriptor is put on the top of the stack.
2. The Q register is reset to the value in TOS (which is the value it had upon entry to the first procedure).
3. The label address is stored in Q−2 (the return address location for the first procedure).
4. The exit instruction on the top of the stack is executed to effectively exit the first procedure and transfer control to the label.

The following situation is illustrated in figure 5-1:

a. The main body calls procedure A and passes the label L as a parameter.

b. Procedure A calls procedure B and passes an integer variable I by-value and the label L as parameters.

c. While in procedure B, a transfer to L is executed —

   1. The label descriptor is loaded onto the stack.
   2. The Q register is reset to Q (A).
   3. The address of L is stored into Q−2 overriding the normal return address from A back to the main body.
   4. The EXIT instruction in S−0 is executed to:
      1. Reset Q to the main body value.
      2. Delete the stack marker for A and the label descriptor passed to A.
      3. Tranfer control to L.

If the first procedure is a function procedure, the space for the return value is left on the stack should you not perform a normal return, but transfer to a place other than where the call was made.

5-12. PASSING PROCEDURES AS PARAMETERS

Procedures may be passed to other procedures as call-by-reference parameters. The Load Label (LLBL) instruction is used to load the external address of the procedure onto the stack. When calling a procedure which was passed as a parameter, the parameters are assumed to be call-by-reference. To pass call-by-value parameters to such a procedure, you must stack them before calling the procedure and use the * in the procedure call. A procedure which has been declared with OPTION VARIABLE requires a special technique for being passed to another procedure and then called. Such procedures

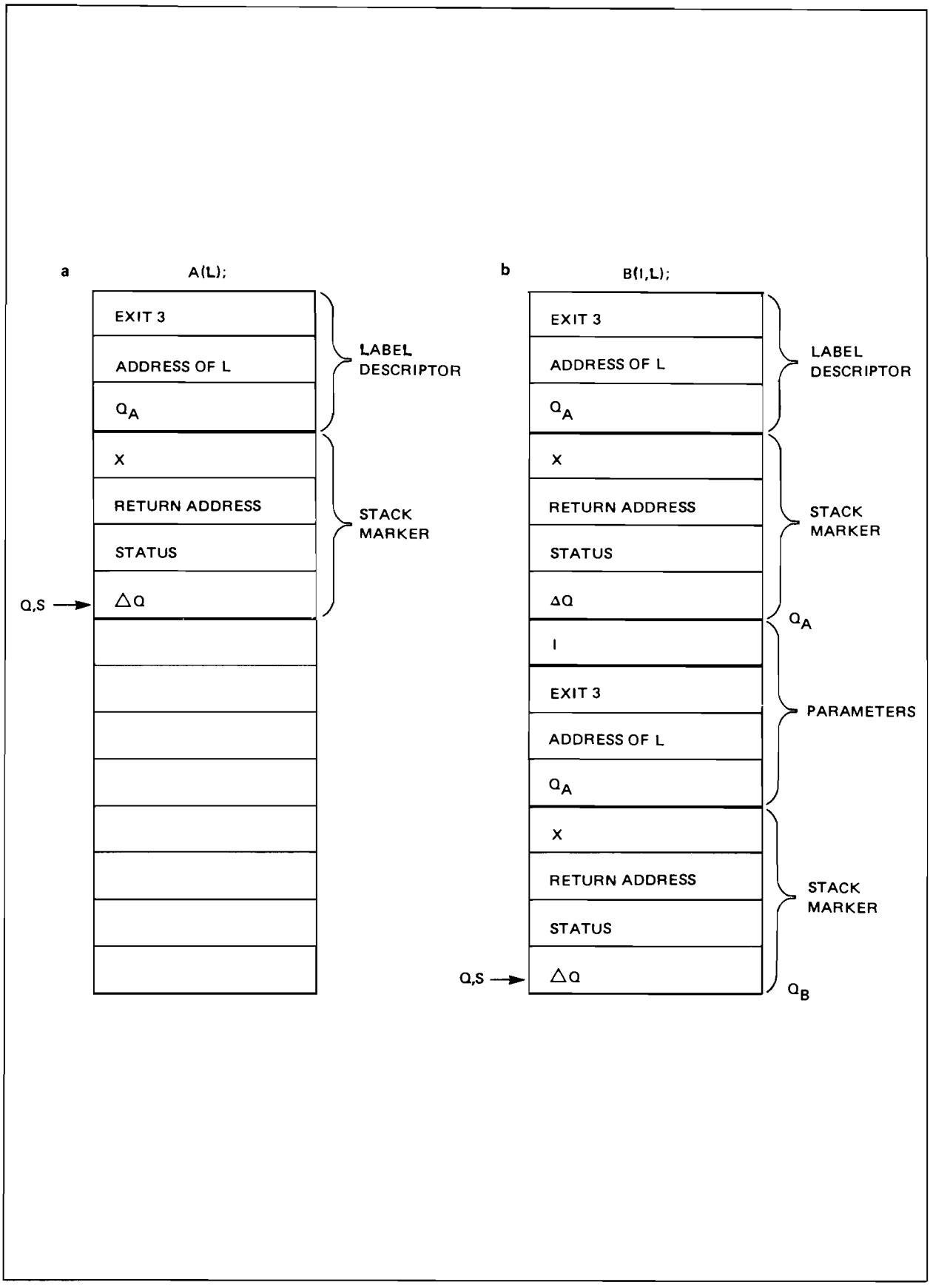


Figure 5-1. Passing a Label as a Parameter

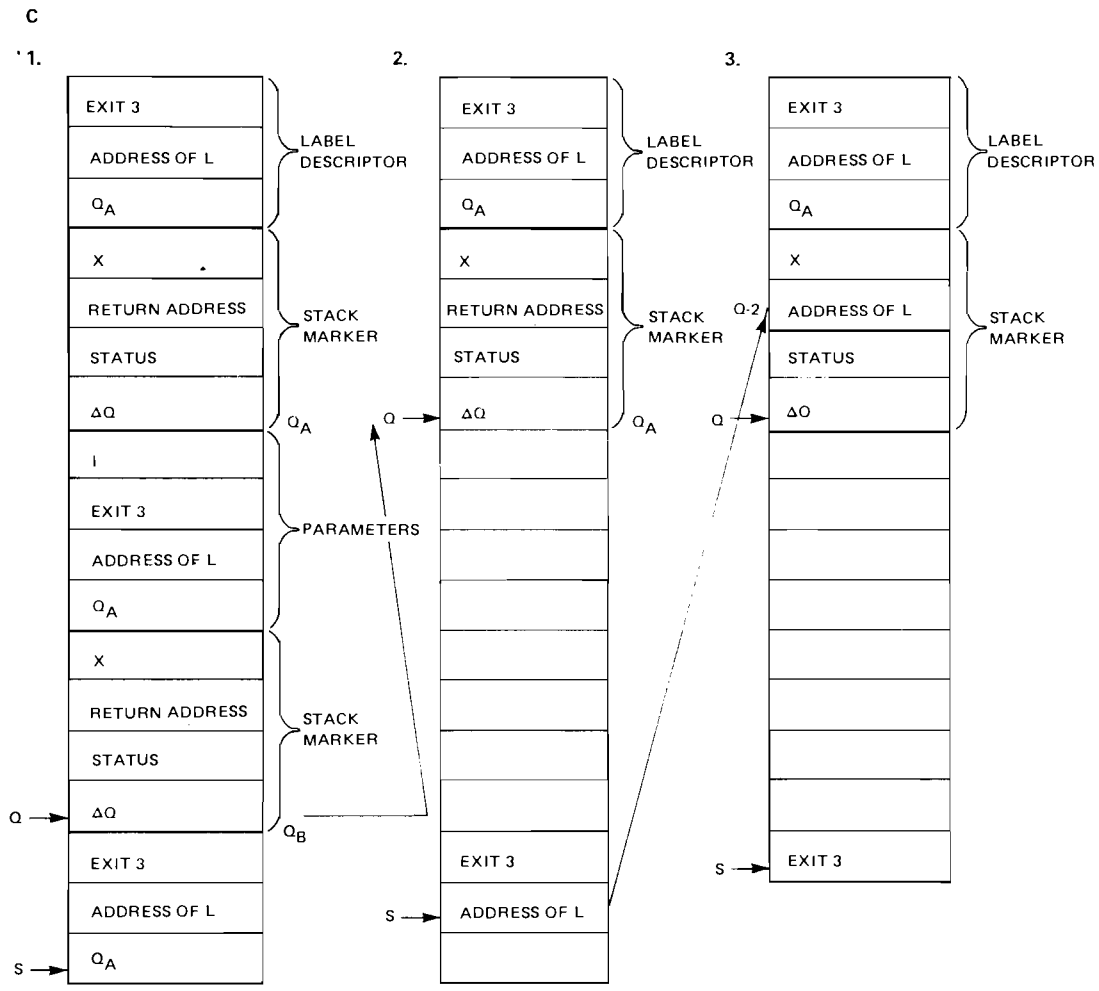


Figure 5-1. Passing a Label as a Parameter (Continued)

require a bit mask in Q-4, and Q-5 if there are more than 16 formal parameters. If you call such a procedure you must generate your own bit mask. For example, consider the declarations:

```
PROCEDURE P(A,B);...;OPTION VARIABLE;...
PROCEDURE P1(F); PROCEDURE F;
```

If P is passed as an actual parameter to P1, such as:

```
P1(P);
```

Then, a call to P within P1 would look like

```
F(A,B,3);
```

where 3 is the bit mask indicating that both parameters are present. Since the last parameter is a constant instead of an address reference, a warning message is issued. An alternative method is to stack all parameters and the bit mask:

```
TOS:=@ A;
TOS:=@ B;
TOS:= 3;
F(*,*);
```

For further discussion of OPTION VARIABLE procedures, see paragraph 7-10.

5-13. SUBROUTINE CALL STATEMENT

The subroutine call statement is used to invoke a previously declared subroutine and pass a list of actual parameters to it. When a subroutine is completed, control normally returns to the statement following the call; however, the subroutine can override this return. A global subroutine can branch to a label in the main body and a local subroutine can branch to a label in the procedure body.

The form of a subroutine call statement is:

subroutine-name **[(*actual-parameter*[,...,*actual-parameter*])]**

EXAMPLES:

```
S(A+B,B,C);
S(*,*,C);
S1;
```

where

subroutine-name
identifies the subroutine to which control is transferred. The *subroutine-name* is an *identifier* which has previously been declared in a subroutine declaration.

actual-parameter
is one of the following:

identifier[(*index*)]
arithmetic-expression
logical-expression
assignment-statement
*

identifier
identifies a call-by-reference parameter. The following items can be passed: *simple-variables, array-names, pointer-names, procedure-names,* and *entry-points.*

index
denotes an array or pointer element. The *index* is an expression or assignment statement of type INTEGER, LOGICAL, or BYTE and can only be specified for *array-names* and *pointer-names*. If an index is not specified, the zero element is used.

arithmetic-expression, logical-expression, and *assignment-statement*
are evaluated to pass a value as a call-by-value parameter. The forms for these items are described in paragraphs 4-11 through 4-17 and 4-20.

The * is used to indicate that you have already put the parameter onto the stack. See paragraph 7-4 for a discussion of the correspondence between the actual parameters in a subroutine call and the formal parameters in a subroutine declaration.

Note that a label cannot be passed as a parameter to a subroutine nor can parameters be omitted (OPTION VARIABLE cannot be specified for a subroutine). Alternate entry points are not allowed in subroutines.

If a function subroutine is called using a subroutine call statement instead of a function-designator in an expression, the return value is deleted from the stack upon returning to the calling routine unless the subroutine overrides the normal return.

When a subroutine call statement is executed, the actual parameters are loaded onto the stack and an SCAL instruction is executed. (SCAL may be replaced with an LRA and a BR.) The SCAL instruction puts the return address onto the stack and transfers control to the subroutine entry-point. The Q-register is not changed — all parameters are addressed using S-negative addressing. Recursive subroutines (that is, subroutines which call themselves) are allowed.

The discussion in paragraphs 5-9 and 5-12 conncerning stacking parameters and passing procedures as parameters applies to subroutines as well as procedures except that labels and subroutines cannot be passed as parameters to a subroutine.

5-14. RETURN STATEMENT

The RETURN statement is used to exit a procedure or subroutine at some place other than the last END of the body. Additionally, the RETURN statement can be used to leave some or all of the parameters on the stack after returning to the point of call.

The form of the RETURN statement is:

RETURN [*count*]

EXAMPLES:

```
RETURN;
RETURN 2;
```

where

count
indicates how many words to delete from the stack. The *count* is an unsigned decimal, based, composite, or equated integer constant.

A RETURN statement within a procedure generates an EXIT instruction, whereas a RETURN statement within a subroutine generates an SXIT instruction. Multiple RETURN statements within a single procedure or subroutine are allowed. You can also use a RETURN statement in the main-body of a program to terminate the program.

If a *count* is not specified, all parameters are deleted from the stack. If the count equals n, then only the top n words are deleted. If the count equals 0, all parameters are left on the stack. Note that count is a word count and not a parameter count. You can specify a count greater than the number of words passed as parameters; however, you should be very careful that you only delete values you want to delete.

The calling program must know how many parameters will be left on the stack upon returning because it must take care of them (examine, save, or delete them). INTEGER, LOGICAL, and BYTE values use one word; DOUBLE and REAL values use two words; labels use three words; and LONG values use four words. Call-by-reference parameters (except labels) use one word.

MACHINE LEVEL CONSTRUCTS

SECTION VI

6-1. ASSEMBLE STATEMENT

The ASSEMBLE statement is used to generate code by specifying the mnemonics for the hardware instructions. Instructions within an ASSEMBLE statement can be labeled, and control can be transferred to these labeled instructions from outside the ASSEMBLE statement. Additionally, identifiers which are outside the ASSEMBLE statement can be referenced within the statement, but any indirect references or indexing must be explicitly specified.

The form of an ASSEMBLE statement is:

ASSEMBLE ([*label*:] *instruction* [;...; [*label*:] *instruction*])

EXAMPLES:

```
ASSEMBLE (LOAD A;
          L1: DUP,ZERO;
          STOR C;
          STOR D);
ASSEMBLE (LOAD P+0;ZERO;STD A);
```

where

label
identifies the instruction. The label is an SPL identifier.

instruction
indicates a machine instruction to be executed or a pseudo-op to generate a constant. The instruction conforms to one of the ten formats shown in figure 6-1.

The following conventions are used in the instruction formats:	
I	Indirection
X	Index Register or Indexing
label id	A statement or instruction label within addressing range.
variable id	A data item identifier within addressing range.
usi	An unsigned integer less than or equal to the integer specified. For example usi255 means an unsigned integer between 0 and 255 inclusive.

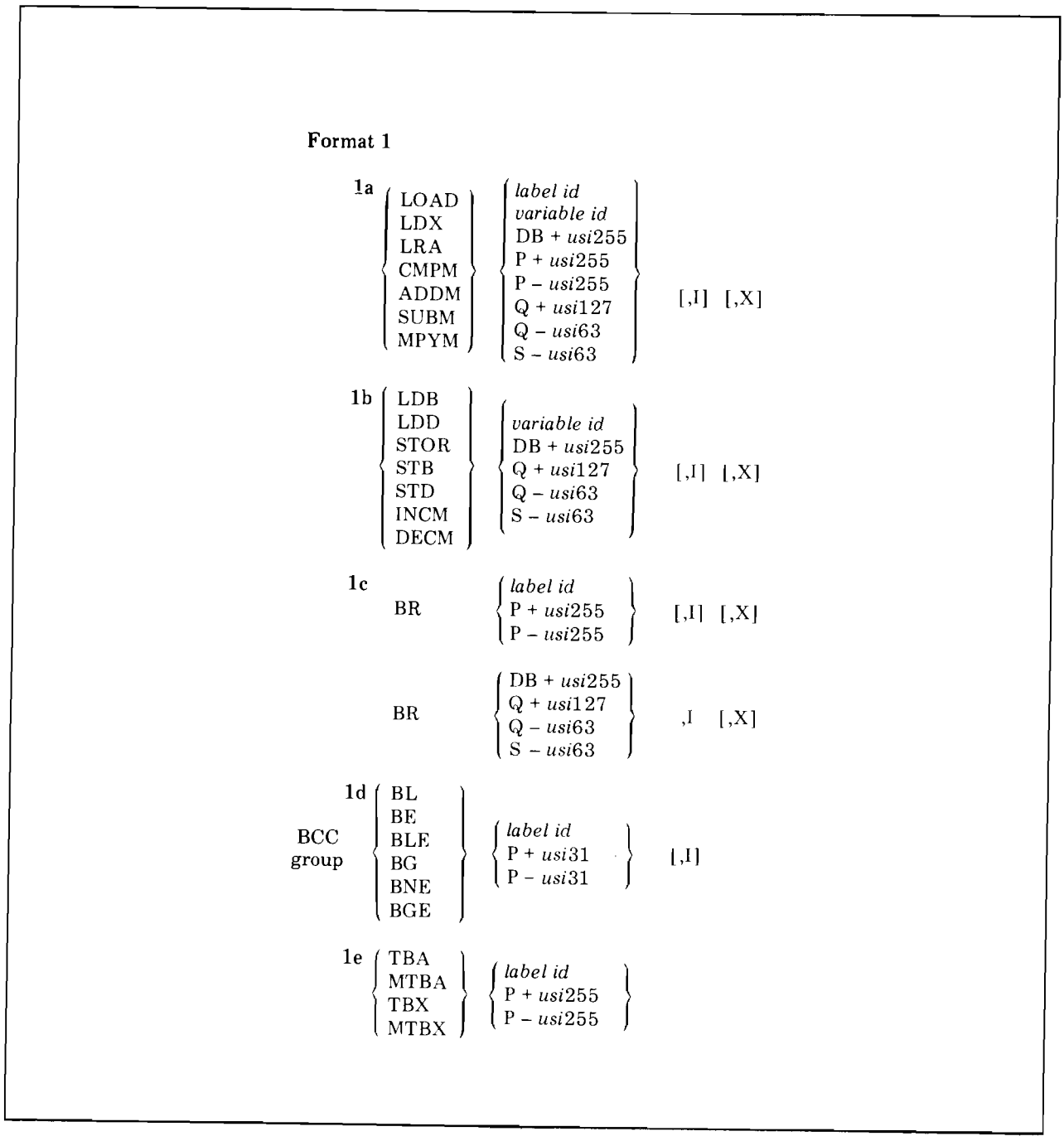


Figure 6-1. Instruction Formats

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 { IABZ, IXBZ, DXBZ, BCY, BNCY, CPRB, DABZ, BOV, BNOV, BRO, BRE } { *label*, P ± *usi*31, * ± *usi*31 } [,I]

Figure 6-1. Instruction Formats (Continued)

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 { ASL, ASR, LSL, LSR, CSL, CSR, SCAN, TASL, TASR, TNSL, DASL, DASR, DLSL, DLSR, DCSL, DCSR, TBC, TRBC, TSBC, TCBC, QASL, QASR } *usi*63 [,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 { LDI, LDXI, CMPI, ADDI, SUBI, MPYI, DIVI, PSHR †, LDNI, LDXN, CMPN, SETR† } *usi*255

† = a privileged instruction for some registers

4b { EXF, DPF } *usi*15 : *usi*15

For example,

```
ASSEMBLE (LDI 255; ADDI 5; EXF 7:9);
```

Figure 6-1. Instruction Formats (Continued)

Format 5

RSW
LLSH†
PLDA†
PSTA†
LSEA†
SSEA†
LDEA†
SDEA†
IXIT†
LOCK†
PCN†
UNLK†

† = a privileged instruction

For example,

```
ASSEMBLE (RSW; PLDA; . . . LLSH; . . . PSTA);
```

Format 6

PAUS SED XEQ SIO RIO WIO TIO CIO CMD SIN HALT LST SST	*usi*15
XCHD SMSK RMSK PSDB DISP PSEB SCLK RCLK	*miniop*-5

For example,

```
ASSEMBLE (XEQ 4);
```

All of these instructions except XEQ and RMSK are privileged.

Figure 6-1. Instruction Formats (Continued)

Format 7

{ PCAL
SCAL
EXIT
SXIT
ADXI
SBXI
LLBL
LDPP
LDPN
ADDS
SUBS
ORI
XORI
ANDI } *usi*255

PCAL *procedure identifier*
SCAL (user must load label onto stack)
LLBL *procedure identifier*

For example,

ASSEMBLE (PCAL READ;....SCAL 0;...ORI %377);

Format 8

8a { MOVE
MVB
CMPB } [PB] [,0
,1
,2
,3]

If item two is empty, a DB relative move is assumed.
If item three is empty, the stack decrement is 3.

8b MVBW { A
N
AN
AS
ANS } [,0
,1
,2]

If item three is empty, the stack decrement is 2.

*8c { MVBL†
MVLB†
SCW
SCU } [,0
,1
,2
,3]

†Privileged instruction.

Figure 6-1. Instruction Formats (Continued)

If item two is missing, the stack decrement is 3. For example,

```
ASSEMBLE (SCW, 1);
ASSEMBLE (MVBW AN, 0);
ASSEMBLE (CMPB PB, 1);
```

*8d { MABS† / MTDS† / MDS† / MFDS† } [0 / 1 / 2 / 3 / 4 / 5 for MABS and MDS]

*If there is no stack-decrement, the default is equal to the number of parameters.

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 { CVAD / CVBD } [0 / 1]

Figure 6-1. Instruction Formats (Continued)

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
```
CVDB [ 0 ]
     [ 1 ]
```

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
```
{ ADDD }
{ SUBD }
{ MPYD } [ 0 ]
{ CMPD } [ 1 ]
{ SLD  } [ 2 ]
{ NSLD }
{ SRD  }
```

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
```
CVDA [ 0                    ]
     [ 1                    ]
     [ ABS  [ , { 0 } ] ]
     [ NABS [   { 1 } ] ]
```

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.

Figure 6-1. Instruction Formats (Continued)

A list of the mnemonics with their meanings is shown in table 6-1. For a complete description of the instructions, refer to the *Machine Instruction Set Reference Manual.*

Table 6-1. Machine Instruction Mnemonics

ALPHABETIC 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
BE	Branch on equals	3a
BG	Branch on greater than	
BGE	Branch on greater than or equal	
BL	Branch on less than	
BLE	Branch on less than or equal	
BNE	Branch on not equal	
BCY	Branch on carry	
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
DCSR	Double circular shift right	3b

BCC = BE, BG, BGE, BL, BLE, BNE

Table 6-1. Machine Instruction Mnemonics (Continued)

ALPHABETIC LISTING OF INSTRUCTIONS

MNEMONIC	FUNCTION	FORMAT
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
DUMP	Load soft dump program	
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
HIOP	Halt I/O program	
IABZ	Increment A, branch if zero	3a
INCA	Increment A	2
INCB	Increment B	2
INCM	Increment memory	1b
INCX	Increment index register	2
INIT	Initialize I/O channel	
IXBZ	Increment X, branch if zero	3a
IXIT	Interrupt exit	5

Table 6-1. Machine Instruction Mnemonics (Continued)

ALPHABETIC LISTING OF INSTRUCTIONS

MNEMONIC	FUNCTION	FORMAT
LADD	Logical add	2
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
MCS	Memory controller read status	
MDS	Move using data segment	8
MFDS	Move from data segment	8
MOVE	Move words	8a
MPY	Multiply	2
MPYD	Decimal Multiply	10d
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

Table 6-1. Machine Instruction Mnemonics (Continued)

ALPHABETIC LISTING OF INSTRUCTIONS

MNEMONIC	FUNCTION	FORMAT
PSEB	Pseudo interrupt enable	6
PSHR	Push registers	4a
PSTA	Privileged store into absolute address	5
QASL	Quadruple arithmetic shift left	3b
QASR	Quadruple arithmetic shift right	3b
RCCR	Read system clock	
RCLK	Read clock	6
RIO	Read I/O	6
RIOA	Read I/O adapter	
RIOC	Read I/O channel	
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
SCLR	Set system clock limit	
SCU	Scan until	8c
SCW	Scan while	8c
SDEA	Store double word into extended address	5
SED	Set enable/disable external interrupts	6
SEML	Semaphore load	
SETR	Set registers	4a
SIN	Set interrupt	6
SINC	Set system clock interrupt	
SIO	Start I/O	6
SIOP	Start I/O channel program	
SIRF	Set internal interrupt reference flag	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
STRT	Programmatic warm start	
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
TOFF	Hardware timer off	
TON	Hardware timer on	

Table 6-1. Machine Instruction Mnemonics (Continued)

ALPHABETIC LISTING OF INSTRUCTIONS

MNEMONIC	FUNCTION	FORMAT
TRBC	Test and reset bit, set condition code	3b
TSBC	Test, set bit, set condition code	3b
TSBM	Test and set bit in memory	3b
UNLK	Unlock resource	5
WIO	Write I/O	6
WIOA	Write I/O adapter	
WIOC	Write I/O channel	
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
ZERO	Push zero	2
ZROB	Zero B	2
ZROX	Zero X	2

6-2. DELETE STATEMENT

The delete statement allows you to delete words from the stack without using the ASSEMBLE statement.

The form of the delete statement is one of the following:

1. **DEL**
2. **DELB**
3. **DDEL**

The mnemonics have the same meanings as in the ASSEMBLE statement:

DEL Delete the top of stack (S– 0) and decrement the S-register by 1.

DELB Delete the contents of S– 1 by storing S– 0 into it and decrement the S-register by 1.

DDEL Delete the contents of S– 0 and S– 1 and decrement the S-register by 2.

See figure 6-2 for the effect of the delete statement on the stack.

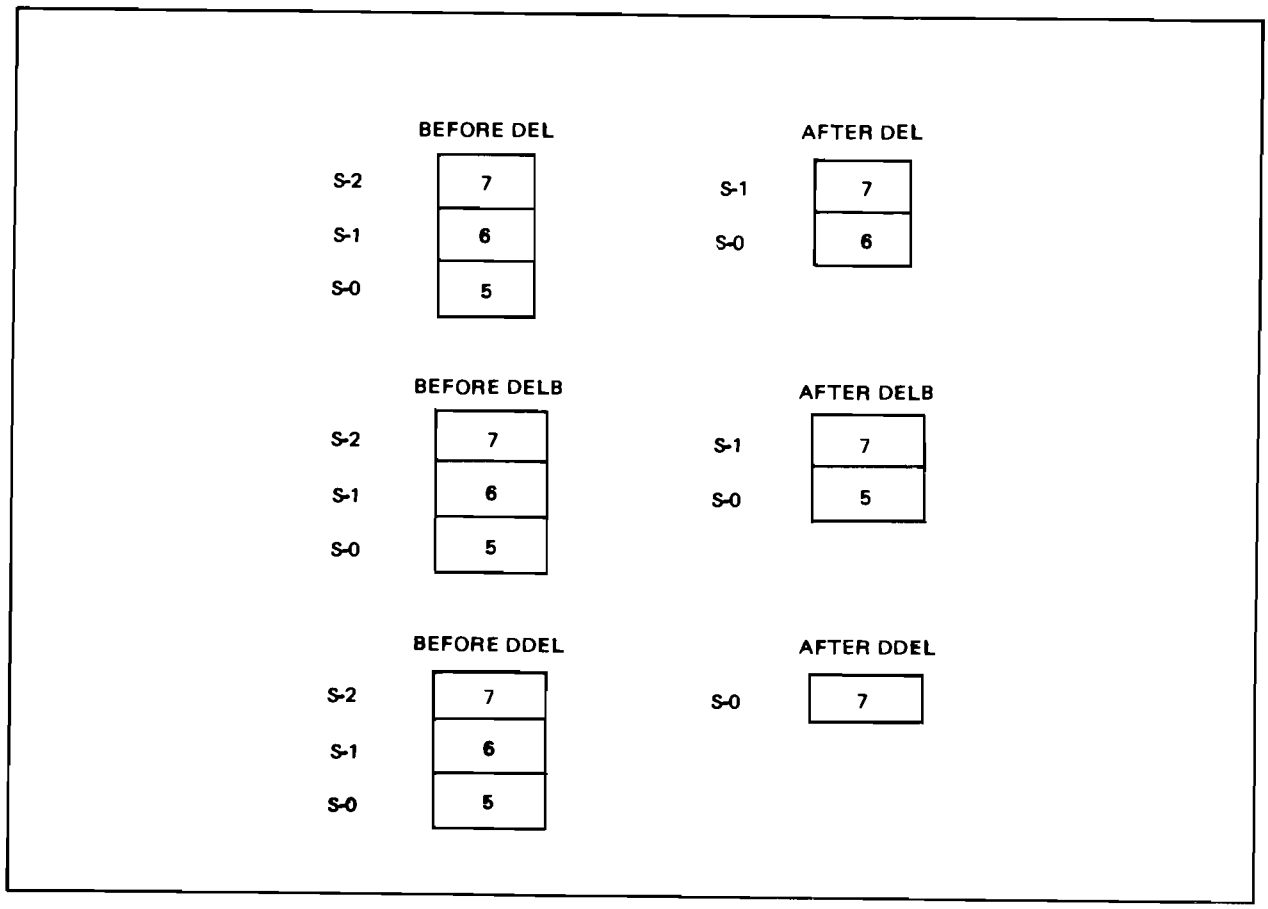


Figure 6-2. Delete Statement

6-3. PUSH STATEMENT

The PUSH statement puts the contents of any or all of the registers onto the stack using the PSHR instruction.

The form of the PUSH statement is:

PUSH (*register* [,...,*register*])

EXAMPLES:

```
PUSH (X,Q,STATUS);
PUSH (DL);
```

where

register
is one of the following hardware registers: S,Q,X,STATUS,Z,DL, DB, or SBANK.

If more than one register is specified, they are stacked in the order shown below (regardless of the order in which they are listed in the PUSH statement):

REGISTER	VALUE STACKED
S	S– DB (relative S before PSHR instruction)
Q	Q– DB (relative Q)
X	Index Register
STATUS	Status Register
Z	Z– DB (relative Z)
DL	DL– DB (relative DL)
DB	DB (absolute address — 2 words)
SBANK	Stack Bank

Thus, if you use the statement:

```
PUSH(STATUS,X,DL);
```

The stack would look like:

S-2	Index Register
S-1	Status Register
S-0	Relative DL

Privileged mode is required to push either DB or SBANK.

6-4. SET STATEMENT

The SET statement is used to set the contents of any or all registers using values taken from the stack. The SETR instruction is used to perform this operation:

The form of the SET statement is:

SET (*register* [,...,*register*])

EXAMPLES:

```
SET(S);
SET(Q,S);
```

where

register
is one of the following hardware registers: S,Q,X,STATUS,Z,DL,DB, or SBANK.

Privileged mode is required to set SBANK, DB, DL, Z, and parts of the Status register. If you are not in privileged mode and you set the STATUS register, only the Traps Enabled bit, the Carry and Overflow bits, and the Condition Code are set. The rest of the STATUS register is not altered.

Before using a SET statement, the appropriate values must be loaded onto the stack. If more than one register is specified, they are taken from the stack in the following order (regardless of the order in which they are listed in the SET statement):

REGISTER	VALUE TAKEN FROM THE STACK
SBANK	Stack Bank
DB	DB (absolute address — 2 words)
DL	DL– DB (relative DL)
Z	Z– DB (relative Z)
STATUS	Status Register
X	Index Register
Q	Q– DB (relative Q)
S	S– DB (relative S)

Relative addresses in the stack are added to the absolute value of DB before setting the registers. The values are deleted from the stack by the SETR instruction.

Note that the order in which the registers are set is the reverse of the order in which they are pushed. This reversal is consistent with the last-in, first-out stack architecture of the HP 3000.

6-5. WITH STATEMENT

The WITH statement is intended specifically for privileged users running in split-stack mode (see the final paragraph of section 8-1). It performs a syntactic check to ensure that only split-stack compatible code is generated. Reliability is increased by limiting the code inside a WITH statement to certain DB-relative offsets. The variables used inside the WITH block must have been declared inside a corresponding DATASEG declaration, or be Q- or S-relative, unless a move is also included. The only form of move allowed inside the WITH statement is the MOVEX between data segments (see Section 4-21A), where the variables used may have been declared in any DATASEG declaration. Checking will be performed as for OPTION SPLIT (see Section 7-13A).

The form of the WITH statement is:

```
WITH dataseg-name DO

  BEGIN
    .
    .
    .
  END;
```

where

dataseg-name
is an SPL identifier.

The actual switching of data segments is left up to the SPL/3000 programmer.

PROCEDURES, INTRINSICS, AND SUBROUTINES

SECTION VII

7-1. SUBPROGRAM UNITS

There are three types of subprogram units in SPL: procedures, intrinsics, and subroutines. Procedures and intrinsics are identical except for their location and how they are declared in a program. Subroutines are less powerful than procedures and intrinsics and use different hardware instructions to call and exit. The declarations for procedures and intrinsics follow the global data declarations and precede any global subroutine declarations as shown below.

```
      BEGIN
        [global-data-declarations]
----> [procedures/intrinsics] <-----
        [global-subroutines]
        [main-body]
      END.
```

Local subroutine declarations are within the procedure body following the other local declarations in the procedure declaration and preceding the executable statements of the procedure body.

7-2. PROCEDURE DECLARATION

A procedure declaration defines an identifier as a procedure and specifies what attributes the procedure will have:

- Data type of result for function procedures.
- Type and number of formal parameters.
- Options (external body, variable number of parameters,etc.).
- Local variables.
- Statements of the procedure body.

Procedures are called by means of the identifier and a list of actual parameters. Procedure declarations are not allowed within other procedures unless they are declared without a body (that is, OPTION EXTERNAL).

The form of a procedure-declaration is:

[*type*] PROCEDURE *procedure-name*
[(*formal-parm*[,...,*formal-parm*]); [*value-part*] *specification-part*]
[*option-part*;]
[*procedure-body*;]

where

type
indicates that the procedure is a function procedure which returns a value of the specified data type. The *type* is INTEGER, LOGICAL, BYTE, DOUBLE, REAL, or LONG.

procedure-name
is an SPL identifier used to identify the procedure.

formal-parm
is an SPL *identifier* which is used as a local *identifier* to reference an actual parameter.

value-part
indicates which formal parameters are to be passed by-value. All parameters which are not specified in the *value-part* are passed by-reference. The *value-part* is of the form: VALUE *formal-parm* [,...,*formal-parm*];

specification-part
indicates the characteristics of each formal parameter. The *specification-part* is of the form: *specification* [;...;*specification*];

specification
is one of the following:

type formal-parm [,...,*formal-parm*]
[*type*] ARRAY *formal-parm* [,...,*formal-parm*]
LABEL *formal-parm* [,...,*formal-parm*]
[*type*] POINTER *formal-parm* [,...,*formal-parm*]
[*type*] PROCEDURE *formal-parm* [,...,*formal-parm*]

option-part
specifies which options are to be in effect. The *option-part* is of the form: OPTION *option* [,...,*option*]

option
is UNCALLABLE, PRIVILEGED, EXTERNAL, CHECK *level,* VARIABLE, FORWARD, INTERRUPT, or INTERNAL. Each *option* is described fully below, starting with paragraph 7-5.

level
is an unsigned decimal, based, composite, or equated integer constant between 0 and 3 inclusive.

procedure body
is one of the following:

1. *statement*
2. BEGIN
   [*local-data-declarations*]
   [*external-procedure/intrinsic-declarations*]
   [*local-subroutine-declarations*]
   statement [;...; statement]
   END

statement
is any executable SPL statement (see Sections IV through VI).

local-data-declarations
include any or all of the following (intermixed in any order):

define declaration(s)
equate declaration(s)
local simple variable declaration(s)
local array declaration(s)
local pointer declaration(s)
label declaration(s)
switch declaration(s)
entry declaration(s)

external-procedure/intrinsic-declarations
are intrinsic declarations and procedure declarations for external procedures, intermixed in any order.

local-subroutine-declarations
are local subroutine declarations (described fully later in this section).

A procedure is a self-contained section of code which is called to perform a function. Procedures are hardware-dependent in SPL — they are called using the PCAL instruction and return using the EXIT instruction; the PRIVILEGED and UNCALLABLE options are hardware-defined and checked; and local variables can be allocated relative to the Q-register since it is set to a fresh area of the stack by the PCAL instruction. Because of the hardware capability provided for procedures, they can be called recursively (that is, a procedure can call itself). For the syntax and semantics of calling procedures, see "Function Designator" in paragraph 4-6 and "Procedure Call Statement" in paragraph 5-8. Multiple entry points for procedures are covered under "Entry Declaration" in paragraph 7-30.

7-3. DATA TYPE

If a data type is specified for a procedure, that procedure is a function and can be called within expressions. It returns a value of the type specified by assigning the value to its name somewhere within the procedure body in an assignment statement. For details on calling functions, see "Function Designator" in paragraph 4-6.

If a data type is not specified, the procedure does not return a value and cannot be called as a function.

7-4. PARAMETERS

The formal parameters (if any) of a procedure must be fully specified as to type and whether each is call-by-value or call-by- reference. The formal parameters can then be used within the procedure body as if they were locally declared identifiers. When the procedure is called, an actual parameter is supplied for each dummy (formal) parameter. Up to 31 formal parameters can be specified for each procedure.

Simple variables, arrays, labels, pointers, and procedures can be passed as parameters. Simple variables and pointers can be passed by value or by reference; procedures, labels, and arrays are passed by reference only.

The VALUE list specifies which parameters are to be passed by value; parameters not listed in the VALUE list are passed by reference. When a parameter is called by value, the value of the actual parameter is specified by an expression and is loaded onto the stack. Value parameters are handled exactly as local variables from that point on; any changes to them are limited to the scope of the procedure. For reference parameters, the address of the parameter is loaded onto the stack instead of a value; changes to reference parameters can change the value of the actual parameter outside the procedure.

The VARIABLE option allows a variable number of parameters to be passed (see "Options," paragraph 7-7).

Actual parameters (when the procedure is called) can be constants, expressions, simple variables, array references, pointer references, procedure identifiers, label identifiers, or stacked values (* in place of a parameter indicates that the parameter value or address has been loaded onto the stack by the user; see "Procedure Call Statement" in paragraph 5-8 for details).

If the formal parameter is a simple variable, it is passed the address (call-by-reference) or actual value (call-by-value) of a data item. If the formal parameter is an array, it is passed the address of the zero element. Thus, all arrays, even direct arrays, are effectively passed as indirect arrays. If the formal parameter is a pointer, it is passed the addresss (call-by-reference) or contents (call-by-value) of the pointer. Parameters are stored in Q-3-n to Q-4 where n is the number of words required for parameter storage (maximum 60). Call-by-reference parameters, except labels, use one word. INTEGER, LOGICAL, and BYTE values also use one word; DOUBLE and REAL values use two words; labels use three words; and LONG values use four words.

Table 7-1 shows what actual parameters can be passed to what formal parameters (a blank space indicates an error condition):

NOTE

If the *actual-parameter* is a byte array and the *formal-parameter* is an array with a different data type, the byte address is converted to a word address by arithmetically right shifting the byte address by one bit. Thus, the maximum byte address is DB+32767 (which equals DB+16383 words). Additionally, the array in the procedure begins on a word boundary regardless of whether or not the starting byte of the *actual-parameter* starts on a word boundary.

Table 7-1. Parameters Passed to Formal Parameters

Actual Parameter	Formal Parameter						
	Simple Variables By Reference	Simple Variables By Value	Arrays	Pointer By Reference	Pointer By Value	Procedures	Labels
Constant	Warning (uses 1 word as address)	Must be same word size.	Warning (uses 1 word as address)	Warning (uses 1 word as address)	Warning (uses 1 word as address)		
Expression		Must be same word size.					
Simple Variable Identifier	OK	Must be same word size.	OK, loads address of simple variable		OK, load address of simple variable		
Array Reference	OK	Must be same word size.	OK		OK		
Pointer Reference	OK	Must be same word size.	OK	OK	OK		
Procedure Identifier						OK	
Label Identifier							OK
* (stacked)	OK	OK	OK	OK	OK	OK	

7-5. OPTIONS

The option part of a procedure declaration consists of the reserved word OPTION followed by a list of option words separated by commas and terminated by a semi-colon. The meaning of the various options are discussed in the following paragraphs.

7-6. OPTION UNCALLABLE. This option causes the "uncallable" bit to be turned on in the Segment Transfer Table entry for the procedure. The uncallable bit is examined by the PCAL instructions to restrict access to procedures that specify this option. Uncallable procedures can only be called by code executing in privileged mode. If this option is not specified, the procedure is callable.

7-7. OPTION PRIVILEGED. This option causes the procedure to be run in privileged mode, assuming that the person running the program is allowed to execute in privileged mode by the operating system. If this option is not specified, the procedure runs in user mode.

7-8. OPTION EXTERNAL. This option specifies that the procedure body (or code) exists external to the program being compiled. The procedure body is not included in the declaration and is linked to the main program later by the operating system. If you need to refer to a procedure compiled separately, you must include an OPTION EXTERNAL declaration for the procedure which indicates to the compiler the type and number of parameters. Intrinsics are the only procedures not requiring a procedure declaration (see "Intrinsic Declaration" in paragraph 7-34). When procedures are compiled separately (to be called later as option EXTERNAL), they can use the EXTERNAL-GLOBAL mechanism to establish data linkages.

7-9. OPTION CHECK. This option is provided for option external procedure declarations which will subsequently be called as externals by other programs. The option specifies how much checking is done by the operating system between the option external declaration in the calling program and the actual procedure declaration as compiled. At PREP time, errors from RL and USL procedures are detected. At RUN time, errors from SL procedures are detected.

If this option is not specified, no checking is performed. Otherwise, the smaller of the two levels, the level specified in the calling program and the level specified in the external procedure, is used to determine the level of checking. Intrinsics determine their level of checking, never the caller. The check values are:

0 — no checking

1 — check procedure type only.

2 — check procedure type and number of parameters.

3 — check procedure type, number of parameters and type of each parameter.

7-10. OPTION VARIABLE. This option specifies that the procedure can be called with a variable number of actual parameters. 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. The correspondence is from right to left with the rightmost bit (bit 15) corresponding to the right parameter. In the procedure call, the occurrence of a right parentheses 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− address, but missing parameters have garbage in their locations.

7-11. OPTION FORWARD. This option specifies that the complete procedure declaration will be introduced later in the program. FORWARD is used to circumvent contradictions incurred by recursion when a procedure calls itself indirectly. Procedures must be declared before being referenced.

7-12. OPTION INTERRUPT. This option specifies that the procedure is an external interrupt procedure. The structure and uses of interrupt routines are covered in the HP 3000 Multiprogramming Executive Operating System (MPE) manuals.

7-13. OPTION INTERNAL. A procedure with this option cannot be called from another segment. This makes processing of the procedure more efficient for the loader subsystem and allows more than one segment to have a procedure with the same name. INTERNAL procedures cannot be moved to another segment or called from a procedure in another segment. This option applies to code segments that are put into the SL only. See the *MPE Segmenter Reference Manual,* Section 3.

7-13A. OPTION SPLIT. This option is intended specifically for privileged users running in split-stack mode to improve the reliability of the generated split-stack code (see section 8-1). When a procedure specifies this option, generation of the following instructions or declarations will result in an error.

- Local indirect (DB-relative) arrays
- OWN variables
- Q-relative LRA's (generated when assigning to a pointer the address of an indexed element of a local array)

7-14. LOCAL DECLARATIONS

Procedures can declare local variables that are known only within the procedure and are normally allocated space in the Q+ area when the procedure is called. Thus, they occupy space only when the procedure is called and are deleted when the procedure exits. As indicated in the syntax, all declaration types are allowed within procedures with these comments:

- Procedures declared within procedures must be OPTION EXTERNAL.
- Data declarations (simple variables, arrays, pointers) must be of the "local" form (see the appropriate paragraphs in this section).

There are 127 words available for storage of local variables for each procedure. All simple variables, pointers, direct arrays, and pointers to indirect arrays, must fit in 127 words. Indirect arrays can extend past this range as long as the pointer to the zero element is within range.

7-15. OWN VARIABLES. OWN variables are a special variety of local variable; they are allocated space in the DB area rather than on the top of the stack. If initialization is specified, they are initialized at the beginning of the program, not every time the procedure is called. Since they are allocated in the global area, they are not deleted when a procedure exists, but are still in existence, with their last value, when the procedure is called again. However, they are directly accessible only by the procedure in which they are declared. OWN variables can be simple variables, pointers, or arrays.

7-16. LOCAL SIMPLE VARIABLE DECLARATIONS

A simple variable declaration specifies the data type, addressing mode, storage allocation, and initialization value for identifiers to be used as single data items. The data type assigned to a variable determines the amount of space allocated to the variable and the set of machine instructions which can operate on the variable.

There are three types of local simple variable declarations: standard, OWN, and EXTERNAL. Standard simple variable declarations can allocate Q-relative storage each time the procedure is called or can specify the use of a location relative to a base register or another variable. OWN variable declarations allocate DB-relative storage at the beginning of the program. EXTERNAL variable declarations link global variables in a separately compiled main program to variables in a procedure; the global variables must be declared with the GLOBAL attribute.

There are two methods which can be used to link global variables to variables in separately compiled procedures. The first method is to use the GLOBAL attribute in the global variable declaration (see paragraph 3-2) and the EXTERNAL attribute in the local variable declaration. The identifiers in both declarations must be the same and the Segmenter is responsible for making the correct linkages. The second method is to include dummy global declarations at the beginning of subprogram compilations. All global declarations must be included, even for identifiers not referenced in the subprogram, and they must be in the same order as in the main program. It is possible, although not recommended, to use different identifiers for the same variable, but you are responsible for keeping them straight. The second method is faster and requires less space in the USL (User Subprogram Library) files, but does not protect you against improper linkages.

7-17. STANDARD LOCAL VARIABLES. A standard local variable declaration specifies identifier(s) which can either be allocated storage each time the procedure is called or which use locations relative to base registers or other identifiers. Local variables cannot be referenced outside the procedure in which they are declared.

The form of a standard local simple variable declaration is:

***type variable-declaration*[,...,*variable-declaration*];**

EXAMPLES:

```
INTEGER I,J:=1245;
DOUBLE II:=-1234579 D;
REAL A,B,C:=1.321E-21,Z=DB+3;
LOGICAL INDX=X,LI=I,JI=J;
BYTE DOLLAR:="$";
```

where

type
specifies the data type of the variables in the declaration. The *type* may be INTEGER, LOGICAL, BYTE, DOUBLE, REAL, or LONG.

variable-declaration
is one of the following forms:

variable [:= *initial-value*]
variable = *register* [*sign offset*]
variable = *reference-identifier* [*sign offset*]

variable
is a legal SPL *identifier*.

reference-identifier
is any legal SPL *identifier* which has been declared as a data item except DB,PB,Q,S, or X.

initial-value
is an SPL constant to be used as the value of the *variable* when the procedure is called.

register
specifies the register to be used in a register reference. The register may be DB, Q, S, or X.

sign
is + or −.

offset
is an unsigned decimal, based, composite, or equated integer constant.

Form 1 of the variable declaration allocates the next available Q-relative location(s) for the *variable*. The amount of space allocated depends on the variable type. If an initial value is specified, the *variable* is initialized when the procedure is called. If the constant used for the initial value is too large, it is truncated on the left except string constants which are truncated on the right. If no initial value is specified, the *variable* is not initialized.

Form 2 of the variable declaration equivalences a *variable* either to the index register (X) or to a location relative to the contents of one of the base registers (DB, Q, or S). Since the index register is 16 bits, only variables of type INTEGER, LOGICAL, and BYTE may be equivalenced to the Index register (X).

Form 3 of the variable declaration equivalences a variable to a location relative to another *variable*. The *reference-identifier* must be declared first. For example, the declarations

```
LOGICAL A;
INTEGER B= A+ 5;
```

equivalence B to the location 5 cells past the location of A. Simple variables which are address referenced to arrays use either the location of the zero element of the array (if direct) or the location of the pointer to the zero element of the array (if indirect). Note that if the *reference-identifier* is an array, only the zero element may be used in a variable reference of a simple variable declaration. In any case, the final address must be within the direct address range.

DB, PB, Q, S, and X cannot be used as the *identifier* on the right side of an equals sign in a variable declaration, because they are interpreted as register references instead of variable references. For example, consider the declaration

```
INTEGER A,B,C,DB,D= DB+ 2;
```

The variable D is equivalenced to the location 2 cells past the cell to which the DB register points — not 2 cells past the location assigned to the variable DB.

The legal combinations of registers, signs, and offsets are shown below

Register	Sign	Offset
DB	+	0 to 255
Q	+	0 to 127
Q	−	0 to 63
S	−	0 to 63
X	none	none

7-18. OWN SIMPLE VARIABLES. OWN simple variables are allocated space in the DB-relative area instead of the Q-relative area. Thus, an OWN variable retains its value from one execution of the procedure to the next. However, the variable can only be referenced within the procedure in which it is declared. If an OWN variable is initialized, it is initialized only at the start of the program instead of each time the procedure is called.

The form of an OWN simple variable declaration is:

OWN ***type variable***[**:=** ***init-value***] [**,...,**]***variable***[**:=** ***init-value***]];

EXAMPLES:

```
OWN INTEGER I:=1,J,K:=10;
OWN REAL R1;
OWN BYTE CHAR:=" ";
```

where

type
specifies the data type of the variables in the declaration. The type may be INTEGER, LOGICAL, BYTE, DOUBLE, REAL, or LONG.

variable
is a legal SPL identifier.

initial-value
is an SPL constant to be used as the value of the variable when the procedure is called.

7-19. EXTERNAL SIMPLE VARIABLES. An EXTERNAL simple variable declaration is used to link global variables for referencing in procedures compiled separately from the main program. The identifiers must be the same used in the global declaration and the GLOBAL attribute must have been specified.

The form of an EXTERNAL simple variable declaration is:

EXTERNAL ***type variable*** [**,...,**]***variable***];

EXAMPLES:

```
EXTERNAL INTEGER I,J,K;
EXTERNAL REAL R;
```

where

type
specifies the data type of the variables in the declaration. The *type* may be INTEGER, LOGICAL, BYTE, DOUBLE, REAL, or LONG.

variable
is a legal SPL *identifier*.

7-20. LOCAL ARRAY DECLARATIONS

An array declaration specifies one or more identifiers to represent arrays of subscripted variables. An array is a block of contiguous storage which is treated as an ordered sequence of "variables" having the same data type. Each "variable" or element of the array is denoted by a unique subscript; note that SPL provides one-dimensional arrays only. An array declaration defines the following attributes of an array:

- The bounds specification (if any) which determines the size of the array and the legitimate range of indexing.
- The data type of the array elements.
- The storage allocation method.
- The initial values, if desired. Note that arrays local to a procedure cannot be initialized unless they are PB-relative.
- The access mode (direct or indirect).

There are two types of access modes used for arrays: indirect and direct. An indirect array uses a pointer to the zero element of the array. Addressing an indirect array element uses both indirect addressing and indexing. If the array is a BYTE array, the pointer contains a DB-relative byte address. For all other data types, the pointer contains a DB-relative word address. A direct array uses a location within the direct address range of one of the registers (DB, Q, or S) as the zero element of the array and then uses indexing to address a specific array element.

There are three types of local array declarations: standard, OWN, and EXTERNAL. A standard local array declaration can allocate Q-relative storage each time the procedure is called, PB-relative storage, or can specify the use of a location relative to a base register or another data item. OWN array declarations allocate DB-relative storage at the beginning of the program. EXTERNAL array declarations link global arrays in a separately compiled main program to arrays in a procedure. The global arrays must be declared with the GLOBAL attribute.

There are two methods which can be used to link global arrays to arrays in separately compiled procedures. The first method is to use the GLOBAL attribute in the global array declaration (see paragraph 3-3) and the EXTERNAL attribute in the local array declaration. The identifiers in both declarations must be the same and the Segmenter is responsible for making the correct linkages. The second method is to include dummy global declarations at the beginning of subprogram compilations. All global declarations must be included, even for identifiers which are not referenced in the subprogram, and they must be in the same order as in the main program. It is possible, although not recommended, to use different identifiers for the same array, but you are responsible for keeping them straight. The second method is faster and requires less space in the USL (User Subprogram Library) files, but does not protect you against improper linkages.

7-21. STANDARD LOCAL ARRAYS. A standard local array declaration specifies identifier(s) which can be allocated storage each time the procedure is called, stored in the code segment, or which use locations relative to base registers or other data items. Local arrays cannot be referenced outside the procedure in which they are declared.

The form of a standard local array declaration is:

[*type*] **ARRAY** [*local-array-dec,...,local-array-dec,*] { *local-array-dec* / *constant-array-dec* } ;

where

type
specifies the data type of the array. The *type* can be INTEGER, LOGICAL, BYTE, DOUBLE, REAL, or LONG. If not specified, the array is type LOGICAL.

local-array-dec
is one of the following forms:

1. *array-name(lower:upper)* [=Q]

   This form is used for an uninitialized array with defined bounds. If =Q is not specified, the array is indirect and the next available Q-relative location is allocated for the pointer to the zero element of the array. If =Q is specified, the array is direct and the next available n cells in the Q+ area are allocated for the array, where n is the number of locations required to store the array. The zero element of the array must be within the direct address range whether or not it is actually an element of the array. For example, consider the declaration:

   INTEGER ARRAY A(−20:−10)=Q;

   The next available Q-relative location is allocated to A(−20), but all indexing is done relative to A(0) even though it is not an actual element of the array. The address which A(0) would have if it were in the array must be between Q−63 and Q+127.

2. *array-name(variable-lower:variable-upper)*

   This form is used for an indirect array with variable bounds. The bounds are evaluated each time the procedure is called and storage is allocated accordingly at execution time. The array cannot be initialized.

3. *array-name*(@)=Q

   This form is used for an indirect array with undefined bounds. The next available Q-relative location is used, without being allocated, as the pointer to the zero element of the array. Space is not allocated for the array nor is initialization allowed.

4. *array-name*(*)=Q

   This form is used for a direct array with undefined bounds. The next available Q-relative location is used, without being allocated, as the zero element of the array. Space is not allocated for the array nor is initialization allowed.

5. *array-name*(@) [=*register sign offset*]

   This form is used for an indirect array with undefined bounds whose pointer is DB, Q, or

S-relative. If a base-register-reference is not specfied, the next available Q-relative cell is allocated for the pointer to the zero element of the array. If a base-register reference is specified, then that DB-, Q-, or S-relative cell is used, without being allocated, as the pointer to the zero element of the array. Space is not allocated for the array nor is initialization allowed.

6. *array-name*(*)

   This form can be used for an indirect array with undefined bounds. The next available Q-relative cell is allocated for the pointer to the zero element of the array. Space is not allocated for the array nor is initialization allowed. This form is equivalent to *array-name*(@) without a base-register reference.

7. *array-name*(*) = *register sign offset*

   This form is used for direct arrays with undefined bounds which are DB-, Q-, or S-relative. The specified cell is used as the zero element of the array; however, space for the array is not actually allocated and the array cannot be initialized.

8. *array-name*(*) = *reference-identifier* [*sign offset*]

   This form is used for equivalencing an array to a location relative to another identifier. The reference identifier may be a simple variable, a pointer variable, or another array and must be declared first. The array is a direct array except when the *reference-identifier* is an indirect array or a pointer variable and no *offset* is specified. If an *offset* is specified, the resulting address must be within the direct address range. For example, if A is at location Q+ 125, then the declaration

   INTEGER B(*)= A+ 10;

   would not be allowed because the direct address range for the Q register is – 63 to + 127. If the array is direct, the referenced location is used as the zero element of the array. If the array is indirect, the referenced location is used as the pointer to the zero element except when either the array or the *reference-identifier*, but not both is type BYTE, in which case the next available Q-relative cell is allocated for the pointer to the zero element. Space is not allocated for the array nor can the array be initialized. DB, PB, Q, S, and X cannot be used as the *reference-identifer* because they are interpreted as register references instead.

9. *array-name*(*) = *reference-identifier (index)*

   This form is used for equivalencing one array to another array. The *reference-identifier* may be either an array or a pointer variable and must be declared first. If the *reference-identifier* is a direct array, the array is a direct array whose zero element is the location of the referenced array element. If the *reference-identifier* is an indirect array or a pointer variable, the array is indirect. In this case, the next available Q-relative cell is allocated for the pointer to the zero element of the array when a non-zero index is specified or when either the array or the *reference-identifier* (but not both) is type BYTE; otherwise, both use the same location for the pointer to the zero element. In any case, space is not allocated for the equivalenced array nor can the equivalenced array be initialized. DB, PB, Q, S, and X cannot be used as the *reference-identifier* because they are interpreted as register references instead.

array-name
is a legal SPL *identifier*.

reference-identifier
is any legal SPL *identifier* which has been declared as a data item except DB,PB,Q,S, or X.

register
specifies the base register in a register reference. The *register* may be DB, Q, or S.

sign
is + or −.

offset
is an unsigned decimal, based, composite, or equated integer constant within the direct address range as shown below:

Register	Sign	Offset
DB	+	0 to 255
Q	+	0 to 127
Q	−	0 to 63
S	−	0 to 63

constant-array-dec
is of the form:

array-name(lower:upper) = PB := *value-group*[,...,*value-group*]

lower
specifies the lower bound of the array. It can be any decimal, based, composite, or equated single-word integer constant or constant expression.

upper
specifies the upper bound of the array. It can be any decimal, based, composite, or equated single-word integer constant or constant expression.

variable-lower
specifies the lower bound of a variable bounds array. The *variable-lower* is an INTEGER, LOGICAL, or BYTE simple variable.

variable-upper
specifies the upper bound of a variable bounds array. The *variable-upper* is an INTEGER, LOGICAL, or BYTE simple variable.

index
indicates the element of the referenced array to be used as the reference location. The *index* can be any decimal, based, composite, or equated single-word integer constant.

value-group
is either of the following:

1. *initial-value*
2. *repetition-factor* (*initial-value* [,...,*initial-value*])

initial-value
is any SPL numeric or string constant.

repetition-factor
specifies the number of times the initial value list will be used to initialize the array elements. The *repetition-factor* can be any unsigned non-zero decimal, based, composite, or equated single-word integer constant.

Local PB-arrays with defined bounds must be initialized. Initialization consists of a := followed by a list of numerical constants or strings. A group of constants can be surrounded by parentheses and preceded by a repetition factor (*n*) to specify that the constants in parentheses are to be used *n* times before going on to the next item in the list. These repeat groups cannot be nested. Elements are initialized starting with the lowest subscript and continuing up until the constant list is exhausted. The initialization list must not contain more values than there are elements in the array. If the constant used for the initial value is too large, it is truncated on the left except string constants which are truncated on the right. If no initial value is specified, the array element is not initialized. Only the last array in a declaration list can be initialized.

A PB-relative array allocates storage in the code segment for an array of constants. The entire PB-relative array must be initialized and cannot be changed during execution. PB-relative arrays can only be accessed within the procedure in which they are declared and they cannot be passed as parameters.

7-22. OWN ARRAYS. OWN arrays are allocated space in the DB-relative area instead of the Q-relative area. Thus, an OWN array retains its values from one execution of the procedure to the next. However, the array can only be referenced within the procedure in which it is declared. An OWN array can be passed as a parameter, however. An OWN array must have defined bounds and may be initialized.

The form of an OWN array declaration is:

OWN [*type*] ARRAY [*own-dec,...,own-dec,*]*own-dec-initial;*

EXAMPLES:

```
OWN ARRAY L1(0:10),L2(0:10),L3(0:10):= 10(17),20;
OWN REAL ARRAY R1(0:10):= 5(2.0),6(3.5);
```

where

own-dec
is of the form: *array-name(lower:upper)*

own-dec-initial
is of the form: *array-name(lower:upper)*[:=*value-group,...,value-group*]]

array-name
is a legal SPL *identifier*.

lower
specifies the lower bound of the array. It is a decimal, based, composite or equated single-word integer constant.

upper
specifies the upper bound of the array. It is a decimal, based, composite, or equated single-word integer constant.

value-group
is either of the following:

1. *initial-value*
2. *repetition-factor (initial-value* [,...,*initial-value*])

initial-value
is an SPL numeric or string constant.

repetition-factor
specifies the number of times the initial value list will be used to initialize the array elements. The *repetition-factor* can be any unsigned non-zero decimal, based, composite, or equated single-word integer constant.

7-23. EXTERNAL ARRAYS. An EXTERNAL array declaration is used to link global arrays to arrays in procedures compiled separately from the main program. The *array-names* must be the same as used in the global declarations and the GLOBAL attribute must have been specified.

The form of an EXTERNAL array declaration is:

EXTERNAL [*type*] **ARRAY** *array-name* { (*) / (@) } [[,...,*array-name* { (*) / (@) }]];

EXAMPLES:

```
EXTERNAL ARRAY L1(*),L2(@);
EXTERNAL REAL ARRAY R1(@);
```

where

type
specifies the data type of the array. The type may be INTEGER, LOGICAL, BYTE, DOUBLE, REAL, or LONG. If not specified, the array is LOGICAL.

array-name
is a legal SPL *identifier*.

Array bounds are not specified in an EXTERNAL array declaration. An asterisk (*) is used to signify a direct array and an @ is used for an indirect array.

7-24. LOCAL POINTER DECLARATIONS

A pointer declaration defines an identifier as a "pointer" — a single word quantity used to contain the DB-relative address of another data item — the object of the pointer. A pointer declaration defines the following attributes of a pointer:

- The data type of the object of the pointer.
- The storage allocation method.
- The initial address to be stored in the pointer (optional).

When the pointer is accessed, the object is accessed indirectly through the pointer address. The object is assumed to be (or treated as if it were) the type of the pointer.

As with simple variables and arrays, there are three types of local pointer declarations: standard, OWN, and EXTERNAL. The standard pointer declaration can allocate the next available Q-relative cell or specify a location relative to a base register or another data item to be used as the pointer location. OWN pointer declarations allocate a DB-relative cell for each pointer at the beginning of program execution. EXTERNAL pointer declarations link global pointers in a separately compiled main program to a pointer in a procedure (the global pointers must be declared with the GLOBAL attribute).

There are two methods which can be used to link global pointers to pointers in separately compiled procedures. The first method is to use the GLOBAL attribute (see paragraph 3-4) in the global pointer declaration and the EXTERNAL attribute in the local pointer declaration. The identifiers in both declarations must be the same and the Segmenter is responsible for making the correct linkages. The second method is to include dummy global declarations at the beginning of subprogram compilations. All global declarations must be included, even for identifiers not referenced in the subprogram, and they must be in the same order as in the main program. It is possible, although not recommended, to use different identifiers for the same pointer, but you are responsible for keeping them straight. The second method is faster and requires less space in the USL (User Subprogram Library) files, but does not protect you against improper linkages.

7-25. STANDARD LOCAL POINTERS. A standard local pointer declaration specifies identifier(s) which can either be allocated storage each time the procedure is called or which use locations relative to base registers or other identifiers. Local pointers cannot be referenced outside the procedure in which they are declared. See section 4-4 for examples and information about addresses and pointers.

The form of a standard local pointer declaration is:

[*type*] POINTER *pointer-dec* [,...,*pointer-dec*];

EXAMPLES:

```
INTEGER A; LOGICAL B;
BYTE POINTER P:=@A;
INTEGER ARRAY N(0:10);
INTEGER POINTER PN:=@N(5);
POINTER P3=DB+2,P4,P5:=@A,P6:=B;
```

where

pointer-dec
is one of the following:

1. *pointer-name* [:= @*reference-identifier* [(*index*)]]

   This form allocates the next available Q-relative cell for the pointer variable. If the :=@ *reference-identifier* is used, the pointer is initialized to the address of the *reference-identifier* or array-element if an index is included. The *reference-identifer* must be declared first.

2. *pointer-name* = *reference-identifier* [*sign offset*]

   This form is used to equivalence a pointer variable to a location relative to another identifier. Space is not allocated for the pointer nor can the pointer be initialized. The resulting address for the pointer variable must be within the direct address range of the base register which the *reference-identifier* uses.

3. *pointer-name* = *register* [*sign offset*]

   This form is used to equivalence a pointer variable to a location relative to a base-register. Space is not allocated for the pointer nor can the pointer be intitialized. The resulting address for the pointer variable must be within the direct address range of the specified base-register.

4. *pointer-name* = *offset*

   This form is used only in privileged mode. It is the offset in System DB. The pointer reference must always be subscripted and cannot be preceded by '@'.

type
specifies the data type of the pointer variables in the declaration. The type can be INTEGER, LOGICAL, BYTE, DOUBLE, REAL, or LONG.

pointer-name
is a legal SPL *identifier.*

reference-identifier
is any legal SPL *identifier* which has been declared as a data item except DB,PB,Q,S, or X.

register
specifies the base register in a register reference. The *register* can be DB, Q, or S.

sign
is + or −.

offset
is an unsigned decimal, based, composite, or equated integer within the direct address range as shown below.

Register	Sign	Offset
DB	+	0 to 255
Q	+	0 to 127
Q	–	0 to 63
S	–	0 to 63
ST (system table)	+	> = 0

index
indicates the array element whose address the pointer will contain. The index can be any decimal, based, composite, or equated single-word integer constant.

Pointers are initialized with addresses of other variables or constants. The method is to follow the pointer with :=@ and a data reference (simple variable, pointer element, or array element or := constant). The address of the specified data item, adjusted to the address type of the pointer, is stored in the cell allocated for the pointer. BYTE pointers contain DB-relative byte addresses, whereas all other types of pointers contain DB-relative word addresses.

See "Pointers" (paragraph 2-20) for methods of referring to and through pointers. Pointers can be indexed like arrays and can contain word or byte addresses.

Pointers can be declared with all data types; if no type is specified, type LOGICAL is assumed. The type determines what data type the object of the pointer is assumed to have. This allows objects declared with one type to be accessed as another data type by accessing them through pointers.

Pointers which are not address referenced are allocated the next available Q-relative location and can be initialized. Pointers which are referenced use the address of the referenced item or the specified register relative location and cannot be initialized.

7-26. OWN POINTERS. OWN pointers are allocated space in the DB-relative area instead of the Q-relative area. Thus, an OWN pointer retains its value from one execution of the procedure to the next. However, the pointer can be referenced only within the procedure where it is declared. An OWN pointer cannot be initialized.

The form of an OWN pointer declaration is:

OWN [*type*] POINTER *pointer-name* [,...,*pointer-name*];

EXAMPLES:

```
OWN POINTER PTR;
OWN REAL POINTER RPTR1,RPTR2;
```

where

type
specifies the data type of the objects of the pointers in the declarations. The *type* may be INTEGER, LOGICAL, BYTE, DOUBLE, REAL, or LONG. If not specified, type LOGICAL is assumed.

pointer-name
is a legal SPL *identifier*.

7-27. EXTERNAL POINTERS. An EXTERNAL pointer declaration is used to link global pointers for referencing in procedures compiled separately from the main program. The identifiers must be the same as used in the global declarations and the GLOBAL attribute must have been specified.

The form of an EXTERNAL pointer declaration is:

EXTERNAL [*type*] **POINTER** *pointer-name* [,...,*pointer-name*];

EXAMPLES:

```
EXTERNAL REAL POINTER RPTR1,RPTR2;
EXTERNAL POINTER PTR1;
```

where

type
specifies the data type of the objects of the pointers in the declaration. The *type* may be INTEGER, LOGICAL, BYTE, DOUBLE, REAL, or LONG. If not specified, type LOGICAL is assumed.

pointer-name
is a legal SPL *identifier*.

7-28. LABEL DECLARATIONS

A label declaration specifies that an identifier is used in the program as a label to identify a statement. Labels are referenced when it is necessary to transfer control to a specific statement; they need not be declared explicitly unless the programmer wishes.

The form of a label declaration is:

LABEL *label* [,...,*label*];

EXAMPLES:

```
LABEL L1,L2,L3;
LABEL L;
```

where

label
is a legal SPL *identifier*.

Labels are used to identify statements as follows:

```
LABEL L1;
  •
  •
  •
L1:A:= B;
```

The syntax for labeled statements is given in paragraph 1-3. In SPL, a label implicitly declares itself when it is used to identify a statement, as the object of a GO TO statement, or in a switch declaration. It need not be explicitly declared in a label declaration except as desired for documentation purposes. See "GO TO Statement" (paragraph 5-2) and "Switch Declaration" (below) for use of labels.

7-29. SWITCH DECLARATIONS

A switch declaration relates an identifier to an ordered set of labels. The switch is accessed as a computed (indexed) GO TO statement. The purpose of a switch is to allow selective transfer of control to any of the statements identified by the labels in the switch declaration.

The form of a switch declaration is:

SWITCH ***switch-name*** **:=** ***label*** **[,...,*label*];**

EXAMPLES:

```
SWITCH SW:= L1,L2,L3,L4,L5,L6,L7,L8,L9;
SWITCH ERROR'SELECT:= ERR1,ERR2,ERR3,ERR4,ERR5,ERR6;
```

where

switch-name
is a legal SPL *identifier*.

label
identifies the statement to which control is transfered when the switch is referenced.

Only one *switch-name* can be declared in each switch declaration. Associated with each *label* in the label list, from left-to-right, is an ordinal integer from 0 to $n-1$, (where n is the number of labels in the list). This integer indicates the position of the *label* in the list. Each position in the list must contain a *label* — null elements are not allowed. When the switch is referenced by a GO TO statement (see paragraph 5-2), the value of an integer subscript determines which *label* is selected from the list. Bounds checking in this selection is optional. Entry points are not allowed in switch declarations. Switch labels may not occur in subroutines.

7-30. ENTRY DECLARATION

The purpose of a local entry declaration is to specify multiple entry points to a procedure beyond the implicit entry point which is the first statement of the procedure. Each entry identifier must occur somewhere in the body as a statement label, but cannot be the object of a GO TO.

The form of an entry declaration is:

ENTRY *label* [,...,*label*];

EXAMPLES:

```
ENTRY P1,P2,P3;
ENTRY P1;
```

where

label
identifies the statement to be used as an alternate entry point.

By substituting an entry point label for the *procedure-name* in a function designator or a procedure call statement, the procedure can be entered at an alternate entry point. Refer to paragraph 4-6 for the form of a function designator and paragraph 5-8 for the form of a procedure call statement.

7-31. DEFINE DECLARATION AND REFERENCE

A define declaration assigns a block of text to an identifier. Thereafter, when the identifier is used in the program, the assigned text replaces the identifier. This provides a convenient abbreviation mechanism to avoid repeating long constructs used many times in a program.

The form of a define declaration is:

DEFINE *identifier* = *text#* [,...,*identifier* = *text#*] ;

EXAMPLES:

```
DEFINE AS=ASSEMBLE(#,LA=LONG ARRAY#;
DEFINE DA=DOUBLE ARRAY#;
```

where

identifier
is a legal SPL *identifier*.

text
specifies the block of text to be substituted when the define is referenced. The *text* can be any sequence of ASCII characters; however, # can only be used within a string.

A define reference may occur anywhere except within an identifier, string, or constant. The *text* should make sense when inserted where the define is referenced.

At declaration time, a define has no effect on the compilation of the program. It has effect only in the context where it is referenced. For this reason, undeclared identifiers can appear in defines as long as they have been declared when the define is referenced. Similarly, the define text is checked for syntax errors in the context where it is referenced, not where it is declared.

Define declarations can be nested, that is, define identifiers can be used in other definitions, but they cannot be recursive, that is, a define identifier must not appear within its own text, since this leads to infinite nesting when the define is referenced.

The number sign (#) terminates a define text only if it is not contained in a string. For example, the string "ABCD#"# is valid text terminated by the second #. Incomplete comments cannot appear in DEFINEs.

Only one block of text can be assigned to a particular identifier.

For example, here are some sample define declarations and references.

```
DEFINE I= ARRAY B(0:1)#;
INTEGER I; <<INTEGER ARRAY B(0:1);>>
DEFINE SUM= A+ B+ C+ D+ E#;
J:= SUM; <<J:= A+ B+ C+ D+ E;>>
```

7-32. EQUATE DECLARATION AND REFERENCE

An equate declaration assigns an integer value determined by an expression of integer constants and other equates, to an identifier. The equate mechanism is only a documentation and maintenance convenience; it does not allocate any run-time storage, but merely provides a form of consistent identification for constants. When an equate identifier is used, the appropriate constant is substituted in its place. When equates are used instead of actual constants, programs can be updated easily; instead of replacing every occurrence of a constant, only the equate declaration is changed.

The form of an equate declaration is:

EQUATE ***identifier*** **=** ***equate-expression*** **[,...,** ***identifier*** **=** ***equate-expression*****];**

EXAMPLES:

```
EQUATE BELL= 7,CR= %15;
EQUATE N= 100,M= N+ 50;
```

where

identifier
is a legal SPL *identifier*.

equate-expression
can be either one of or a combination of two forms:

[*sign*] *unsigned-integer* [*operator unsigned-equate-expr*]

(*equate-expression*)

sign
is + or −.

unsigned-integer
is an unsigned decimal, based, composite, or equated single-word integer constant.

operator
is +,−,*, or /.

unsigned-equate-expr
is an unsigned equate-expression.

The value to be assigned to an equate identifier is determined by an equate expression. Equate expressions consist of operators (*,/,+,−), unsigned integers, including previously defined equated integers, and parentheses. Evaluation of the expression proceeds from left to right, except that multiplication and division (*,/) are done before addition and subtraction (+,−) and expressions in parentheses are done before the operators that surround them. The value of an equate expression must fit in a single-word or it will be truncated on the left. Since equate identifiers can be used in equate expressions, a series of related equate declarations can be set up such that changing only the first changes all the rest.

Equate identifiers can be used anywhere in the program that an integer or unsigned integer constant is allowed.

For example, here are some sample equate declarations and references:

```
EQUATE M=1,N=M+1,P=N+1;
EQUATE T=20*P/(20-P+M);
J:=136*T;
  <<M=1, N=2, P=3, T=3, J=408>>
```

7-33. PROCEDURE BODY

The procedure body consists of the local declarations and the statements of the procedure, preceded by a BEGIN and terminated by an END;. The body can contain any executable SPL statements. If the body does not contain any local declarations and only one statement, the BEGIN-END pair can be omitted. The end of the body generates an EXIT instruction; additional exits can be generated using the RETURN statement (see "RETURN Statement", paragraph 5-14).

EXAMPLES

```
PROCEDURE BLANKBUF <<Name>>
   (BUFFER,COUNT); <<Formal Parameters>>
   VALUE COUNT; <<Value part>>
   LOGICAL ARRAY BUFFER; <<Specification>>
   INTEGER COUNT; <<Specification>>
   <<Empty Option Part>>
<<Procedure-Body>>
   BEGIN
      LOGICAL BLANKWORD := " "; <<Data Group>>
      BUFFER := BLANKWORD; <<Statements>>
      MOVE BUFFER(1):=BUFFER,(COUNT);
   END; <<End Procedure Declaration>>

<<Sample Function and Call>>
   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); <<Equivalent to NIX:=NUM/((4+5)*6);>>
   END.

<<OPTION FORWARD example>>
   PROCEDURE PROC1; OPTION FORWARD; <<Dummy declaration>>
   PROCEDURE PROC2; OPTION FORWARD; <<Dummy declaration>>

   PROCEDURE PROC1; <<Real declaration>>
      IF X=(Y:=Y+1) THEN PROC2;
   PROCEDURE PROC2; <<Real declaration>>
      IF X= (Z:=Z+1) THEN PROC1;
```

7-34. INTRINSIC DECLARATIONS

An intrinsic declaration specifies that one or more of the system-provided procedures (intrinsics) will be used by the program. Intrinsics are pre-compiled procedures supplied to SPL programmers for performing input/output, file access, and utility functions as part of the Multiprogramming Executive (MPE). SPL provides a simple interface to intrinsics because SPL does not have built-in constructs for input/output as provided by FORTRAN, BASIC, COBOL, and other high-level languages. Input and output of data in SPL programs must be performed with the MPE file system intrinsics. The user can also declare intrinsics from his own intrinsic file.

The form of an intrinsic declaration is:

INTRINSIC [(*file*)] *procedure-name* [,...,*procedure-name*];

EXAMPLES:

```
INTRINSIC FOPEN, FREAD, FWRITE, PRINT, READ;
INTRINSIC (MYFILE) ASCII, CONVERT, OUTPUT, DATA'MAP3;
```

where

file
is any valid random-access file of the operating system.

procedure-name
is the name of an intrinsic procedure.

Unless an intrinsic file is specified, the procedure names in an intrinsic declaration must be included in an installation-defined intrinsic file. The SPL compiler searches the file for the intrinsic name and, if it is found, inserts the declaration for the intrinsic into the program. The declaration is equivalent to an OPTION EXTERNAL procedure declaration (see "Procedure Declaration", paragraph 7-2) and specifies the procedure's parameters, etc. Operating System intrinsics are described in the *MPE Intrinsics Reference Manual*. These intrinsics are called like normal external procedures.

The programmer can specify his own intrinsic file in parentheses. In this case, the compiler searches for the procedure name and declaration in the file specified, rather than in the system file. Appendix C describes how to build intrinsic files.

7-35. SUBROUTINE DECLARATION

A subroutine declaration defines an identifier as a subroutine and specifies what attributes the subroutine will have:

- Data type of result for function subroutines.
- Type and number of formal parameters.
- Statements of the subroutine body.

Subroutines are called by the identifier and a list of actual parameters. Subroutines can be declared either globally or locally, but global subroutines cannot be accessed locally. Local declarations are not allowed within subroutines.

The form of a subroutine declaration is:

[*type*] **SUBROUTINE** *subroutine-name*
[(*formal-parm* [,...,*formal-parm*]); [*value-part*] *specification-part*]; *statement;*

where

type
indicates that the procedure is a function procedure that returns a value of the specified data type. The *type* is INTEGER, LOGICAL, BYTE, DOUBLE, REAL, or LONG.

subroutine-name
is an SPL *identifier* used to identify the subroutine.

formal-parm
is an SPL *identifier* which is used as a local *identifier* to reference an *actual-parameter*.

value-part
indicates which formal parameters are to be passed by-value. All parameters which are not specified in the *value-part* are passed by-reference. The *value-part* is of the form: VALUE *formal-parm* [,...,*formal-parm*];

specification-part
indicates the characteristics of each formal parameter. The *specification-part* is of the form: *specification* [;...;*specification*]

specification
is one of the following:

type formal-parm [,...,*formal-parm*]
[*type*] ARRAY *formal-parm* [,...,*formal-parm*]
[type] POINTER *formal-parm* [,...,*formal-parm*]
[*type*] PROCEDURE *formal-parm* [,...,*formal-parm*]

statement
is an executable SPL single or compound statement (see sections IV through VI).

Subroutines have the same parameter conventions as procedures except that options such as VARIABLE, EXTERNAL, and CHECK are not provided and subroutines cannot be passed labels. Subroutines can have a data type and can be functions just as procedures can. The subroutine body consists of an executable SPL statement, including a compound statement, but cannot contain declarations. Global subroutines can reference global variables and local subroutines can reference both local and global variables. Subroutines can be called recursively. Subroutines are called using the SCAL or LRA and BR instructions and return using the SXIT instruction. For details on calling subroutines, see "Function Designator" (paragraph 4-6) and "Subroutine Call Statement" (paragraph 5-13).

NOTE

You must not explicitly modify the stack within a subroutine without immediately correcting for any changes. All subsequent parameter addressing may be incorrect and S may not point to the return address when SXIT is executed.

EXAMPLES:

```
INTEGER SUBROUTINE S(A,B,C);
    VALUE A,B,C;
    INTEGER A,B,C;
    S:= (A- 2)+ (B*C);

SUBROUTINE ZERO (ARRY,HISUB);
    VALUE HISUB;
    INTEGER HISUB;
    INTEGER ARRAY ARRY;
BEGIN
  I:= 0; <<global variable>>
  WHILE I <= HISUB DO
      BEGIN
        ARRY(I):= 0;
        I:= I+ 1;
      END;
END;
```

Table 7-1. Procedures vs. Subroutines

PROCEDURES	SUBROUTINES
Parameters	Parameters
Functions	Functions
Preserves calling environment and establishes its own environment	Executes within the calling environment
Local variables	No local variables
High overhead	Very low overhead — extremely fast
Allows for efficient segmentation	Must rewrite to segment subroutines
Can be called from any procedure or from outer block	If declared in the outer block, callable only from outer block
	If declared in a procedure, callable only from that procedure

SECTION VIII INPUT/OUTPUT

SECTION VIII

8-1. INTRODUCTION TO INPUT/OUTPUT

To perform input/output in SPL, you must call MPE intrinsics directly since SPL does not have any input/output statements. This section presents examples of some of the more common input/output intrinsics. For a complete description of all the system intrinsics, refer to the *MPE Intrinsics Reference Manual.* For a complete discussion of MPE file commands, refer to the *MPE Commands Reference Manual.*

Below is a list of some of the more common input/output intrinsics and their names.

Table 8-1. Common Input/Output Intrinsics

FOPEN	Opens a file
READ	Reads an ASCII string from the job/session input device ($STDIN)
READX	Reads an ASCII string from the job/session input device ($STDINX)
FREAD	Reads a logical record from a sequential file on any device to the user's data stack
FREADDIR	Reads a logical record from a direct access file to the user's data stack
PRINT	Prints character string on job/session list device
FWRITE	Writes a logical record from the user's stack to a sequential file on any device
FWRITEDIR	Writes a logical record from the user's stack to a direct access disc file
FUPDATE	Updates a logical record residing in a disc file
FCLOSE	Closes a file
FCHECK	Requests details about file input/output errors
FCONTROL	Performs control operations on a file or terminal device
FSPACE	Spaces forward or backward on a file

All input/output is performed on a word basis using two bytes per word. Although you can pass a byte array to a system intrinsic, the address is converted to a word address and a warning message issued. To avoid this, you can use array equivalencing:

```
BYTE ARRAY BUF(0:71);
ARRAY WBUF(*)= BUF;
```

For all non-input/output operations, you would use BUF, (for example, to prepare the buffer for writing), whereas for all calls to the input/output intrinsics, you would pass WBUF.

SPLIT-STACK OPERATIONS: During normal operation, the DB register points to the user process stack. Some operations with extra data segments require that DB be set to the base of the extra data segment while DL and all other data registers remain associated with the stack. When a process is operating in this mode, it is said to have a split stack. Several of the MPE intrinsics deal with DB in this manner; however, you need not be concerned with the mechanics of the operation because, while the stack is "split", only system code is executing. It is possible, however, if you are a privileged mode user, to force your process to operate in split-stack mode explicitly. If you do this, you must recognize that some of the normal callable intrinsics may not be called when DB does not point to the stack. Such intrinsics, if called by a privileged process in split-stack mode, can result in system failures. If you are not a privileged mode user, you need not concern yourself with this restriction and you may assume that intrinsics will not operate in split-stack mode unless otherwise stated.

WARNING

The normal checks and limitations that apply to the standard users in MPE are bypassed in privileged mode. It is possible for a privileged mode program to destroy system integrity, including the MPE operating system software itself. Hewlett-Packard cannot be responsible for system integrity when programs written by users operate in privileged mode.

8-2. OPENING A NEW DISC FILE

(Please refer to the *MPE Intrinsics Reference Manual* for details on the FOPEN procedure.)

Figure 8-1 contains an SPL program which opens two files: a card reader file and a new disc file.

The second FOPEN call in figure 8-1

```
OUT:= FOPEN(OUTPUT,%4,%101,128);
```

opens the new disc file. The parameters specified are

formaldesignator DATAONE, which is contained in the byte array OUTPUT

foptions %4, for which the bit pattern is as follows:

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	0	0	0	0	1	0	0
														4	

The above bit pattern specifies the following file options:

Domain: New file, no search of system or job temporary file directory is necessary. Bits (14:2) = 00.

ASCII/Binary: ASCII. Bit (13:1) = 1.

aoptions %101, for which the bit pattern is as follows:

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	0	0	0	0	1
								1			0			1	

The above bit pattern specifies the following access options:

Access Type: Write access only. Bits (12:4)= 0001 Exclusive: Exclusive access. Bits (8:2)= 01.

All other parameters are omitted from the FOPEN intrinsic call.

```
PAGE 0001   HEWLETT-PACKARD 32100A.05.1  SPL/3000  TUE, OCT  7, 1975, 10:30 AM


00001000  00000 0   $CONTROL USLINIT
00002000  00000 0   BEGIN
00003000  00000 1     BYTE ARRAY INPUT(0:6):="INFILE ";
00004000  00005 1     BYTE ARRAY DEV(0:4):="CARD ";
00005000  00004 1     BYTE ARRAY OUTPUT(0:7):="DATAONE ";
00006000  00005 1     ARRAY BUFFER(0:127);
00007000  00005 1     INTEGER IN,OUT,LGTH;
00008000  00005 1
00009000  00005 1     INTRINSIC FOPEN,FREAD,FWRITE,FCLOSE,PRINT'FILE'INFO,QUIT;
00010000  00005 1
00011000  00005 1     << END OF DECLARATIONS >>
00012000  00005 1
00013000  00005 1         IN:=FOPEN(INPUT,%5,,40,DEV);          <<CARD READER>>
00014000  00012 1         IF < THEN                             <<CHECK FOR ERROR>>
00015000  00013 1           BEGIN
00016000  00013 2             PRINT'FILE'INFO(IN);            <<PRINT ERROR>>
00017000  00015 2             QUIT(1);                        <<ABORT>>
00018000  00017 2           END;
00019000  00017 1
00020000  00017 1         OUT:=FOPEN(OUTPUT,%4,%101,128);       <<NEW DISC FILE>>
00021000  00030 1         IF < THEN                             <<CHECK FOR ERROR>>
00022000  00031 1           BEGIN
00023000  00031 2             PRINT'FILE'INFO(OUT);           <<PRINT ERROR>>
00024000  00033 2             QUIT(2);                        <<ABORT>>
00025000  00035 2           END;
00026000  00035 1
00027000  00035 1   COPY'LOOP:
00028000  00035 1         LGTH:=FREAD(IN,BUFFER,40);            <<READ A CARD>>
00029000  00043 1         IF < THEN                             <<CHECK FOR ERROR>>
00030000  00044 1           BEGIN
00031000  00044 2             PRINT'FILE'INFO(IN);            <<PRINT ERROR>>
00032000  00046 2             QUIT(3);                        <<ABORT>>
00033000  00050 2           END;
00034000  00050 1         IF > THEN GO END'OF'FILE;             <<CHECK FOR EOF>>
00035000  00051 1
00036000  00051 1         FWRITE(OUT,BUFFER,LGTH,0);            <<COPY CARD TO DISC>>
00037000  00056 1         IF <> THEN                            <<CHECK FOR ERROR>>
00038000  00057 1           BEGIN
00039000  00057 2             PRINT'FILE'INFO(OUT);           <<PRINT ERROR>>
00040000  00061 2             QUIT(4);                        <<ABORT>>
00041000  00063 2           END;
00042000  00063 1
00043000  00063 1         GO COPY'LOOP;                         <<CONTINUE COPYING>>
00044000  00066 1
00045000  00066 1   END'OF'FILE;
00046000  00066 1         FCLOSE(OUT,%11,0);                    <<MAKE PERMANENT>>
00047000  00072 1         IF < THEN                             <<CHECK FOR ERROR>>
00048000  00073 1           BEGIN
00049000  00073 2             PRINT'FILE'INFO(OUT);           <<PRINT ERROR>>
00050000  00075 2             QUIT(5);                        <<ABORT>>
00051000  00077 2           END;
00052000  00077 1   END.
 PRIMARY DB STORAGE=%007;    SECONDARY DB STORAGE=%00213
 NO. ERRORS=000;             NO. WARNINGS=000
 PROCESSOR TIME=0:00:03;     ELAPSED TIME=0:00:44
```

Figure 8-1. Opening a New Disc File

Once the file is opened, the file number (used by other file system intrinsics when referencing this file) is returned to the variable OUT.

The condition code is checked with the

```
IF < THEN
```

statement. If the condition code is CCL, signifying that the FOPEN request was denied, the next four statements, starting with the BEGIN statement are executed.

```
PRINT'FILE'INFO(OUT);
```

calls the PRINT'FILE'INFO intrinsic, which prints a FILE INFORMATION DISPLAY on the standard list device, enabling you to determine the error number returned by FOPEN. The parameter (OUT) specifies the file number returned through the FOPEN intrinsic. If the file was not opened successfully, OUT=0, where 0 specifies that the FILE INFORMATION DISPLAY will reflect the status of the file referenced in the last call to FOPEN. See the *MPE Intrinsics Reference Manual* for a discussion of the FILE INFORMATION DISPLAY.

The QUIT intrinsic call

```
QUIT(2);
```

aborts the process. The parameter (2) is an arbitrary user-supplied number. When a QUIT intrinsic is executed, this number is printed as part of the resulting abort message, allowing you to determine, in the case of multiple QUIT intrinsic calls in a program, which specific QUIT call was executed.

NOTE

> The QUIT intrinsic causes MPE to close all files with no change. Thus, new files are deleted, old files are saved and assigned to the same domain to which they belonged previously.

8-3. READING A FILE IN SEQUENTIAL ORDER

(Please refer to the *MPE Intrinsics Reference Manual* for details on the FREAD procedure.)

To read records, or portions of records, from a file in sequential order, you use the FREAD intrinsic.

When the FREAD intrinsic executes, a logical record pointer advances to the next record. Then, the next time the FREAD intrinsic is called, the next record is read. Even if a portion of a record is read, a subsequent FREAD ignores the unread portion of the last record (because the logical record pointer has advanced) and begins reading the next record.

NOTE

> The logical record pointer is a number kept by MPE to indicate the next sequential record to be accessed in a file.

```
PAGE 0001   HEWLETT-PACKARD 32100A.05.1  SPL/3000  TUE, OCT  7, 1975, 10:30 AM


00001000   00000 0    $CONTROL USLINIT
00002000   00000 0    BEGIN
00003000   00000 1      BYTE ARRAY INPUT(0:6):="INFILE ";
00004000   00005 1      BYTE ARRAY DEV(0:4):="CARD ";
00005000   00004 1      BYTE ARRAY OUTPUT(0:7):="DATAONE ";
00006000   00005 1      ARRAY BUFFER(0:127);
00007000   00005 1      INTEGER IN,OUT,LGTH;
00008000   00005 1
00009000   00005 1      INTRINSIC FOPEN,FREAD,FWRITE,FCLOSE,PRINT'FILE'INFO,QUIT;
00010000   00005 1
00011000   00005 1      << END OF DECLARATIONS >>
00012000   00005 1
00013000   00005 1           IN:=FOPEN(INPUT,%5,,40,DEV);          <<CARD READER>>
00014000   00012 1           IF < THEN                             <<CHECK FOR ERROR>>
00015000   00013 1             BEGIN
00016000   00013 2               PRINT'FILE'INFO(IN);              <<PRINT ERROR>>
00017000   00015 2               QUIT(1);                          <<ABORT>>
00018000   00017 2             END;
00019000   00017 1
00020000   00017 1           OUT:=FOPEN(OUTPUT,%4,%101,128);       <<NEW DISC FILE>>
00021000   00030 1           IF < THEN                             <<CHECK FOR ERROR>>
00022000   00031 1             BEGIN
00023000   00031 2               PRINT'FILE'INFO(OUT);             <<PRINT ERROR>>
00024000   00033 2               QUIT(2);                          <<ABORT>>
00025000   00035 2             END;
00026000   00035 1
00027000   00035 1      COPY'LOOP:
00028000   00035 1           LGTH:=FREAD(IN,BUFFER,40);            <<READ A CARD>>
00029000   00043 1           IF < THEN                             <<CHECK FOR ERROR>>
00030000   00044 1             BEGIN
00031000   00044 2               PRINT'FILE'INFO(IN);              <<PRINT ERROR>>
00032000   00046 2               QUIT(3);                          <<ABORT>>
00033000   00050 2             END;
00034000   00050 1           IF > THEN GO END'OF'FILE;             <<CHECK FOR EOF>>
00035000   00051 1
00036000   00051 1           FWRITE(OUT,BUFFER,LGTH,0);            <<COPY CARD TO DISC>>
00037000   00056 1           IF <> THEN                            <<CHECK FOR ERROR>>
00038000   00057 1             BEGIN
00039000   00057 2               PRINT'FILE'INFO(OUT);             <<PRINT ERROR>>
00040000   00061 2               QUIT(4);                          <<ABORT>>
00041000   00063 2             END;
00042000   00063 1
00043000   00063 1           GO COPY'LOOP;                         <<CONTINUE COPYING>>
00044000   00066 1
00045000   00066 1      END'OF'FILE;
00046000   00066 1           FCLOSE(OUT,%11,0);                    <<MAKE PERMANENT>>
00047000   00072 1           IF < THEN                             <<CHECK FOR ERROR>>
00048000   00073 1             BEGIN
00049000   00073 2               PRINT'FILE'INFO(OUT);             <<PRINT ERROR>>
00050000   00075 2               QUIT(5);                          <<ABORT>>
00051000   00077 2             END;
00052000   00077 1      END.
 PRIMARY DB STORAGE=%007;     SECONDARY DB STORAGE=%00213
 NO. ERRORS=000;              NO. WARNINGS=000
 PROCESSOR TIME=0:00:03;      ELAPSED TIME=0:00:44
```

Figure 8-2. FREAD Intrinsic Example

The program shown in figure 8-2 reads a card file. The FREAD statement

```
LGTH:=FREAD(IN,BUFFER,40);
```

reads a record from the card reader file designated by the variable IN (the file number was assigned to IN when the FOPEN intrinsic opened the file) and transfers this record to the array BUFFER in the stack. The statement reads up to 40 words from the record, then returns a positive value to LGTH which indicates the actual length of the information transferred.

If an error occurs during execution of the FREAD intrinsic, a condition code of CCL is returned. The statement

```
IF < THEN
```

checks the condition code and, if the condition code is CCL, the next four statements are executed. The PRINT'FILE'INFO intrinsic call causes a FILE INFORMATION DISPLAY to be printed on the output device so that you can determine the error number returned by FREAD, and the QUIT intrinsic aborts the process.

When the end-of-file is encountered on the card file, a condition code of CCG is returned. The statement

```
IF > THEN GO END'OF'FILE;
```

checks for this condition code and, when it occurs, transfers program control to the label END'OF'FILE. If the end-of-file condition is not encountered, the FWRITE statement is executed and the

```
GO COPY'LOOP;
```

statement transfers program control back to the beginning of the copy loop. The FREAD intrinsic is called again and the next record is read.

8-4. WRITING RECORDS INTO A FILE IN SEQUENTIAL ORDER

(Please refer to the *MPE Intrinsics Reference Manual* for details on the FWRITE procedure.)

To write records, or portions of records, from your buffer to a file in sequential order, you use the FWRITE intrinsic.

When the FWRITE intrinsic executes, the logical record pointer advances to the next record. Then, the next time the FWRITE intrinsic is called, information is written into the next record position. When information is written to a file composed of fixed-length records (and buffering is not specified in the FOPEN call), the file system will pad all short records with binary zeros for a binary file, or ASCII blanks for an ASCII file to bring the records up to the fixed length required. If nobuff was specified in FOPEN, automatic buffering is not provided by MPE.

The FWRITE statement in figure 8-3

```
FWRITE(OUT,BUFFER,LGTH,0);
```

writes a record from the array BUFFER into the disc file designated by the variable OUT. The file number was assigned to OUT when the FOPEN intrinsic opened the file. The length of the record is specified by LGTH. LGTH was assigned its value when the FREAD intrinsic read the record and transferred it to BUFFER, so in this case the same number of words being read from the card reader are being written to the disc.

The *control* parameter is specified as 0, which specifies that no carriage control code is included in the record. Carriage control, of course, is not necessary for a disc file but the parameter is included because all of FWRITE's parameters are required.

A condition code of CCE signifies that the FWRITE request was granted. The statement

```
IF <> THEN
```

checks for a "not equal" condition code and, if CCG or CCL is returned, the next four statements are executed. The PRINT'FILE'INFO intrinsic causes a FILE INFORMATION DISPLAY to be printed on the output device, enabling you to determine the error number returned by FWRITE. The QUIT intrinsic aborts the process.

If CCE is returned, the next four statements are not executed, the GO COPY'LOOP statement is executed, and the FREAD and FWRITE intrinsic calls are repeated until FREAD detects the end of the card file.

```
PAGE 0001   HEWLETT-PACKARD 32100A.05.1  SPL/3000  TUE, OCT  7, 1975, 10:30 AM


00001000   00000 0    $CONTROL USLINIT
00002000   00000 0    BEGIN
00003000   00000 1      BYTE ARRAY INPUT(0:6):="INFILE ";
00004000   00005 1      BYTE ARRAY DEV(0:4):="CARD ";
00005000   00004 1      BYTE ARRAY OUTPUT(0:7):="DATAONE ";
00006000   00005 1      ARRAY BUFFER(0:127);
00007000   00005 1      INTEGER IN,OUT,LGTH;
00008000   00005 1
00009000   00005 1      INTRINSIC FOPEN,FREAD,FWRITE,FCLOSE,PRINT'FILE'INFO,QUIT;
00010000   00005 1
00011000   00005 1      << END OF DECLARATIONS >>
00012000   00005 1
00013000   00005 1           IN:=FOPEN(INPUT,%5,,40,DEV);          <<CARD READER>>
00014000   00012 1           IF < THEN                             <<CHECK FOR ERROR>>
00015000   00013 1             BEGIN
00016000   00013 2               PRINT'FILE'INFO(IN);              <<PRINT ERROR>>
00017000   00015 2               QUIT(1);                          <<ABORT>>
00018000   00017 2             END;
00019000   00017 1
00020000   00017 1           OUT:=FOPEN(OUTPUT,%4,%101,128);       <<NEW DISC FILE>>
00021000   00030 1           IF < THEN                             <<CHECK FOR ERROR>>
00022000   00031 1             BEGIN
00023000   00031 2               PRINT'FILE'INFO(OUT);             <<PRINT ERROR>>
00024000   00033 2               QUIT(2);                          <<ABORT>>
00025000   00035 2             END;
00026000   00035 1
00027000   00035 1      COPY'LOOP:
00028000   00035 1           LGTH:=FREAD(IN,BUFFER,40);            <<READ A CARD>>
00029000   00043 1           IF < THEN                             <<CHECK FOR ERROR>>
00030000   00044 1             BEGIN
00031000   00044 2               PRINT'FILE'INFO(IN);              <<PRINT ERROR>>
00032000   00046 2               QUIT(3);                          <<ABORT>>
00033000   00050 2             END;
00034000   00050 1           IF > THEN GO END'OF'FILE;             <<CHECK FOR EOF>>
00035000   00051 1
00036000   00051 1           FWRITE(OUT,BUFFER,LGTH,0);            <<COPY CARD TO DISC>>
00037000   00056 1           IF <> THEN                            <<CHECK FOR ERROR>>
00038000   00057 1             BEGIN
00039000   00057 2               PRINT'FILE'INFO(OUT);             <<PRINT ERROR>>
00040000   00061 2               QUIT(4);                          <<ABORT>>
00041000   00063 2             END;
00042000   00063 1
00043000   00063 1           GO COPY'LOOP;                         <<CONTINUE COPYING>>
00044000   00066 1
00045000   00066 1      END'OF'FILE;
00046000   00066 1           FCLOSE(OUT,%11,0);                    <<MAKE PERMANENT>>
00047000   00072 1           IF < THEN                             <<CHECK FOR ERROR>>
00048000   00073 1             BEGIN
00049000   00073 2               PRINT'FILE'INFO(OUT);             <<PRINT ERROR>>
00050000   00075 2               QUIT(5);                          <<ABORT>>
00051000   00077 2             END;
00052000   00077 1      END.
 PRIMARY DB STORAGE=%007;    SECONDARY DB STORAGE=%00213
 NO. ERRORS=000;             NO. WARNINGS=000
 PROCESSOR TIME=0:00:03;     ELAPSED TIME=0:00:44
```

Figure 8-3. FWRITE Intrinsic Example

8-5. UPDATING A FILE

(Please refer to the *MPE Intrinsics Reference Manual* for details on the FUPDATE procedure.)

To update a logical record of a disc file, you use the FUPDATE intrinsic.

The FUPDATE intrinsic affects the logical record (or block for NOBUF files) last accessed by any intrinsic call for the file named, and writes information from a buffer in the stack into this record. Note that the record number is not supplied in the FUPDATE intrinsic call; FUPDATE automatically updates the last record referenced in any intrinsic call.

The file containing the record to be updated must have been opened with the update aoption specified in the FOPEN call and must not contain variable-length records.

Figure 8-4 contains a program that opens an old disc file and updates records in the file. The update information (employee number) is entered from a terminal (the program was run interactively) into a buffer in the stack, then the contents of the buffer are used to update the record.

The statement

```
LGTH:= FREAD(DFILE1,BUFFER,128);
```

reads an employee record from the file specified by DFILE1 into the array BUFFER in the stack.

The statement

```
FWRITE(LIST,BUFFER,—20,%320);
```

then displays this record on the terminal ($STDLIST has been opened with the FOPEN intrinsic and the resulting file number was assigned to LIST).

The statement

```
DUMMY:= FREAD(IN,BUFFER(30),5);
```

reads an employee number, entered on the terminal ($STDIN has been opened with the FOPEN intrinsic and the resulting file number was assigned to IN), into word 30 of the array BUFFER.

The statement

```
FUPDATE(DFILE1,BUFFER,128);
```

then calls the FUPDATE intrinsic to update the last record accessed in the file specified by DFILE1. The contents of BUFFER (including the employee number entered from the terminal) are written into this record. Up to 128 words are written.

If the FUPDATE request was granted, a CCE condition code results. The statement

```
IF <> THEN FILERROR(DFILE1,9);
```

checks for a "not equal" condition code and, if such is the case, calls the error-check procedure FILERROR. The procedure FILERROR prints a FILE INFORMATION DISPLAY on the terminal, enabling you to determine the error number returned by FUPDATE, then aborts the programs's calling process.

```
PAGE 0001   HEWLETT-PACKARD 32100A.05.1  SPL/3000  TUE, OCT  7, 1975, 10:32 AM

00001000  00000 0   $CONTROL USLINIT
00002000  00000 0   BEGIN
00003000  00000 1     BYTE ARRAY DATA1(0:7):="DATAONE ";
00004000  00005 1     ARRAY BUFFER(0:127);
00005000  00005 1     INTEGER DFILE1,LGTH,DUMMY,IN,LIST;
00006000  00005 1
00007000  00005 1     INTRINSIC FOPEN,FREAD,FUPDATE,FLOCK,FUNLOCK,FCLOSE,
00008000  00005 1               PRINT'FILE'INFO,QUIT,FWRITE,FREAD;
00009000  00005 1
00010000  00005 1     PROCEDURE FILERROR(FILENO,QUITNO);
00011000  00000 1       VALUE QUITNO;
00012000  00000 1       INTEGER FILENO,QUITNO;
00013000  00000 1       BEGIN
00014000  00000 2         PRINT'FILE'INFO(FILENO);
00015000  00002 2         QUIT(QUITNO);
00016000  00004 2       END;
00017000  00000 1
00018000  00000 1     <<END OF DECLARATIONS>>
00019000  00000 1
00020000  00000 1         DFILE1:=FOPEN(DATA1,%5,%345,128);      <<OLD DISC FILE>>
00021000  00011 1         IF < THEN FILERROR(DFILE1,1);          <<CHECK FOR ERROR>>
00022000  00015 1
00023000  00015 1         IN:=FOPEN(,%244);                      <<$STDIN>>
00024000  00024 1         IF < THEN FILERROR(IN,2);              <<CHECK FOR ERROR>>
00025000  00030 1
00026000  00030 1         LIST:=FOPEN(,%614,%1);                 <<$STDLIST>>
00027000  00040 1         IF < THEN FILERROR(LIST,3);            <<CHECK FOR ERROR>>
00028000  00044 1
00029000  00044 1   UPDATE'LOOP:
00030000  00044 1         FLOCK(DFILE1,1);                       <<LOCK FILE/SUSPEND>>
00031000  00047 1         IF < THEN FILERROR(DFILE1,4);          <<CHECK FOR ERROR>>
00032000  00053 1
00033000  00053 1         LGTH:=FREAD(DFILE1,BUFFER,128);        <<GET EMPLOYEE RECD>>
00034000  00061 1         IF < THEN FILERROR(DFILE1,5);          <<CHECK FOR ERROR>>
00035000  00065 1         IF > THEN GO END'OF'FILE;              <<CHECK FOR EOF>>
00036000  00070 1
00037000  00070 1         FWRITE(LIST,BUFFER,-20,%320);          <<EMPLOYEE NAME>>
00038000  00075 1         IF <> THEN FILERROR(LIST,6);           <<CHECK FOR ERROR>>
00039000  00101 1
00040000  00101 1         DUMMY:=FREAD(IN,BUFFER(30),5);         <<EMPLOYEE NUMBER>>
00041000  00110 1         IF < THEN FILERROR(IN,7);              <<CHECK FOR ERROR>>
00042000  00114 1         IF > THEN GO END'OF'FILE;
00043000  00115 1
00044000  00115 1         FUPDATE(DFILE1,BUFFER,128);            <<EMPLOYEE RECORD>>
00045000  00121 1         IF <> THEN FILERROR(DFILE1,8);         <<CHECK FOR ERROR>>
00046000  00125 1
00047000  00125 1         FUNLOCK(DFILE1);                       <<ALLOW OTHER ACCESS>>
00048000  00127 1         IF <> THEN FILERROR(DFILE1,9);         <<CHECK FOR ERROR>>
00049000  00133 1
00050000  00133 1         GO UPDATE'LOOP;                        <<CONTINUE UPDATE>>
00051000  00140 1
00052000  00140 1   END'OF'FILE:
00053000  00140 1         FUNLOCK(DFILE1);                       <<ALLOW OTHER ACCESS>>
00054000  00142 1         IF <> THEN FILERROR(DFILE1,10);        <<CHECK FOR ERROR>>
00055000  00146 1
00056000  00146 1         FCLOSE(DFILE1,0,0);                    <<DISP=NO CHANGE>>
00057000  00151 1         IF < THEN FILERROR(DFILE1,11);         <<CHECK FOR ERROR>>
00058000  00155 1   END.
 PRIMARY DB STORAGE=%007;    SECONDARY DB STORAGE=%00204
 NO. ERRORS=000;             NO. WARNINGS=000
 PROCESSOR TIME=0:00:03;     ELAPSED TIME=0:00:17
```

Figure 8-4. FUPDATE Intrinsic Example

8-6. NUMERIC DATA INPUT/OUTPUT

There are several intrinsics available for converting integer data for transfer between an ASCII file and the data stack. These intrinsics are as follows:

- ASCII — Converts 16-bit binary number to ASCII representation.
- DASCII — Converts 32-bit binary number to ASCII representation.
- BINARY — Converts an ASCII numeric string to a 16-bit binary numeric.
- DBINARY — Converts an ASCII numeric string to a 32-bit binary number.

(Please refer to the *MPE Intrinsics Reference Manual* for a complete description of these intrinsics.)

For handling floating point numbers, refer to the EXTIN' and INEXT' procedures in the *Compiler Library Reference Manual.*

8-7. FILE EQUATIONS

The standard attributes of files used by an SPL program can be modified through the use of the MPE :FILE command.

NOTE

Read the discussion of files in the *MPE Commands Reference Manual* before attempting to change file attributes with the :FILE command.

The specifications in a :FILE command do not take effect until the compiled program is running and the referenced file is opened. The :FILE command specifications hold throughout the entire program unless superseded by another :FILE command or revoked by a :RESET command. At job or session termination, however, all :FILE commands are cancelled.

COMPILER COMMANDS

SECTION IX

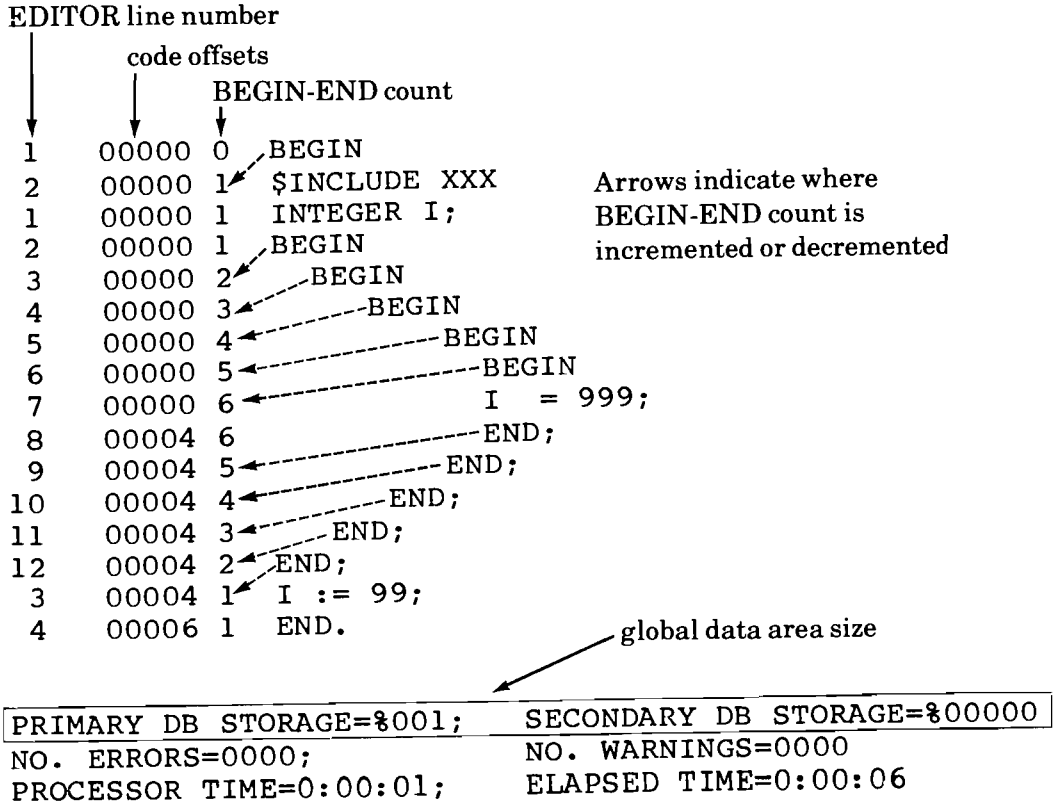
9.0 COMPILER FORMAT

A compiler listing presents three groups of numbers preceding the program statements. The first group shows the Editor line numbers of the listing file in decimal format. The second column of five numbers indicates the machine instruction code reference which is RBM-relative. The third set gives the BEGIN-END count, or level.

The BEGIN-END count is useful information for program debugging in locating BEGIN-END pair mismatches. This is the third group of numbers listed in a compile. It indicates the nesting level of the statements that follow the BEGIN or END. The count starts at zero and is incremented by one after each BEGIN statement; it is decremented by one after each END statement. Since the last END statement ends the compile process, the BEGIN-END count is never decremented to zero.

NOTE

Pressing CONTROL-Y during a compilation causes the current line number to be displayed along with the number of errors and warnings.



9-1. USE AND FORMAT OF COMPILER COMMANDS

In general, compiler options such as source input merging, listing, format specification, or warning message suppression are determined by default settings assigned by the compiler. However, the user can override these settings and select different options by issuing compiler commands. These commands take effect only after access to the compiler is established. They are directed only to the compiler and are not effective during program execution.

Compiler commands differ in both function and format from compiler language source statements, and thus are not considered true SPL statements even though they are part of the source program file. The SPL compiler commands do conform, however, to the general formats for other HP 3000 language translators such as FORTRAN, COBOL, and RPG. For each function used by more than one language translator, the same command name is used and, in most cases, the same command parameters also apply.

The general form of a compiler command is:

$[$]*command-name* [*parameter,...,parameter*]

EXAMPLES:

```
$CONTROL CODE,ADR,MAP
$$PAGE
$TITLE "UPDATE PROGRAM"
```

where

command-name
specifies the compiler command. The *command-name* is one of the following: CONTROL, IF, SET, TITLE, PAGE, EDIT, TRACE or COPYRIGHT.

parameter
specifes an option of the compiler command. The form of a parameter is dependent on the *command-name* and is discussed with the appropriate command. In general a parameter is one of the following:

character-string
symbolic-name
keyword [=*sub-parameter*]

The first dollar sign ($) is required and must be in column 1. The second dollar sign is optional. If specified, the command is not transmitted to the newfile if a newfile is created during compilation. The command-name must follow the first $ (or second $ if present) without any intervening spaces. The list of parameters is separated from the command-name by one or more spaces. Within the list, parameters are separated from each other by commas. Spaces are allowed before and after the parameters. The parameter list may continue through column 72 of the source record.

The sequence field (columns 73-80) of a record containing a compiler command is not part of the command; however, it may be used for sequence checking during editing and merging operations as described later under the EDIT command.

NOTE

Only upper-case letters, numbers, and special characters are used in compiler commands. When lower-case letters are entered as part of a command, the compiler interprets them as their upper-case equivalent except within character strings as defined below.

A *character-string* consists of a sequence of ASCII characters enclosed in quotation marks ("). Blank characters may be included in the string and null strings are allowed. Quotation marks within a string are entered as two adjacent quotation marks, ("") to distinguish them from the quotation marks that begin and end the string.

A *keyword* is a reserved word with respect to a given command; they are described under the appropriate commands. A *sub-parameter* is a *character-string,* a *symbolic name,* or a *decimal number.*

Comments may be included within any command. A comment is generally used to document the purpose of coding or to make notations about program logic. A comment is not interpreted as part of the command, and has no effect upon compilation. It is syntactically treated as a space and can appear in either of the following locations:

- Following the command-name, separated from it by at least one space.
- Preceding or following any parameter in the parameter list.

A comment cannot be embedded within a parameter; for instance, it cannot appear within a keyword, preceding or following an equals sign, or within a quoted string. Furthermore, a comment cannot be continued from one record to the next.

A comment can contain any ASCII character. The comment must begin with two adjacent less-than signs (<<) and terminate with two adjacent greater-than signs (>>). Since adjacent greater-than signs terminate a comment, they cannot appear within the comment itself. The comment may continue through column 72.

The following examples illustrate various ways in which comments can be included in compiler commands.

1. Following the *command-name*:

```
$PAGE <<PAGE EJECT,NO TITLE CHANGE.>>
```

2. Following the last parameter in a parameter list:

```
$SET X1=ON,X2=ON,X3=ON<<SWITCHES 1-3 ON.>>
```

3. Embedded within the parameter list:

```
$SET X1=ON,X2=ON,<<LAST SW OFF>>X3=OFF
```

When the length of a command exceeds one physical record (source card or entry line), the user can enter an ampersand (&) as the last non-blank character of this record and continue the command on

the next record. This is called a continuation record. The text portion of the continuation record, in turn, must begin with a dollar sign ($) in column 1. Even when a command begins with double dollar signs, its continuation records still begin with only a single dollar sign. When EDIT/3000 is used to enter a source program containing compiler command continuation records, a space must be entered after the ampersand so the ampersand is not interpreted as an EDIT/3000 continuation line.

NOTE

A compiler command record must never be separated from its continuation record by an SPL source record.

In continuing a command onto another record, you cannot divide a primary command element (a *command-name, keyword, subparameter* — including strings, or comment) — no primary element is allowed to span more than one line.

When the compiler encounters a command containing one or more continuation records, each continuation record is concatenated to the preceding record beginning with the character following the $; each $ and continuation ampersand is replaced by a space.

The following command is continued onto a second record:

```
$CONTROL LIST,SOURCE,WARN,MAP,&
$CODE,LINES=36
```

It is interpreted as:

```
$CONTROL LIST,SOURCE,WARN,MAP, CODE,LINES=36
```

Even though a comment cannot be divided over more than one line, extensive commentary text requiring several lines can be entered by enclosing it within separate comments that each occupy one line.

The following command includes commentary text spread over three lines:

```
$CONTROL NOWARN <<WARNING MESSAGES ON TRIVIAL ERRORS>>&
$                <<WILL NOT BE LISTED, BUT MESSAGES ON>>&
$                <<FATAL ERRORS WILL APPEAR.>>
```

A command does not take effect until all of its parameters have been interpreted. Thus, a command that suppresses source listing output does not affect the listing of any continuation records within the command itself. Parameters are interpreted from left-to-right. In some cases, parameters may be redundant or supersede previous parameters within the same command. In other cases, certain parameters are allowed only once within a command.

In the following command, the redundant parameters LIST and NOLIST each appear twice:

```
$CONTROL LIST,NOLIST,NOLIST,LIST
```

Because the final redundant parameter in any $CONTROL command always takes effect, the above command is equivalent to:

```
$CONTROL LIST
```

A summary of the compiler commands for SPL appears in table 9-1.

Table 9-1. Compiler Command Summary

COMMAND	PURPOSE
$CONTROL	Restricts access to listfile; suppresses source text, object code, and symbol table listing; suppresses warning messages; sets maximum number of lines listed per page; sets maximum number of severe errors allowed; starts a new segment; initializes the USL file; lists mnemonics for code generated; assigns a name to the outer block; allows subprogram compilation; makes outer block privileged; makes outer block uncallable; lists address mode and displacement of variables declared.
$IF	Interrogates software switches for conditional compilation.
$SET	Sets software switches for conditional compilation.
$TITLE	Establishes or changes page title on listing.
$PAGE	Establishes or changes page title, and ejects page.
$EDIT	Specifies editing options during merging such as, omitting sections of old source program and re-numbering sequence fields.
$COPYRIGHT	Specifies copyright information to be copied to the list, USL, and program files.
$SPLIT	Enables split-stack checking.
$NOSPLIT	Disables split-stack checking.
$INCLUDE	Permits inclusion of text from another file into the SPL source file.

9-2. $CONTROL COMMAND

When you call the compiler without specifying a $CONTROL command, the following default options are in effect:

The compiler is given unrestricted access to *listfile*.

All source records passed to the compiler by its editor are listed unless the listfile and primary input file (normally the *textfile*) are assigned to the same terminal.

Warning messages are listed.

Listing of the symbol table is suppressed.

Listing of the object code generated is suppressed.

The number of lines appearing on each printed page (output to *listfile*) is a maximum of 60.

The maximum number of severe errors allowed before compilation is terminated is 100.

SPL is called in the program mode, as opposed to subprogram mode.

The segment name is SEG'.

The outer block name is OB'.

The mnemonic listing is suppressed.

The USL (User Subprogram Library) file is not initialized unless it is a new file.

Callable, non-privileged outer block.

The above default options can be overridden by entering the $CONTROL compiler command. This command allows you to restrict access to the listfile, suppress source record listings, produce object code and symbol table listings, change the maximum number of lines per printed page, and otherwise alter the normal compiler control options.

The form of the $CONTROL command is:

$[$]CONTROL ***parameter*** **[,...,*parameter*]**

EXAMPLES:

```
$CONTROL CODE,MAP,INNERLIST
$CONTROL NOLIST
```

where

parameter
specifies an option of the $CONTROL command. A *parameter* is one of the following: LIST, NOLIST,

SOURCE, NOSOURCE, WARN, NOWARN, MAP, NOMAP, AUTOPAGE, CODE, NOCODE, LINES = *nnnn*, ERRORS = *nnn*, USLINIT, DEFINE, SEGMENT = *segname*, ADR, INNERLIST, MAIN = *program-name*, UNCALLABLE, PRIVILEGED, or SUBPROGRAM [(*procedure-name*[*] [,*procedure-name*[*]]...)].

Each *parameter* in the parameter list specifies a different option as described below. Unless otherwise noted, each *parameter* can appear in a $CONTROL command placed anywhere in the source input. Each *parameter* remains in effect until explicitly cancelled by an opposing parameter (for example, NOLIST cancelling LIST), or until the compilation terminates. In any $CONTROL command, at least one *parameter* must be specified. Within the parameter list, the *parameters* can appear in any order. In the descriptions below, default parameters are shown in **boxes**

LIST
Allows the compiler unrestricted access to the *listfile*, permitting the SOURCE, MAP, CODE, and LINES parameters to take effect when issued. The LIST parameter remains in effect until a $CONTROL command specifying NOLIST is encountered.

NOLIST
Allows only source records that contain errors, appropriate error messages, and subsystem initiation and completion messages to be written to the *listfile*. NOLIST remains in effect until a $CONTROL command specifying LIST appears.

SOURCE
Requests listing of all source records, as edited by the compiler's editor, while LIST is in effect. When the compiler is called with *listfile* and the primary input file assigned to the same terminal, NOSOURCE is initially the default. In all other cases SOURCE is the default.

NOSOURCE
Suppress the listing of source text, cancelling the effect of any previous SOURCE parameter. NOSOURCE remains in effect until SOURCE is subsequently encountered.

WARN
Permits the reporting of doubtful minor error conditions in the source input. These reports are transmitted to the *listfile* in the form of a warning message. The WARN parameter remains in effect until a $CONTROL command specifying the NOWARN parameter is encountered.

NOTE

NOLIST does not suppress warning messages — they are suppressed solely by NOWARN.

NOWARN
Suppresses warning messages. The NOWARN parameter remains in effect until a $CONTROL command specifying WARN appears.

MAP
Requests printing of user-defined symbols and their addresses following the source text listing if LIST is in effect. Reference parameters are flagged with an 'R'. The MAP parameter remains in effect until a NOMAP parameter is encountered. Figure 9-1 shows a sample symbol map.

NOMAP
Suppresses printing of symbol map of user-defined symbols thereby cancelling any previous MAP parameter. The NOMAP option remains in effect until a MAP parameter is encountered.

```
00001000  00000 0   $CONTROL MAP
00002000  00000 0   BEGIN
00003000  00000 1     INTEGER I,J:=10;
00004000  00000 1     REAL R1,R2;
00005000  00000 1     ARRAY A(0:10);
00006000  00000 1   R1:=R2:=20E9;
00007000  00004 1   FOR I:=0 UNTIL J DO
00008000  00011 1     A(I):=2*I;
00009000  00022 1   END.
```

IDENTIFIER	CLASS	TYPE	ADDRESS
A	ARRAY	LOGICAL	DB+006
I	SIMP. VAR.	INTEGER	DB+000
J	SIMP. VAR.	INTEGER	DB+001
R1	SIMP. VAR.	REAL	DB+002
R2	SIMP. VAR.	REAL	DB+004
TERMINATE'	PROCEDURE		

```
PRIMARY DB STORAGE=%007;     SECONDARY DB STORAGE=%00013
NO. ERRORS=000;              NO. WARNINGS=000
PROCESSOR TIME=0:00:00;      ELAPSED TIME=0:01:16

END OF PROGRAM
:
```

Figure 9-1. Symbol Map

AUTOPAGE

Causes a page eject whenever a procedure declaration is the first token found on a line. If the declaration is preceded by "COMMENT" or "<<" no page eject will be issued; however, if the embedded "declaration" occurs on the second or later line of a comment, one will be issued. Similarly, any documentation placed before the procedure declaration will appear on the preceding page.

CODE

Requests listing of object code generated following the listing of the source text if LIST is in effect. The CODE parameter remains in effect until the NOCODE parameter is encountered. Figure 9-2 shows a sample CODE listing.

NOCODE

Suppresses listing of object code, thereby cancelling the effect of any previous CODE parameter. The NOCODE parameter remains in effect until a CODE parameter is encountered.

LINES=*nnnn*

Limits the number of lines printed on listfile to *nnnn* lines per page. Whenever the next line sent to listfile would overflow the line count (*nnnn*), the page is ejected and the standard page heading and two blank lines are printed at the top of the page, followed by the line to be transmitted. A page heading and its following two blank lines are counted against the total line count, *nnnn*. The

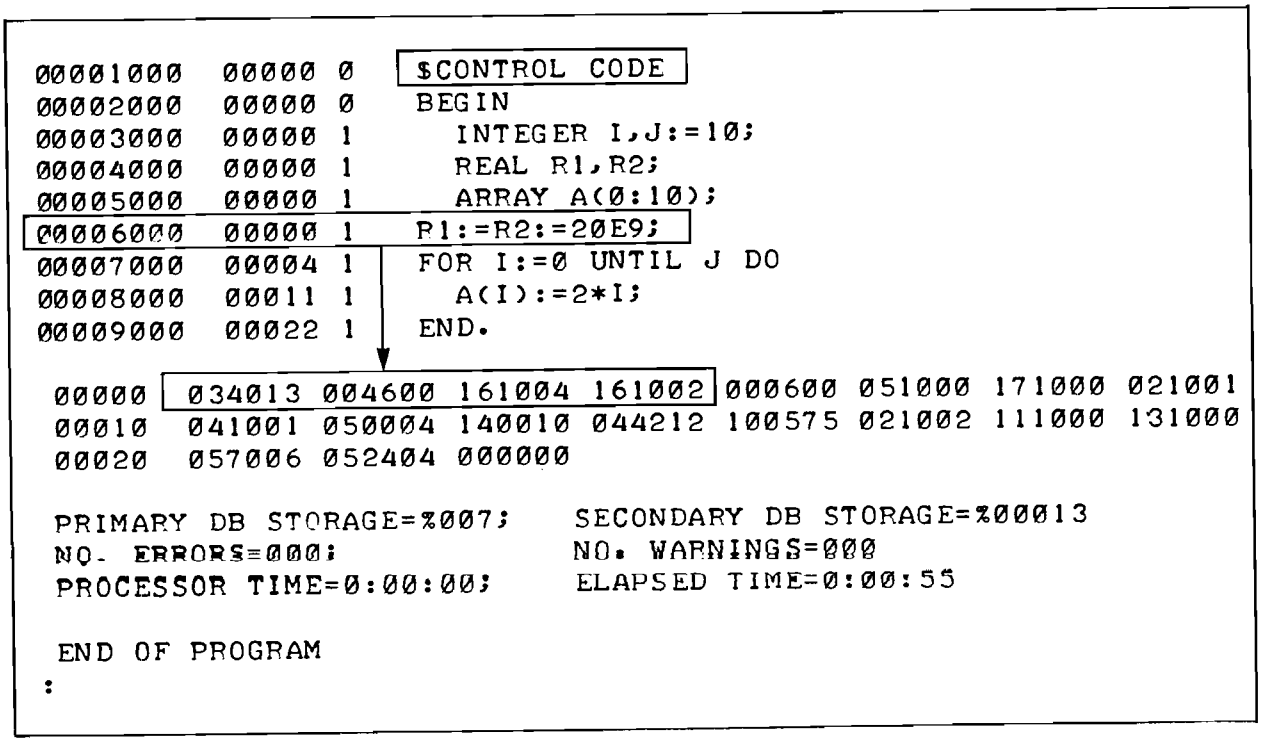


Figure 9-2. $CONTROL CODE Output

subparameter *nnnn* is an integer ranging from 10 to 9999. The LINES=*nnnn* parameter remains in effect until another LINES=*nnnn* parameter appears. If this parameter is omitted, the default value assigned is:

60 lines per page for devices other than terminals.
32767 lines per page for terminals.

ERRORS=*nnn*
Sets the maximum number of severe errors allowed during compilation to *nnn*; if this limit is exceeded, compilation terminates and the *uslfile* is unchanged. If the limit specified has already been exceeded when the ERRORS=*nnn* parameter is encountered, compilation terminates. If the ERRORS=*nnn* parameter is omitted, *nnn* is set to 100 by default.

USLINIT
Initializes the *uslfile* to empty status prior to generation of object code. If you do not specify a *uslfile* or if you specify a uslfile whose contents are obviously incorrect, the compiler automatically initializes the *uslfile* to empty status whether or not USLINIT is specified.

DEFINE
Causes the bodies of DEFINEs to be written out to a disc file, thereby increasing the amount of symbol table space available to the compiler. The $CONTROL option must be invoked before any DEFINEs are declared.

```
00001000  00000 0   $CONTROL ADR
00002000  00000 0   BEGIN
00003000  00000 1     INTEGER I,J:=10;
                            DB+000 ---- I
                            DB+001 ---- J
00004000  00000 1     REAL R1,R2;
                            DB+002
                            DB+004
00005000  00000 1     ARRAY A(0:10);
                            DB+006
00006000  00000 1   R1:=R2:=20E9;
00007000  00004 1   FOR I:=0 UNTIL J DO
00008000  00011 1     A(I):=2*I;
00009000  00022 1   END.
 PRIMARY DB STORAGE=%007;    SECONDARY DB STORAGE=%00013
 NO. ERRORS=000;             NO. WARNINGS=000
 PROCESSOR TIME=0:00:00;     ELAPSED TIME=0:01:05

 END OF PROGRAM
:
```

Figure 9-3. $CONTROL ADR Output

SEGMENT=*segname*
Starts a new segment with the specified *segname*. The *segname* can consist of up to 15 alphanumeric characters starting with an alphabetic character. Apostrophes are allowed within the *segname* except as the first character. The *segname* stays in effect until explicitly overridden by another $CONTROL SEGMENT or compilation terminates. For a main-body which is to be in a segment by itself, the $CONTROL SEGMENT should be placed after the procedure and intrinsic declarations and before the global subroutines and main-body. See figure 1-2 for a sample program using this parameter.

ADR
After each declaration, a record is sent to the *listfile* if LIST is in effect showing the addressing mode and displacement of the declared variables. This option is turned off by NOLIST. Figure 9-3 shows a sample compilation with ADR specified.

INNERLIST
After each statement line, the mnemonics for unoptimized code generated by the compiler are sent to the *listfile* if LIST is in effect. In addition to the mnemonic, the octal value and approximate execution time in microseconds of each instruction are shown. This option is turned off by NOLIST. Figure 9-4 shows a sample INNERLIST output. *NOTE:* Some address and constant initialization is resolved in later passes of the compiler and segmenter, so the machine code displayed does not always reflect the exact machine code executed. (The times shown on the listing are sample times only and are not accurate for any specific HP3000 model.)

MAIN=*program-name*
Assigns the specified *program-name* to the main program. The format for program names is the same as for segment names. Starting with page 2, the *program-name* is listed in columns 13-27 of the heading.

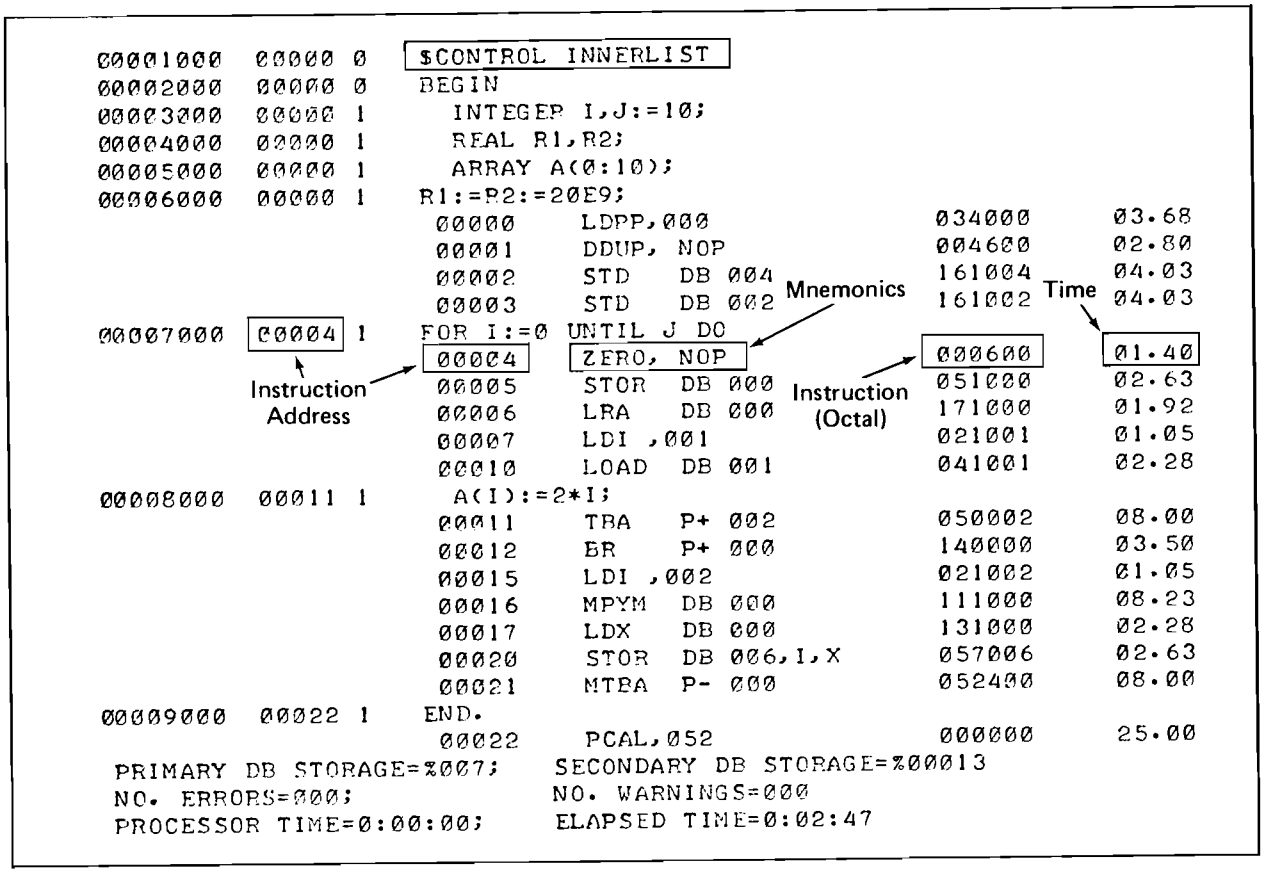


Figure 9-4. $CONTROL INNERLIST Output

UNCALLABLE
Makes the outer block entry point uncallable except by code running in privileged mode. If used, this parameter must be specified at the beginning of the source file.

PRIVILEGED
Makes the code segment containing the outer block privileged. If used, this parameter must be specified before the first BEGIN. *NOTE:* Hewlett-Packard cannot be responsible for system integrity when programs written by users operate in privileged mode.

SUBPROGRAM [*(procedure-name*[*] [*,...,procedure-name*[*]])]
Places the compiler in subprogram mode. If used, this parameter must be specified at the beginning of the program. If no parameters are specified, all of the procedures in the merged source program are compiled, but the outer block or main program if present is not compiled.

If procedure parameters appear, only those procedures specified are compiled. All others are skipped. In addition, procedure-names which are followed by an asterisk (*) are compiled with LIST, CODE, and MAP options on. Those without an * are compiled but not listed. The asterisk mechanism is overridden by explicit CONTROL commands specifying LIST, ADR, etc.

The default mode for compilation is program mode.

Even in subprogram mode, global declarations and OPTION FORWARD and OPTION EXTERNAL procedure declarations must be included in the source file, if they are to be referenced by the procedures being compiled. The compiler includes these items in its symbol table, but does not allocate

any space. All INTERNAL procedures and secondary entry points should be declared OPTION FORWARD.

Compiler commands are recognized at any point in the source file. For segmented programs, the segmentation scheme should be preserved in the subprogram mode. The compiler gives procedures the last segment name declared and links each procedure to all other procedures in the same USL file which have the same segment name, even those resulting from a previous compilation. The compiler also automatically CEASEs any existing procedures in the file with the same *procedure-name* as the one currently being compiled, except for INTERNAL procedures. See the *MPE Segmenter Subsystem Reference Manual* for a discussion of CEASE.

EXAMPLES:

```
$CONTROL SUBPROGRAM
$CONTROL SUBPROGRAM(PROC1,PROC2*)
```

The default parameters of $CONTROL are:

LIST
WARN
NOMAP
ERRORS= 100
NOCODE
SEGMENT= SEG'
MAIN= OB'
program mode
ADR off
INNERLIST off
LINES= 60 (except for terminals)
USL file not initialized
CALLABLE, non-privileged outer block.

The following $CONTROL command requests unrestricted access to the *listfile*, listing of all source text, symbol table information, and object code, suppression of warning messages but not of error messages. By default, the maximum number of lines per printed page is limited to 60, the maximum number of errors allowed is 100, the *uslfile* is not initialized to empty status, and SPL is in program mode.

```
$CONTROL LIST,SOURCE,MAP,CODE,NOWARN
```

The following $CONTROL command illustrates the default values for the command parameters. It produces the same effect as if no $CONTROL command were entered:

```
$CONTROL LIST,SOURCE,WARN,NOMAP,NOCODE,LINES= 60,ERRORS= 100
```

9-3. $IF COMMAND (CONDITIONAL COMPILATION)

Generally, when you submit a program to the compiler, you want the entire program compiled. However, occasionally, you may only want to have a portion of the program compiled. You can request such conditional compilation by delimiting the source code to be compiled (or omitted) with a series of $IF compiler commands. These $IF commands, interrogate any of ten switches, X0 through X9, inclusive. You can set these switches by using the $SET command described in paragraph 9-4. When the condition specified in the $IF command is true, all source records are compiled until the next $IF command is encountered which is false. When the condition specified is false, all source records are omitted until a $IF command which is true is executed. However, $EDIT, $PAGE, and $TITLE commands are never ignored.

The form of a $IF command is:

$[$]IF [Xn= {OFF / ON}]

EXAMPLES:

```
$IF X0=ON
$IF
$$IF X9=OFF
```

where

n
specifies which switch is to be tested. It is any digit between 0 and 9 inclusive.

Spaces are not allowed between the X and the digit *n*.

A $IF command can appear anywhere in the source text. The appearance of a $IF command always terminates the influence of any preceding $IF command. When a $IF command is entered without a parameter, it has the same effect as an $IF command whose condition is true. That is, the text following the command is compiled and any previous $IF command is cancelled.

The source text is listed regardless of whether or not it is compiled if the $CONTROL command LIST and SOURCE options are in effect.

The *textfile-masterfile* merging operation and transmission of merged/edited text to the *newfile* are not affected by $IF commands. Merging and editing are described in the discussion of the $EDIT command.

An example illustrating the use of the $IF command is presented together with the $SET command discussion below.

9-4. $SET COMMAND (SOFTWARE SWITCHES FOR CONDITIONAL COMPILATION)

When the compiler is first called, all ten switches (X0-X9) are turned off. You can turn them on and off again with the $SET command.

The form of the $SET command is:

$[$]SET [Xn= {OFF / ON} [,Xn= {OFF / ON}] ...]

EXAMPLES:

```
$SET X0=OFF,X1=ON
$SET
$SET X3=ON
```

where

n
indicates which switch is to be set. It can be any digit between 0 and 9 inclusive.

A $SET command can appear anywhere in the source text. If a $SET command is encountered which does not have a parameter list, all ten switches are turned off.

In the following source text, switches X4 and X5 are set on and interrogated with the results indicated by the comments:

```
          .
          .
          .
$SET X4=ON, X5=ON     <<SET SWITCHES X4 AND X5 ON>>
          .
          .
          .
$IF X5=ON             <<REQUESTS COMPILATION OF SOURCE BLOCK 1>>
          .
  (SOURCE BLOCK 1)
          .
          .
$IF X5=OFF            <<REQUESTS THAT SOURCE BLOCK 2 BE IGNORED>>&
$                     <<BY CANCELLING PREVIOUS $IF COMMAND>>
          .
          .
  (SOURCE BLOCK 2)
          .
          .
          .
$IF                   <<CANCELS PREVIOUS $IF COMMAND SO THAT>>&
$                     <<SOURCE BLOCK 3 IS COMPILED>>
          .
          .
  (SOURCE BLOCK 3)
```

9-5. $TITLE COMMAND (PAGE TITLE IN STANDARD LISTING)

On each page of output listed during compilation, a standard heading appears. Positions 29 through 132 of this heading are reserved for a title, usually describing the page content, optionally specified with the $TITLE command.

The form of the $TITLE command is:

```
$[$]TITLE [string [,string]...]
```

EXAMPLES:

```
$TITLE "FILE CREATE PROGRAM"
$TITLE
$$TITLE "UPDATE MASTER DATA FILE",&
$       "AND PRINT REPORTS"
```

Each string parameter is a character string bounded by quotation marks that is combined with any other strings specified to form the title. In forming the title, the strings are stripped of their delimiting quotation marks and they are then concatenated left-to-right. The entire parameter list can specify up to 104 characters, including spaces within the strings but excluding delimiters and spaces between the strings. If the title contains fewer than 104 characters, the unused portion is filled to the right with spaces. If no string parameters are present in the $TITLE command, or if no $TITLE command or $PAGE command with a title specification is entered, the title portion of the heading is blank. When a new $TITLE command is encountered, it supersedes any previously specified title from that point on.

When a $TITLE command is interpreted and the NOLIST parameter of the $CONTROL command is in effect, title specification or replacement occurs even when the $TITLE command appears within the range of an $IF command whose relation is evaluated as false.

9-6. $PAGE COMMAND (PAGE TITLE AND EJECTION)

You can specify a program title (as with the $TITLE command) together with page ejection by entering the $PAGE command. This allows varied listing formats. For example, individual sections of the program can be listed starting on a new page, and each section can have its own descriptive title.

The form of the $PAGE command is:

```
$[$]PAGE [string[,string]...]
```

EXAMPLES:

```
$PAGE "FILE OPEN SECTION"
$PAGE
$$PAGE "READ RECORD SECTION"
$PAGE "VERIFY INPUT DATA",&
$       "AND UPDATE DATA BASE"
```

Each string parameter has the same format, meaning, result, and constraints as in the $TITLE command. If no parameter is specified in the $PAGE command, the previous title, if any, remains in effect.

If the LIST parameter of the $CONTROL command is in effect when a $PAGE command is encountered, the following steps take place:

1. A page eject is generated.

2. The standard page heading including the new title, if one is specified, is printed followed by two blank lines.

If a new title is not specified, the standard heading with the old title is printed followed by two blank lines.

If the LIST parameter is not in effect, the new title replaces any previous title, but no printing or page ejecting occurs. The new title appears when LIST is put into effect.

The $PAGE command itself is never listed.

9-7. $EDIT COMMAND (SOURCE TEXT MERGING AND EDITING)

You can request the following merging and editing operations:

- Merge corrections or additional source text on *textfile* with an existing source program and commands on *masterfile* to produce a new source program and commands. This new input is compiled and optionally copied to *newfile*, which can be saved for recycling through an MPE :FILE command.

- Check source-record sequence numbers for ascending order.

- Omit sections of the old source program during merging.

- Re-number the sequence fields of the records in the new, merged source program.

The editing done by the compiler is limited to linear source text modification. Extensive or more sophisticated editing is possible with the HP 3000 text editor, EDIT/3000.

9-8. MERGING

You can specify merging simply by using actual file names for the *textfile, masterfile,* and (optionally) *newfile* parameters of the MPE :SPL command when the compiler is called. A sample merging operation is shown below; however, for a complete description of the :SPL command see paragraph 10-11.

To specify merging of a *textfile* TFILE with a *masterfile* MFILE, you could enter the following :SPL command:

```
:SPL TFILE,,,MFILE,NFILE
```

The merged source text is copied to the *newfile* NFILE, with the object code and listing output written to the default files $NEWPASS and $STDLIST respectively.

Prior to merging, the records in both *textfile* and *masterfile* must be arranged in ascending order according to the value of the sequence field on any record, or the sequence fields must be blank. The order of sequencing is based on the ASCII Collating Sequence as shown in Appendix A. There are no restrictions regarding blank sequence fields; the sequence fields of some or all of the records in either the *textfile* or *masterfile*, or both files, can be blank, and such records can appear anywhere in either file.

The merging operation is also based on ascending order of sequence fields according to the ASCII Collating Sequence. During merging, the sequence fields of the records in both files are checked for ascending order. If their order is improper, the offending records are skipped during merging and appropriate diagnostic messages are sent to the *listfile*. During each comparison step in merging, one record is read from each file and these records are compared with one of three results:

1. If the values of the sequence fields of the *masterfile* and the *textfile* are equal, then the *textfile* record is compiled and, optionally, passed to the *newfile*; the *masterfile* record is ignored; and one more record is read from each file for the next comparison.

2. If the value of the sequence field of the *masterfile* record is less than that of the *textfile* record, the *masterfile* record is compiled and, optionally, passed to the *newfile*; the *textfile* record is retained for comparison with the next *masterfile* record; and the next *masterfile* record is read.

3. If the value of the sequence field of the *textfile* record is less than that of the *masterfile* record, the *textfile* record is compiled and, optionally, passed to the *newfile*; the *masterfile* record is retained for comparison with the next *textfile* record; and the next *textfile* record is read.

During merging, a record with a blank sequence field is assumed to have the same sequence field as that of the last record with a non-blank sequence field read from the same file, or as a null sequence field, if no record with a non-blank sequence field has yet been encountered in the file. Thus, a group of one or more records with blank sequence fields residing on the *masterfile* are never replaced by records from the *textfile*; they can only be deleted through use of the $EDIT command as explained below.

Records from the *masterfile* that are replaced during merging and thus neither compiled nor sent to the *newfile* are not listed during compilation.

When an end-of-file condition is encountered on either the *textfile* or the *masterfile*, merging terminates, except for the continuing influence of an unterminated VOID parameter in an $EDIT command, as discussed later. At this point, the subsequent records on the remaining file are checked for proper sequence, compiled, and, optionally, passed to the *newfile*. However, *masterfile* records within the range of a VOID parameter are neither compiled nor sent to the *newfile*.

The sequence field values of records transmitted to the *newfile* are not normally changed by the merging operation. However, you can request the assignment of new sequence characters by using the $EDIT command.

9-9. CHECKING SEQUENCE FIELDS

The presence of a *masterfile* during compilation implicitly requests the checking of source records for proper sequence. Thus, when you specify both a *textfile* and a *masterfile* as input files for the compiler, or when you specify a *masterfile* alone, sequence-checking is done on both files. But when you specify a *textfile* as the only input file, sequence checking is not performed. Therefore, when you want to have

your input sequence-checked without merging two input files, you can read the input from either the *textfile* or the *masterfile* and use $NULL for the other file. For example,

```
:SPL SOURCE,,$NULL
```

9-10. EDITING

Editing operations during merging consist of omitting sections of the old source program residing on the *masterfile* and/or renumbering the sequence fields of the new, merged source program residing on the *newfile*. Both of these operations are requested through the $EDIT command.

The form of the $EDIT command is:

$[$]EDIT [*parameter* [,*parameter*]...]

EXAMPLES:

```
$EDIT SEQNUM=50,INC=10
$EDIT VOID=100
$EDIT NOSEQ
```

where

parameter
specifies an option of the $EDIT command. The parameter is one of the following: VOID=*sequence-value,* SEQNUM=*sequence-number,* NOSEQ, or INC=*incnumber.*

The parameters are discussed individually below. The parameters can be specified in any order.

VOID=*sequence-value*
Requests the compiler to bypass during merging all records on the *masterfile* whose sequence fields contain a value less than or equal to the *sequence-value*, plus any subsequent records with blank sequence fields. This parameter remains in effect until a *masterfile* record with a sequence field value higher than the *sequence-value* is encountered. The VOID parameter is initially disabled when the compiler is invoked. The *sequence-value* is either a legal sequence number of from one to eight digits or a character string. If the *sequence-value* is less than eight characters, SPL left-fills with ASCII zeros and sequence character strings with spaces. *NOTE:* $EDIT VOID in $INCLUDE files must reference lines in the INCLUDEd file only.

SEQNUM=*sequence-number*
Requests re-numbering of the merged source records on the *newfile*, beginning with the value specified by the *sequence-number*. This value replaces the *sequence-number* of the next record sent to the *newfile*. The *sequence-number* of each succeeding record is incremented according to the value specified by the INC parameter or its default as described below. If the SEQNUM=*sequence-number* parameter is present but a *newfile* does not exist, the re-numbering request is ignored. If this parameter is present and the *newfile* exists, the re-numbering request remains in effect until an $EDIT command with the NOSEQ parameter is encountered. When the merged output is listed, records actually transmitted to the newfile appear with blank sequence fields. The re-sequencing request is initially disabled when the compiler is called. The *sequence-number* is a legal *sequence-number* of from one to eight digits. If less than eight digits, the SPL compiler left-fills with ASCII zeros.

NOSEQ
Suspend re-numbering of merged records on the *newfile*; the current sequence numbers are retained. If neither SEQNUM nor NOSEQ are specified, NOSEQ takes effect by default until superseded by SEQNUM.

INC=*incnumber*
Sets the increment by which records sent to the *newfile* are renumbered if SEQNUM is in effect. The increment is specified by *incnumber*, which is a value ranging from 1 through 99999999. Notice, however, that very large increments are of limited value since they may cause the eight-digit sequence-number to overflow. Re-numbering only occurs if SEQNUM is specified or the last parameter is not overridden by a NOSEQ parameter, and a *newfile* exists. If SEQNUM is specified but INC is not, the *sequence-number* is incremented by the default value of 1000 for each succeeding record. This default value applies until an INC parameter specifying a new value is encountered.

$EDIT commands are normally input from the *textfile*. You can input them from the *masterfile*, but this procedure is not recommended since any $EDIT command containing a VOID parameter on the *masterfile* could void its own continuation records. $EDIT commands themselves are never sent to the *newfile*; thus, the $$EDIT... form of the command, while permitted, is redundant.

While sequence fields are allowed, and usually necessary, on records containing $EDIT commands, continuation records for such commands should have blank sequence fields.

During merging, a group of one or more *masterfile* records with blank sequence fields are never replaced by lines from the *textfile*; they can only be deleted by an $EDIT command with a VOID=*sequence-value* parameter at least as great as the last non-blank sequence field preceding the group. In this case, the entire group of *masterfile* records with blank sequence number fields is deleted.

Since voided records are never passed to the *uslfile* or *newfile*, their sequence is never checked, and they never generate an out-of-sequence diagnostic message.

A VOID parameter does not affect records in the *textfile*.

Any *masterfile* record replaced by a *textfile* record is treated as if voided, except that following records with blank sequence fields are not also voided. If a replaced record would have been out-of-sequence, the *textfile* record that replaces it produces an out-of-sequence diagnostic message.

In general, whenever a record sent to the *newfile* has a non-blank sequence field lower in value than that of the last record with a non-blank sequence field, a diagnostic message is printed.

For example, suppose you want to merge text input from the standard input device (default for *textfile* is $STDIN) with an old program on the file OLDPROG, creating new source input on the file NEWPROG and you want to re-number the merged source records on NEWPROG beginning with the value 50, incrementing the sequence number of each subsequent record by 10. After logging on, you would enter:

```
:SPL ,,,OLDPROG,NEWPROG
    .
    .
    .
$EDIT SEQNUM=50,INC=10
    .
```

(New text or corrections to be merged with old program.)

.
.
.

9-11. $SPLIT/$NOSPLIT COMMANDS

The $SPLIT and $NOSPLIT commands are intended for privileged users in split-stack mode to delimit an area of code to be checked for split-stack errors (see section 8-1). These commands perform the same function as OPTION SPLIT. However, OPTION SPLIT is effective for an entire procedure, while $NOSPLIT can be used to reset $SPLIT. (Please see OPTION SPLIT, 7-13A.)

9-12. $COPYRIGHT COMMAND

You can specify copyright information which is transmitted to the USL and program files by using the $COPYRIGHT command.

The form of the COPYRIGHT command is:

$[$]COPYRIGHT *string*[[,*string*]...]

EXAMPLE:

```
$COPYRIGHT "(C) Copyright Hewlett-Packard Company 1976." ,&
$          "All rights reserved. No part of this program may be",&
$          "photocopied, reproduced, or transmitted without",&
$          "prior written consent of Hewlett-Packard Company."
```

Each string parameter is a character string bounded by quotation marks that is combined with any other strings specified to form the copyright information copied to the USL and program files. The $COPYRIGHT command must precede the outer block BEGIN. The maximum number of characters is 510.

9-13. CROSS REFERENCE LISTING

To obtain a cross reference listing of the identifiers used in an SPL program, run the CROSSREF program.* Use file equations for the formal designators LIST and TEXT for the list file and text file respectively. Figure 9-5 shows a sample CROSSREF output. The listing shows, for each identifier, the sequence number of each record in the source program in which the identifier occurs.

*The CROSSREF program is available through the HP 3000 Contributed Library package offered by HP Computer Systems Division. Contact your local HP Sales Office for more information.

```
:FILE LIST=$STDLIST
:FILE TEXT=SPLEX
:RUN CROSSREF.PUB.SYS

          S.P.L. CROSS REFERENCE TABLE--- AUG  9, 1974 VERSION

                              SPLEX.PUB.GNOMON
                          MON, JAN 26, 1976,  3:26 PM

NUMBER OF CARD IMAGES=9. NUMBER OF SYMBOLS=5. NUMBER OF REFERENCES=7.

A (ARRAY)
          00005000  00008000

I (INTEGER),
          00003000  00007000  00008000  00008000

J (INTEGER)
          00003000  00007000

R1 (REAL)
          00004000  00006000

R2 (REAL)
          00004000  00006000
```

Figure 9-5. Cross Reference Listing

9-14. $INCLUDE COMMAND

The $INCLUDE command permits inclusion of text from another file into the SPL source file.

The form of the $INCLUDE command is:

$INCLUDE *filename;*

EXAMPLE:

$INCLUDE Myfile;

where

filename
is the fully qualified name of the file to be included. The Included file may contain other $INCLUDEs to a maximum of 10. INCLUDE files are treated as unnumbered files; $EDIT VOID in Included files must reference lines in the INCLUDEd file only.

MPE COMMANDS

SECTION X

10-1. MPE COMMANDS

Communication with the MPE Operating System is initiated through commands. Commands are requests issuesd to MPE to perform various functions external to an SPL source program. For example, commands are used to initiate and terminate batch jobs and interactive sessions, compile and execute source programs, call various MPE subsystems, and obtain job/session status information. Commands can be entered through any standard input file such as a card reader file or a terminal file. Commands which you will use most often with SPL programs are summarized in table 10-1. A complete description of all MPE commands is in the *MPE Commands Reference Manual.*.

Table 10-1. MPE Commands

COMMAND	FUNCTION
:JOB	Initiates a batch job
:HELLO	Initiates an interactive session
:FILE	Specifies characteristics of a file
:BUILD	Creates a new file
:PURGE	Deletes a file from the system
:CONTINUE	Disregards batch job error condition
:SPL	Compiles an SPL source program
:SPLPREP	Compiles and prepares an SPL source program
:SPLGO	Compiles, prepares, and executes an SPL source program
:PREP	Prepares a compiled program
:PREPRUN	Prepares and executes a compiled program
:RUN	Executes a prepared program
:EOD	Signifies the end of data
:EOJ	Terminates a job
:BYE	Terminates a session

In general, the form of of an MPE command is:

:*command* [*parameter-list*]

In interactive mode, the colon is prompted by MPE; however, in batch mode, you must provide the colon in column 1 of the command record.

The *parameter-list* can contain zero, one, or more parameters that specify files, values, and options for the command. The end of each parameter in a list is signified by a delimiter. A delimiter is a character that separates one item from another. Delimiters consist of commas, semicolons, equal signs, or other punctuation marks.

A space must separate the *command* from the *parameter-list*; however, the *command* must immediately follow the colon without any intervening spaces.

The meanings of parameters in some commands are determined by their positions in the *parameter-list*. For example, in an :SPL command:

:SPL *textfile,uslfile,listfile,masterfile,newfile*

the parameters are positional and their positions in the list designate their meanings. The omission of an optional positional parameter from a *parameter-list* is signified by adjacent delimiters, as shown below:

:SPL *textfile,,listfile*

When parameters are omitted from the end of a list, no adjacent delimiters are required as shown in the example by the omission of *masterfile* and *newfile*.

10-2. SPECIFYING FILES FOR PROGRAMS

Both the SPL compiler and the MPE Operating System read input from and write output to files handled through the MPE file facility. For example, the compiler reads source code from a *textfile*, writes object code to an object file (*uslfile*), produces listings to a *listfile*, and performs editing and merging operations using an old *masterfile* for input and a *newfile* for output. Each file has a formal file designator. You are responsible for equating actual file designators to these formal file designators in one of three ways.

1. By naming the files as positional parameters in the MPE commands to compile, prepare, and execute.

2. By omitting optional parameters from the MPE compilation, preparation, or execution command, thus allowing default file designators to be in effect.

3. By using MPE :FILE commands to equate the formal file designators to the actual file designators. If you use this method, you must call the compiler with the MPE :RUN command using a PARM= parameter signifying which files are present, as described later. This method can only be used for compilation and not for preparation or execution.

You can also use MPE :FILE commands to equate the formal file designators for your execution-time files to actual file designators. See the *MPE Commands Reference Manual* for a complete description of the :FILE command.

10-3. SPECIFYING FILES AS COMMAND PARAMETERS

You can name the following types of files as parameters in a compilation, preparation, or execution command:

- System Defined Files
- User Pre-defined Files
- New Files
- Old Files

10-4. SYSTEM-DEFINED FILES. System-defined file designators indicate those files that MPE uniquely identifies as standard input/output files for a job/session. These files are shown in table 10-2.

10-5. USER PRE-DEFINED FILES. A user pre-defined file is any file that was previously defined or redefined in a :FILE command. In other words, it is a back-reference to that :FILE command. In compilation, preparation, or execution commands, the actual file designator of this type of file is the formal file designator preceded by an asterisk to indicate that it was previously defined. For example,

```
:FILE S=MYTEXT
:FILE LP;DEV=LP
:SPL *S,,*LP
```

Table 10-2. System-Defined Files

ACTUAL FILE DESIGNATOR	DEVICE/FILE REFERENCED
$STDIN	A filename indicating the standard job or session input file (from which the job or session is initiated). For a job, this is typically a card reader; for a session this typically indicates a terminal. Input data records in the $STDIN file should not contain a colon in position one, since this indicates the end of the source input. Use the :EOD command to indicate the physical end of a source program. (The same command is used to indicate the end of a data file.)
$STDINX	Equivalent to $STDIN, except that MPE/3000 command records (those with a colon in position one) encountered in a data file are read without indicating the end of data. (However, the commands :JOB, :DATA, :EOJ, and :EOD are exceptions that always indicate the end of data and are never read as data.)
$STDLIST	A filename indicating the standard job or session listing file corresponding to the particular job or session input device being used. (For each potential job/session input device, a user with MPE/3000 System Supervisor capability designates a corresponding job/session listing device during system configuration.) The job or session listing device is customarily a printer for a batch job and a terminal for a session.
$NULL	The name of a non-existent "ghost" file that is always treated as an empty file. When referenced as an input file by a program, that program receives only an end of data indication upon first access. When referenced as an output file, the associated write request is accepted by MPE/3000 but no physical output is actually performed. Thus, $NULL can be used to discard unneeded output from an executing program.

10-6. NEW FILES. New files are files that have not yet been created, and are being created for the first time by the current batch job or interactive session. New files can have actual file designators as shown in table 10-3.

Table 10-3. New Files

FILE	PURPOSE	FORMAL FILE DESIGNATOR	DEFAULT FILE DESIGNATOR
Textfile	Contains source program, correction text to be merged, and/or compiler subsystem commands.	SPLTEXT	$STDIN
Listfile	Destination of listing output.	SPLLIST	$STDLIST
Uslfile	Destination of object program code.	SPLUSL	$NEWPASS
Masterfile	Old source program to be merged and edited with new text input from *textfile*.	SPLMAST	$NULL
Newfile	New source program resulting from (optional) merging of *textfile* and *masterfile*.	SPLNEW	$NULL
Progfile	Destination of executable object program.	None	$NEWPASS

10-7. OLD FILES. Old files are existing files in the system. They may be named by the designators shown in table 10-4.

Table 10-4. Old Files

ACTUAL FILE DESIGNATOR	FILE REFERENCED
$OLDPASS	The name of the temporary file last closed as $NEWPASS.
filereference	Any other old file to which you have access. It may be a job/session temporary file created in the current or a previous program in the current job/session, or a permanent file saved by any program in any job/session. The format is the same as *filereference*, noted in table 10-5.

10-8. INPUT/OUTPUT SETS. All of the preceding actual file designators can be classified as those used as input parameters (input set) and those used as output parameters (output set). These sets are defined as follows:

INPUT SET	
$STDIN	The job/session input file.
$STDINX	The job/session input file with commands allowed.
$OLDPASS	The last file passed.
$NULL	A constantly-empty file that will produce an end-of-file condition whenever it is read.
**formaldesignator*	A back-reference to a previously defined file.
filereference	A file name, and perhaps account and group names and a lockword.

OUTPUT SET	
$STDLIST	The job/session listing file.
$OLDPASS	The last file passed.
$NEWPASS	A new temporary file to be passed.
$NULL	A constantly-empty file.
**formaldesignator*	A back-reference to a previously defined file.
filereference	A file name, and perhaps account and group names and a lockword.

10-9. SPECIFYING FILES BY DEFAULT

When you omit an optional file parameter from a compilation, preparation, or execution command, MPE assigns one of the members of the input or output sets by default. The file designator assigned depends on the specific command, parameter, and operating mode as noted later in this section. The default file designators are shown in table 10-5.

Table 10-5. SPL Compiler File Designators

ACTUAL FILE DESIGNATOR	FILE REFERENCED
$NEWPASS	A temporary disc file that can be passed automatically to any succeeding MPE/3000 command within the same job or session which references it by the filename $OLDPASS. (Passing is explained in the compilation, preparation, and execution command examples.) Only one such file can exist in the job or session at any one time. (When $NEWPASS is closed, its name is changed to $OLDPASS automatically, and any previous file named $OLDPASS is deleted.)
filereference	Any other new file to which you have access. Unless you specify otherwise, this is a temporary file, residing on disc, that is destroyed upon termination of the program. If no :FILE command specifies otherwise, any such SPL files are closed as job/session temporary files, saved until the end of the job/session, and then are purged. If closed as permanent files (by specifying SAVE in a :FILE command), they are saved until you purge them. Typically, this format consists of a file name containing up to eight alphanumeric characters, beginning with a letter. In addition, other elements (such as a group name, account name, or lockword) can be specified. The complete rules governing the *filereference* format are contained in the *MPE Commands Reference Manual*.

10-10. COMPILING, PREPARING, AND EXECUTING SPL SOURCE PROGRAMS

The commands used for compilation, preparation, and execution of SPL source programs are:

:SPL or :RUN SPL.PUB.SYS	Compiles a source program.

:SPLPREP	Compiles and prepares a source program.
:SPLGO	Compiles, prepares, and executes a source program.
:PREP	Prepares source programs which have been compiled into a USL file.
:RUN	Executes programs that have been compiled and prepared (and therefore exist on program files).
:PREPRUN	Prepares and executes programs compiled into USL files.

10-11. :SPL COMMAND

The :SPL command compiles an SPL source program.

The form of an :SPL command is:

:SPL [*textfile*] [,[*uslfile*] [,[*listfile*] [,[*masterfile*] [,[*newfile*]]]]] [;INFO = *quoted string*]

EXAMPLES:

```
:SPL MYSOURCE,,LIST
:SPL
:SPL MYSOURCE,USL,*LP,MASTER,NEWMAST
```

where

textfile
is the name of an input file from which the source program is to be read. If omitted, the program will be read from the standard input file $STDIN. Do not use the designator SPLTEXT for this parameter.

uslfile
is the name of the USL (User Subprogram Library) file on which the object program is to be written. If this parameter is included in an :SPL command, it must indicate a file previously created in one of two ways:

1. By saving a USL file with a :SAVE command from a previous compilation.

2. By creating a new file with a :BUILD command and designating it as a USL file with a file code of 1024 or USL. For example,

   :BUILD MYUSL;CODE= 1024 or :BUILD MYUSL;CODE= USL

If the *uslfile* is omitted, the default file $OLDPASS is used. Do not use the designator SPLUSL for this parameter.

listfile
is the name of the file to which the program listing is to be sent. If omitted, the default file $STDLIST is assigned. Typically $STDLIST is the terminal in a session or the line printer in batch. Do not use the designator, SPLLIST for this parameter.

masterfile
is the name of a file to be optionally merged with *textfile* and written onto a file named *newfile*. If *masterfile* is omitted, no merging takes place. Do not use the designator SPLMAST for this parameter.

newfile
is the name of a file on which the re-sequenced records from the *textfile* and the *masterfile* are optionally merged. When *newfile* is omitted, no *newfile* is created. Do not use the designator SPLNEW for this parameter.

All parameters of an :SPL command are optional. However, direct interactive input is not recommended since it is impossible to correct an error after pressing the carriage return key. To create source files, use the HP 3000 Text Editor (See the *EDIT/3000 Reference Manual*).

quoted string
is a list of compiler commands enclosed in single or double quotes in the format described in section 9-1.

INFO = parameter

The INFO keyword on the SPL, SPLPREP, and SPLGO commands allows compiler commands to be added to a program without changing the source. These commands logically precede any other source. On the listing, these commands have a sequence field of INFO= to indicate their source as illustrated in the example below. These compiler commands read from the quoted string are not sent to *newfile*.

```
:SPL EXAMPLE;INFO="$CONTROL MAP$CONTROL INNERLIST"

PAGE 0001    HP32100A.08.02 [4W] (C) HEWLETT-PACKARD COMPANY 1982

  IN FO=   00000 0    $CONTROL MAP
  IN FO=   00000 0    $CONTROL INNERLIST
    1      00000 0    BEGIN
    2      00000 1    INTEGER I;
    3      00000 1
    4      00000 1    I := 99;
                        00000    LDI ,143              021143  01.05
                        00001    STOR  DB 000          051000  03.15
    5      00002 1    END.
                        00002    PCAL,052              000000  14.90

      IDENTIFIER            CLASS          TYPE       ADDRESS

       I                  SIMP. VAR.      INTEGER      DB+000
       TERMINATE`         PROCEDURE

PRIMARY DB STORAGE=%001;       SECONDARY DB STORAGE=%00000
NO. ERRORS=0000;               NO. WARNINGS=0000
PROCESSOR TIME=0:00:01;        ELAPSED TIME=0:00:05

END OF PROGRAM
```

10-12. RUN SPL.PUB.SYS COMMAND

An alternative way to call the SPL compiler is by using the :RUN command. Before using the :RUN command, you must use file equations for the files normally specified on the :SPL command. The formal file designators are:

SPLTEXT	*(textfile)*
SPLLIST	*(listfile)*
SPLUSL	*(uslfile)*
SPLMAST	*(masterfile)*
SPLNEW	*(newfile)*

Table 10-6. PARM Values

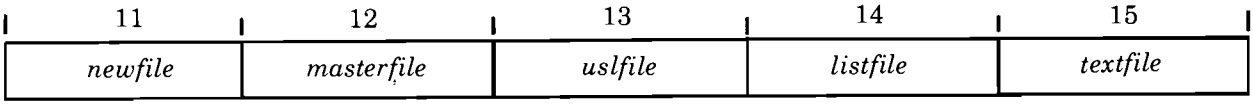
PARAMETERNUM	FILES PRESENT
0	None
1	*textfile*
2	*listfile*
3	*listfile, textfile*
4	*uslfile*
5	*uslfile, textfile*
6	*uslfile, listfile*
7	*uslfile, listfile, textfile*
8	*masterfile*
9	*masterfile, textfile*
10	*masterfile, listfile*
11	*masterfile, listfile, textfile*
12	*masterfile, uslfile*
13	*masterfile, uslfile, textfile*
14	*masterfile, uslfile, listfile*
15	*masterfile, uslfile, listfile, textfile*
16	*newfile*
17	*newfile, textfile*
18	*newfile, listfile*
19	*newfile, listfile, textfile*
20	*newfile, uslfile*
21	*newfile, uslfile, textfile*
22	*newfile, uslfile, listfile*
23	*newfile, uslfile, listfile, textfile*
24	*newfile, masterfile*
25	*newfile, masterfile, textfile*
26	*newfile, masterfile, listfile*
27	*newfile, masterfile, listfile, textfile*
28	*newfile, masterfile, uslfile*
29	*newfile, masterfile, uslfile, textfile*
30	*newfile, masterfile, uslfile, listfile*
31	*newfile, masterfile, uslfile, listfile, textfile*

Thus, to compile from the file MYSOURCE and send the listing to the line printer, you would use

```
:FILE SPLTEXT=MYSOURCE
:FILE SPLLIST;DEV=LP
```

before using the :RUN command.

Additionally, you must specify a PARM=*parameternum* parameter on the :RUN command to indicate which files are present unless the default values are used. The value is between 0 and 31 as shown in table 10-6. Basically, the low order five bits in *parameternum* represent the five files which can be specified as shown below:



For example, to invoke the compiler with the *textfile* and *listfile* present, you would use the command:

```
:RUN SPL.PUB.SYS;PARM=3;INFO="$CONTROL NOLIST"
```

10-13. ENTERING PROGRAM SOURCE INTERACTIVELY

If you do not specify a textfile when compiling in session mode, you must enter the program source from the terminal. You are prompted for each source line with a greater-than sign (>). Each program unit (procedure, subroutine, or main body) is compiled as it is completed. To exit from the compiler, enter :EOD in response to the prompt character >.

10-14. :SPLPREP COMMAND

The :SPLPREP command compiles and prepares an SPL source program.

The form of the :SPLPREP command is:

:SPLPREP [*textfile*] [,[*progfile*] [,[*listfile*] [,[*masterfile*] [,*newfile*]]]] [;INFO = *quoted string*]

EXAMPLES:

```
:SPLPREP MYSOURCE,MYPROG,*LP
:SPLPREP MYSOURCE,,,MAST
```

where

textfile, listfile, masterfile, newfile, quoted string
have the same meanings as described under the :SPL command.

progfile
is the name of the file on which the prepared program is written. If this parameter is included, it must reference a file created in one of two ways:

1. By using the :BUILD command with a filecode of 1029 or PROG. For example,

   :BUILD PROGF;CODE= 1029

   or

   :BUILD PROGF;CODE= PROG

2. By specifying a non-existent file in the parameter, in which case a temporary file of the correct size and type will be created. To save the file for future jobs/sessions, you must use the :SAVE command after preparation.

If the *progfile* parameter is omitted, the default file $NEWPASS is assigned. This file is renamed $OLDPASS upon completion.

All :SPLPREP parameters are optional.

10-15. :SPLGO COMMAND

The :SPLGO command compiles, prepares, and executes an SPL source program.

The form of the :SPLGO command is:

:SPLGO [*textfile*] [,[*listfile*] [,[*masterfile*] [,*newfile*]]] [;INFO = *quoted string*]

EXAMPLES:

```
:SPLGO MYSOURCE,*LP
:SPLGO MYSOURCE,,MAST
```

where

textfile, listfile, masterfile, newfile, quoted string
all have the same meaning as described under the :SPL command.

All :SPLGO parameters are optional.

10-16. :PREP COMMAND

The :PREP command prepares source programs that have been compiled into a USL file.

The form of the :PREP command is:

:PREP *uslfile, progfile* [;ZERODB] [;PMAP] [;MAXDATA=*segsize*] [;STACK=*stacksize*]
[;DL=*dlsize*] [;CAP=*caplist*] [;RL=*filename*]

EXAMPLES:

```
:PREP MYUSL,MYPROG;PMAP;MAXDATA=4096
:PREP $OLDPASS,PROGF
```

where

uslfile
is the name of the USL file onto which the program file has been compiled.

progfile
is the name of the program file onto which the prepared program is to be written. This file must be created in one of two ways:

1. By creating a new file with the :BUILD command using a filecode of 1029 or PROG, as follows:

   ```
   :BUILD PROGF;CODE=1029
   ```

   or

   ```
   :BUILD PROGF;CODE=PROG
   ```

2. By specifying a non-existent file in this parameter, in which case a temporary file of the correct size and type will be created. To save this file for future jobs/sessions, you must use the :SAVE command.

Both the *uslfile* and the *progfile* parameters are required in a :PREP command.

ZERODB
is a request to set the initially defined DL-DB and DB-Q (initial) areas of the stack to zero.

PMAP
is a request to list certain information about the prepared program.

segsize
specifies a maximum size for the stack area in words. The segmenter normally establishes this value, but you can use this value to override the Segmenter's estimate.

stacksize
When a process is created by the system, the user is allocated MAXDATA words of virtual memory, but only stacksize words in main memory. The main memory space is expanded as required. This parameter allows you to override the Segmenter estimate.

dlsize
the DL-DB area size to be initially assigned to the stack. If not specified, MPE will estimate the value for each program.

caplist
the capability-class attributes associated with your program. The default values are BA (batch access) and IA (interactive access).

filename
the name of a relocatable procedure library to be searched to satisfy external references during program preparation. If not specified, no library is searched.

10-17. :PREPRUN COMMAND

The :PREPRUN command prepares and executes programs that have been compiled into USL files.

The form of the :PREPRUN command is:

```
:PREPRUN uslfile [,entry-point] [;NOPRIV] [;PMAP] [;DEBUG]
        [;LMAP] [;ZERODB] [;MAXDATA=segsize]
        [;PARM=parameternum] [;STACK=stacksize] [;DL=dlsize]
        [;RL=filename] [;LIB=library] [;CAP=caplist]
        [;NOCB]

EXAMPLES:

:PREPRUN $OLDPASS;PMAP;DEBUG;LIB=P
:PREPRUN MYUSL
```

where

uslfile
is the name of the USL file on which the program has been compiled.

entry-point
specifies the *entry-point* where execution is to begin. If not specified, execution begins at the primary *entry-point*.

NOPRIV
is a request to place a privileged program in non-privileged mode. If not specified, a privileged program executes in privileged mode.

PMAP
is a request to list certain information about the prepared program.

DEBUG
is a request to set a breakpoint on the first executable instruction of the program for entering debug commands. Refer to the *MPE DEBUG/ STACK DUMP Reference Manual.*

LMAP
is a request to list certain information about the loaded program.

ZERODB
is a request to set the initially defined DL-DB and DB-Q (initial) areas to zero.

segsize
specifies the maximum stack area (Z– DL) size permitted, in words. This value is normally set by the Segmenter, but you can use this parameter to override the Segmenter estimate.

parameternum
is a value that can be passed to your program as a general parameter for control or other purposes. If not specified, a zero is passed.

stacksize
When a process is created by the system, the user is allocated MAXDATA words of virtual memory but only *stacksize* words in main memory. The main memory is expanded as required. This parameter allows you to override the Segmenter estimate. If not specified, the *stacksize* is determined by the Segmenter for each individual program.

dlsize
is the size of the DL-DB area to be initially assigned to the stack. If not specified, it is established by MPE.

filename
is the name of a relocatable procedure library to be searched to satisfy external references during program preparation. If not specified, no library is searched.

library
specifies the order in which segmented procedure libraries are to be searched to satisfy external references during segmentation. The *library* can be either G (Group first), P (Public group first), or S (System first). If not specified, the System library is searched first.

caplist
specifies the capability-class attributes associated with your program. If not specified, BA (Batch Access) and IA (Interactive Access) are used.

NOCB
Requests that the file system not use stack segment (PCBX) for its control blocks, even if sufficient space is available. This permits you to expand your stack (via the DLSIZE or ZSIZE intrinsics) to the maximum possible limit at a later time, but causes the File Management System to operate more slowly for this program.

NOTE

You should only use this parameter if the program absolutely requires the largest stack possible.

10-18. :RUN COMMAND

The :RUN command executes a program that has been compiled and prepared into a program file.

The form of the :RUN command is:

:RUN ***progfile*** **[,*entry-point*] [;NOPRIV] [;LMAP] [;DEBUG]**
[;MAXDATA=*segsize*] [;PARM=*parameternum*] [;STACK=*stacksize*]
[;DL=*dlsize*] [;LIB=*library*] [;NOCB]

EXAMPLES:

```
:RUN PROGF,P1;DEBUG;LIB=P
:RUN $OLDPASS;MAXDATA=4096
```

where

progfile
is the name of the file which contains the compiled and prepared program to be executed.

The other parameters have the same meaning as shown with the :PREPRUN command.

10-19. USING EXTERNAL PROCEDURE LIBRARIES

Compiled SPL programs are stored in files called User Subprogram Libraries (USL's) that reside on disc. In any particular USL, each compiled program unit exists as a Relocatable Binary Module (RBM). To prepare a program, and any program unit it references, for execution, the MPE Segmenter selects the appropriate RBM's from the USL and binds them into linked segments written on a program file. For more information on the Segmenter, USL's and RBM's, refer to the *MPE Segmenter Subsystem Reference Manual.*

When you prepare and run programs in SPL, it is possible to reference external procedures in procedure libraries. You can build, modify, and maintain two types of procedure libraries within your log-on group and account: Relocatable Libraries (RL's) and Segmented Libraries (SL's).

10-20. RELOCATABLE LIBRARIES

A Relocatable Library (RL) is a specially formatted file that is searched at program preparation time to satisfy references to external procedures called by your program. Within such libraries, these procedures are placed in a single segment and linked to your program. Within such libraries, these procedures exist in RBM form (as they would on a USL). When a program is prepared, these procedures are placed in a single segment and linked to your program in the resulting program file.

For example, to specify that an RL named RLPROC be searched during preparation of a program from the USL file USL1 to the program file PROG1, you would enter the following :PREP command:

```
:PREP USL1,PROG1;RL= RLPROG
```

10-21. CREATING AND MAINTAINING RELOCATABLE LIBRARIES. To create and maintain relocatable libraries, you must access the Segmenter by entering the MPE :SEGMENTER command.

The form of the :SEGMENTER command is:

:SEGMENTER [*listfile*]

where

listfile
is an ASCII file from the output set (the formal designator is SEGLIST) to which is written any listable output generated by the Segmenter commands. The designator SEGLIST should not be used as the actual file designator. If the *listfile* is omitted, the standard job/session list device ($STDLIST) is assigned by default.

If you are in an interactive session, the Segmenter prompts you with a dash (-). Once the Segmenter is accessed, the following commands are used to create and maintain an RL:

-BUILDRL
Creates a permanent, formatted RL file.

-USL
References the USL file from which the procedure is to be obtained.

-RL
Identifies an existing RL.

-ADDRL
Adds a procedure to the currently identified RL.

-PURGERL
Deletes a procedure from an RL.

-LISTRL
Lists information concerning the currently identified RL.

The form of a -BUILDRL command is:

-BUILDRL *filereference,records,extents*

where

filereference
is the file name of the new RL, optionally including group and account identifiers.

records
is the total maximum capacity of the file, specified in terms of 128-word, binary logical records.

extents
is the total number of disc extents that can be dynamically allocated to the file as logical *records* are written to it. The size of each extent is determined by the records parameter value divided by the *extents* parameter value. The *extents* value must be between 1 and 16 inclusive.

The form of a -USL command is:

-USL ***filereference***

where

filereference
is the name and optional group and account names, of the USL file to be manipulated.

The form of the -RL command is:

-RL ***filereference***

where

filereference
is the name, plus optional group and account names, of the RL to be modified.

The form of the -ADDRL command is:

-ADDRL ***name*** **[*(index)*]**

where

name
is the name of the procedure to be added to the RL. This *name* is called the primary *entry-point* of the RBM containing the procedure.

index
is an integer further identifying the RBM. The *index* may be used when the currently-managed USL contains more than one active RBM of the same *name*. If *index* is omitted, a value of zero is assigned.

The form of the -PURGERL command is:

-PURGERL **[*rlspec,*]** ***name***

where

rlspec
is either UNIT or ENTRY. UNIT is used to delete the procedure identified by *name*. ENTRY is used to delete the entry-point identified by *name*. If *rlspec* is omitted, ENTRY is used.

name
if *rlspec* is UNIT, *name* is the name of the procedure to be deleted. If *rlspec* is ENTRY, *name* is the name of the entry-point to be deleted.

The form of a -LISTRL command is:

-LISTRL

Refer to the *MPE Segmenter Subsystem Reference Manual* for further discussions of these Segmenter commands.

10-22. SEGMENTED LIBRARIES

Segmented libraries (SL's) are specially formatted files that are searched at program run time to satisfy references to external procedures. These libraries, like program files, contain procedures in segmented (prepared) form. An individual procedure may exist in a segment containing many other procedures. When a procedure is referenced, the segment containing it is loaded with your program. Since the segmentation is not altered when different programs reference procedures in an SL, these procedures may be shared concurrently by other programs.

To specify that an SL file in your group account be searched, add the keyword parameter LIB=*library* in the :RUN command as follows:

```
:RUN PROG1;LIB=G
```

10-23. CREATING AND MAINTAINING SEGMENTED LIBRARIES. To create and maintain segmented libraries, you must first access the Segmenter by entering the MPE :SEGMENTER command.

The form of the :SEGMENTER command is:

:SEGMENTER [*listfile*]

where

listfile
is an ASCII file from the output set (the formal designator is SEGLIST) to which is written any listable output generated by the Segmenter commands. The designator SEGLIST should not be used as the actual file designator. If the *listfile* is omitted, the standard job/session list device ($STDLIST) is assigned by default.

If in an interactive session, you are prompted with a dash (-) for Segmenter commands. Once the Segmenter is accessed, the following commands are used to create and maintain an SL:

-BUILDSL
Creates a permanent, formatted SL file.

-SL
Identifies an existing SL file.

-ADDSL
Adds a procedure to the SL file currently being managed.

-PURGESL
Purges an entry-point from a segment in an SL, or the entire segment from the SL.

-LISTSL
Lists the procedures in the currently managed SL file.

In addition, the -USL and -LISTUSL Segmenter commands can be used as discussed under "Relocatable Libraries" (paragraph 10-20).

The form of a -BUILDSL command is:

-BUILDSL ***filereference,records,extents***

where

filereference
is a file whose local name is SL, plus optional group and account names.

NOTE

You can create an SL file with a local name other than SL, but such a file cannot be searched by the :RUN command.

records
is the total maximum file capacity, specified in terms of 128-word binary logical records.

extents
is the total number of disc extents that can be dynamically allocated to the file as logical records are written to it. The size of each extent is determined by the *records* parameter value divided by the *extents* parameter value. The extents value must be an integer between 1 and 16 inclusive.

The form of an -SL command is:

-SL ***filereference***

where

filereference
is the name of the SL to be modified, optionally including group and account names.

The form of an -ADDSL command is:

-ADDSL ***name*** **[;PMAP]**

where

name
is the name of the segment to be added to the SL.

PMAP
indicates that a listing describing the prepared segment will be produced on the *listfile* device specified in the :SEGMENTER command. If PMAP is omitted, the prepared segment is not listed.

The form of a -PURGESL command is:

-PURGESL [*unitspec*,] *name*

where

unitspec
is either ENTRY or SEGMENT. ENTRY is used to delete the entry-point identified by *name*. SEGMENT is used to delete the segment identified by *name*. If neither ENTRY nor SEGMENT is specified, ENTRY is used.

name
is the name of the entry-point or segment to be deleted.

The form of the -LISTSL command is:

-LISTSL

For further descriptions of these Segmenter commands, see the *MPE Segmenter Subsystem Reference Manual*.

ASCII CHARACTER SET

APPENDIX A

CHAR	BYTE POSITION Left	Right	Dec.
NUL	000000	000000	0
SOH	000400	000001	1
STX	001000	000002	2
ETX	001400	000003	3
EOT	002000	000004	4
ENQ	002400	000005	5
ACK	003000	000006	6
BEL	003400	000007	7
BS	004000	000010	8
HT	004400	000011	9
LF	005000	000012	10
VT	005400	000013	11
FF	006000	000014	12
CR	006400	000015	13
SO	007000	000016	14
SI	007400	000017	15
DLE	010000	000020	16
DC1	010400	000021	17
DC2	011000	000022	18
DC3	011400	000023	19
DC4	012000	000024	20
NAK	012400	000025	21
SYN	013000	000026	22
ETB	013400	000027	23
CAN	014000	000030	24
EM	014400	000031	25
SUB	015000	000032	26
ESC	015400	000033	27
FS	016000	000034	28
GS	016400	000035	29
RS	017000	000036	30
US	017400	000037	31
SPACE	020000	000040	32
!	020400	000041	33
"	021000	000042	34
#	021400	000043	35
$	022000	000044	36
%	022400	000045	37
&	023000	000046	38
'	023400	000047	39
(	024000	000050	40
)	024400	000051	41
*	025000	000052	42
+	025400	000053	43
,	026000	000054	44
-	026400	000055	45
.	027000	000056	46
/	027400	000057	47
0	030000	000060	48
1	030400	000061	49
2	031000	000062	50
3	031400	000063	51
4	032000	000064	52
5	032400	000065	53
6	033000	000066	54
7	033400	000067	55
8	034000	000070	56
9	034400	000071	57
:	035000	000072	58
;	035400	000073	59
<	036000	000074	60
=	036400	000075	61
>	037000	000076	62
?	037400	000077	63

CHAR	BYTE POSITION Left	Right	Dec.
@	040000	000100	64
A	040400	000101	65
B	041000	000102	66
C	041400	000103	67
D	042000	000104	68
E	042400	000105	69
F	043000	000106	70
G	043400	000107	71
H	044000	000110	72
I	044400	000111	73
J	045000	000112	74
K	045400	000113	75
L	046000	000114	76
M	046400	000115	77
N	047000	000116	78
O	047400	000117	79
P	050000	000120	80
Q	050400	000121	81
R	051000	000122	82
S	051400	000123	83
T	052000	000124	84
U	052400	000125	85
V	053000	000126	86
W	053400	000127	87
X	054000	000130	88
Y	054400	000131	89
Z	055000	000132	90
[	055400	000133	91
\	056000	000134	92
]	056400	000135	93
^	057000	000136	94
_	057400	000137	95
`	060000	000140	96
a	060400	000141	97
b	061000	000142	98
c	061400	000143	99
d	062000	000144	100
e	062400	000145	101
f	063000	000146	102
g	063400	000147	103
h	064000	000150	104
i	064400	000151	105
j	065000	000152	106
k	065400	000153	107
l	066000	000154	108
m	066400	000155	109
n	067000	000156	110
o	067400	000157	111
p	070000	000160	112
q	070400	000161	113
r	071000	000162	114
s	071400	000163	115
t	072000	000164	116
u	072400	000165	117
v	073000	000166	118
w	073400	000167	119
x	074000	000170	120
y	074400	000171	121
z	075000	000172	122
{	075400	000173	123
|	076000	000174	124
}	076400	000175	125
~	077000	000176	126
DEL	077400	000177	127

RESERVED WORDS

APPENDIX B

The following symbols have special meaning in SPL/3000 and thus, cannot be used as identifiers:

ABSOLUTE	ELSE	LAND	REAL
ALPHA	END	LOGICAL	RETURN
AND	ENTRY	LONG	SCAN
ARRAY	EQUATE	LOR	SET
ASSEMBLE	EXTERNAL	MOD	SPECIAL
BEGIN	FALSE	MODD	SPLIT
BYTE	FIXR	MOVE	STEP
CARRY	FIXT	MOVEX	SUBROUTINE
CASE	FOR	NOCARRY	SWITCH
CAT	FORWARD	NOT	THEN
CHECK	GLOBAL	NOVERFLOW	TO
COMMENT	GO	NUMERIC	TOS
DABZ	GOTO	OF	TRUE
DATASEG	IABZ	OPTION	UNCALLABLE
DDEL	IF	OR	UNTIL
DEFINE	INTEGER	OVERFLOW	VALUE
DEL	INTERNAL	OWN	VARIABLE
DELB	INTERRUPT	POINTER	VIRTUAL
DO	INTRINSIC	PRIVILEGED	WHILE
DOUBLE	IXBZ	PROCEDURE	WITH
DXBZ	LABEL	PUSH	XOR

BUILDING AN INTRINSIC FILE

APPENDIX C

The program BUILDINT is used to build or change intrinsic disc files. The program uses formal designators INTDECL and OUT for input and list output files respectively. The default files are $STDIN and $STDLIST. The intrinsic data file is opened as SPLINTR.

The command to execute the program is

```
:RUN BUILDINT.PUB.SYS
```

The input data consists of SPL procedure head declarations (OPTION EXTERNAL is required) and optional commands.

Without commands, the procedure head declarations are added to the intrinsic file.

Commands have the following purposes:

$PURGE	Removes all entries from the intrinsic file.
$REMOVE	Removes all entries which follow this command, until a $BUILD. Input has the same format as for adding entries.
$BUILD	Adds all subsequent input entries to the intrinsic file. $BUILD is required only if $REMOVE is used.

Any input data which is not a procedure head terminates input. At this point, the program prints a formatted list of all intrinsics and terminates.

For example,

```
:PURGE MYFILE
:BUILD MYFILE
:FILE SPLINTR= MYFILE
:RUN BUILDINT.PUB.SYS
INTEGER PROCEDURE M(A,B,C); VALUE A; INTEGER A,B;LOGICAL C;
OPTION EXTERNAL; PROCEDURE COMP(N,M'); VALUE N,M'; DOUBLE N;REAL M';
OPTION EXTERNAL;
PROCEDURE BYT(L,M,N,O); LABEL L; PROCEDURE M; BYTE ARRAY N;
LOGICAL POINTER O; OPTION EXTERNAL;
:EOD
```

See the next page for the formatted output for this file.

```
PAGE 0002   HEWLETT-PACKARD                    SPL INTRINSIC BUILDER

      TYPE                          OPTIONS                          PARAMETERS
   N  NONE                 0,1,2,3  LEVEL OF CHECKING      COLUMN 1          COLUMN 2          COLUMN 3
   L  LOGICAL              E        EXTERNAL              V  VALUE          T  SEE TYPE       S  SIMPLE VARIABLE
   I  INTEGER              V        VARIABLE              R  REFERENCE                        A  ARRAY
   B  BYTE                 I        INTERRUPT                                                 P  POINTER
   D  DOUBLE               U        UNCALLABLE                                                T  PROCEDURE
   R  REAL                                                                                    L  LABEL
   E  EXTENDED

  NAME         TYPE  OPTIONS  #PAR                                  PARAMETERS
                                      1    2    3    4    5    6    7    8    9   10   11   12   13   14   15   16   17   18   19   20
BYT              N     0E       4    RNL  RNT  RBA  RLP
COMP             N     0E       2    VDS  VRS
M                I     0E       3    VIS  RIS  RLS
 NO. ERRORS=000
```

Figure C-1. BUILDINT Output

Table C-1. BUILDINT Error Messages

MESSAGE	MEANING	ACTION
DECLARED TWICE	The identifier in question is not unique.	Correct to unique identifier.
EXPECTS A SEMICOLON	Only a comma or a semicolon is legal at this point.	
EXPECTS IDENTIFIER	An identifier is the only legal symbol at this point.	
EXPECTS NUMBER	The CHECK option has been specified but no legal check level follows.	
FORWARD OPTION IS ILLEGAL	The FORWARD option has been specified in a context where it is illegal.	
ILLEGAL SYMBOL	A left bracket, asterisk, or slash has been encountered, none of which are acceptable.	
INTERRUPT PROCEDURE MUST NOT HAVE PARAMETER	An interrupt procedure has been declared with a parameter; a parameter is illegal in this context.	
MISSING SPECIFICATION	A formal parameter has not been given a type specification.	
NUMERIC SYMBOL NOT ALLOWED	A fraction has been encountered which is not acceptable.	
READ ERROR	An error occurred while reading from the input file.	
SPECIFICATION DOES NOT CORRESPOND	There is no formal parameter with the name used in this specification.	
SUBROUTINES NOT ALLOWED	Subroutines are illegal in the intrinsic file.	Rewrite the intrinsic without subroutines.
TOO MANY PARAMETERS	There are more than 31 formal parameters.	Reduce the number of formal parameters.
TOO MANY OR ILLEGAL ATTRIBUTES	A specification for an identifier was made with more than one type or more than one class.	
VALUE SPECIFICATION DOES NOT CORRESPOND	A value specification exists for a non-existent formal parameter.	Either include the formal parameter or remove the value specification.

MPE INTRINSICS

APPENDIX D

Table D-1. Summary of MPE Intrinsics

INTRINSIC NAME	PURPOSE	CAPABILITY REQUIRED
ACCEPT	Accepts (and completes) a request received by the preceding GET intrinsic call. (Used only with DS/3000.)	Standard
ACTIVATE	Activates a process.	Process Handling
ADJUSTUSLF	Adjusts directory space in a USL file.	Standard
ALTDSEG	Alters the size of an extra data segment.	Data Segment Management
ARITRAP	Enables or disables internal interrupt signals from all hardware arithmetic traps.	Standard
ASCII	Converts a number from binary to ASCII code.	Standard
BINARY	Converts a number from ASCII to binary code	Standard
CALENDAR	Returns the calendar date.	Standard
CAUSEBREAK	Requests a session break.	Standard
CLEANUSL	Deletes inactive entries from USL file.	Standard
CLOCK	Returns the actual time.	Standard
CLOSELOG	Closes access to the logging facility.	LG Capability
COMMAND	Executes an MPE command programmatically.	Standard
CREATE	Creates a process.	Process Handling
CREATE PROCESS	Provides ability to assign $STDIN and $STDLIST to any file.	Process Handling
CTRANSLATE	Converts a string of characters from EBCDIC to ASCII or from ASCII to EBCDIC.	Standard
DASCII	Converts a value from double-word binary to ASCII code.	Standard
DATELINE	Returns date and time information.	Standard
DBINARY	Converts a number from ASCII code to a double-word binary value.	Standard
DEBUG	Calls the DEBUG facility.	Standard

Table D-1 Summary of MPE Intrinsics (Continued)

INTRINSIC NAME	PURPOSE	CAPABILITY REQUIRED
DLSIZE	Changes size of DL to DB area.	Standard
DMOVIN	Copies block from data segment to stack.	Data Segment Management
DMOVOUT	Copies block from stack to data segment.	Data Segment Management
EXPANDUSLF	Changes length of a USL file.	Standard
FATHER	Requests Process Identification Number (PIN of father process.	Process Handling
FCARD	Drives the HP 7260A Optical Mark Reader.	Standard
FCHECK	Requests details about file input/output errors.	Standard
FCLOSE	Closes a file.	Standard
FCONTROL	Performs control operations on a file or terminal device.	Standard
FDELETE	Deactivates a R10 record.	Standard
FDEVICE CONTROL	Adds control directives to a spooled device file.	Standard
FERRMSG	Returns message corresponding to FCHECK error number.	Standard
FFILEINFO	Provides access to file information.	Standard
FGETINFO	Requests access and status information about a file.	Standard
FINDJCW	Searches Job Control Word (JCW) table for specified JCW.	Standard
FLOCK	Dynamically locks a file.	Standard
FMTCALENDAR	Formats calendar date.	Standard
FMTCLOCK	Formats time of day.	Standard
FMTDATE	Formats calendar date and time of day.	Standard
FOPEN	Opens a file.	Standard
FPOINT	Resets the logical record pointer for a sequential disc file.	Standard
FREAD	Reads a logical record from a sequential file (on any device) to the user's data stack.	Standard
FREAD BACKWARD	Reads a logical record beginning at a point prior to the current record printer.	Standard
FREADDIR	Reads a logical record from a direct access file to the user's data stack.	Standard

Table D-1 Summary of MPE Intrinsics (Continued)

INTRINSIC NAME	PURPOSE	CAPABILITY REQUIRED
FREADLABEL	Reads a user file label.	Standard
FREADSEEK	Prepares, in advance, for reading from a direct-access file.	Standard
FREEDSEG	Releases an extra data segment.	Data Segment Management
FREELOCRIN	Frees all local Resource Identification Numbers (RIN's) from allocation to a job.	Standard
FRELATE	Determines if a file pair is interactive or duplicative.	Standard
FRENAME	Renames a disc file.	Standard
FSETMODE	Activates or de-activates file-access modes.	Standard
FSPACE	Spaces forward or backward on a file.	Standard
FUNLOCK	Dynamically unlocks a file.	Standard
FUPDATE	Updates a logical record residing in a disc file.	Standard
FWRITE	Writes a logical record from the user's stack to a sequential file (on any device).	Standard
FWRITEDIR	Writes a logical record from the user's stack to a direct-access disc file.	Standard
FWRITELABEL	Writes a user file label.	Standard
GENMESSAGE	Accesses MPE message system.	Standard
GET	Receives the next request from a remote master program. (Used only with DS/3000.)	Standard
GETDSEG	Creates an extra data segment.	Data Segment Management
GETJCW	Fetches contents of system job control word (JCW).	Standard
GETLOCRIN	Acquires local RIN's.	Standard
GETORIGIN	Determines source of process activation call.	Process Handling
GETPRIORITY	Changes the priority of a process.	Process Handling
GETPRIVMODE	Dynamically enters privileged mode.	Privileged Mode
GETPROCID	Requests PIN of a son process.	Process Handling

Table D-1 Summary of MPE Intrinsics (Continued)

INTRINSIC NAME	PURPOSE	CAPABILITY REQUIRED
GETPROCINFO	Requests status information about a father or son process.	Process Handling
GETUSERMODE	Dynamically returns to non-privileged mode.	Privileged Mode
INITUSLF	Initializes a USL file to the empty state.	Standard
IODONTWAIT	Initiates completion operations for an I/O request.	Privileged Mode
IOWAIT	Initiates completion operations for an I/O request.	Privileged Mode
KILL	Deletes a process.	Pocess Handling
LOADPROC	Dynamically loads a library procedure.	Standard
LOCKGLORIN	Locks a global RIN.	Standard
LOCKLOCRIN	Locks a local RIN.	Standard
LOCRINOWNER	Identifies process locking a local RIN.	Standard
MAIL	Tests mailbox status.	Process Handling
MYCOMMAND	Parses (delineates and defines parameters) for user-supplied command image.	Standard
OPENLOG	Provides access to a logging facility.	LG Capability
PAUSE	Suspends calling process for a specified number of seconds.	Standard
PCHECK	Returns an integer code specifying the completion status of the most recently executed DS/3000. (Used only with DS/3000.)	Standard
PCLOSE	Terminates program-to-program communication with a remote slave program. (Used only with DS/3000.)	Standard
PCONTROL	Exchanges tag fields with a remote slave program. (Used only with DS/3000.)	Standard
POPEN	Initiates program-to-program communication with a remote slave program. (Used only with DS/3000.)	Standard
PREAD	Requests a block of data from a remote slave program. (Used only with DS/3000.)	Standard
PRINT	Prints character string on job/session list device.	Standard
PRINTFILEINFO	Prints file information display.	Standard

Table D-1 Summary of MPE Intrinsics (Continued)

INTRINSIC NAME	PURPOSE	CAPABILITY REQUIRED
PRINTOP	Prints a character string on the Operator's Console.	Standard
PRINTOPREPLY	Prints a character string on the Operator's Console and solicits a reply.	Standard
PROCTIME	Returns a process' accumulated central processor time.	Standard
PTAPE	Accepts input from paper tapes which do not contain X-OFF control characters.	Standard
PUTJCW	Puts value of a given JCW in JCW table.	Standard
PWRITE	Sends a block of data to a remote slave program.	Standard
QUIT	Aborts a process.	Standard
QUITPROG	Aborts the user process structure.	Standard
READ	Reads an ASCII string from the job/session input device ($STDIN).	Standard
READX	Reads an ASCII string from the job/session input device ($STDINX).	Standard
RECEIVEMAIL	Receives mail from another process.	Process Handling
REJECT	Rejects the request received by the preceding GET intrinsic call. (Used only with DS/3000.)	Standard
RESETCONTROL	Resets terminal to accept CONTROL Y signal.	Standard
RESETDUMP	Disables the abort stack analysis facility.	Standard
SEARCH	Searches an array for a specified entry or name.	Standard
SENDMAIL	Sends mail to another process.	Process Handling
SETDUMP	Enables the abort stack analysis facility.	Standard
SETJCW	Sets the value of the system job control word (JCW).	Standard
STACKDUMP	Dumps selected parts of stack to file.	Standard
SUSPEND	Suspends a process.	Process Handling
SWITCHDB	Switches DB register pointer.	Privileged Mode
TERMINATE	Terminates a process.	Standard
TIMER	Returns job or session timer bit count.	Standard
UNLOADPROC	Dynamically unloads a library procedure.	
UNLOADGLORIN	Unlocks a global RIN.	Standard

Table D-1 Summary of MPE Intrinsics (Continued)

INTRINSIC NAME	PURPOSE	CAPABILITY REQUIRED
UNLOCKLOCRIN	Unlocks a local RIN.	Standard
WHO	Returns user attributes.	Standard
WRITELOG	Writes a record to a logging file.	LG Capability
XARITRAP	Arms or disarms the software arithmetic trap.	Standard
XCONTRAP	Arms or disarms the CONTROL-Y trap.	Standard
XLIBTRAP	Arms or disarms the library trap.	Standard
XSYSTRAP	Arms or disarms the system trap.	Standard
ZSIZE	Changes size of Z to DB area.	Standard

COMPILER ERROR MESSAGES

APPENDIX E

Table E-1. SPL Compiler Error Messages

MESSAGE	MEANING	ACTION
ARITHMETIC RIGHT SHIFT EMITTED	Compiler has issued an ASR to convert a byte address to a word address.	None, unless word address is supposed to be greater than DB+ 16383 in which case the ASR causes an error.
BEGIN END DO NOT MATCH	When END. encountered, there were more BEGINs than ENDs.	Check your code and correct.
CASE STATEMENT OVERFLOW	The number of cases in a CASE statement exceeds 256.	Check your code; decrease the number of cases.
CONVERSION ERROR	An illegal type conversion was attempted.	Check manual for legal type conversions; note that types cannot be mixed in arithmetic operations.
DECLARATION NOT ALLOWED IN SUBROUTINE	A subroutine may not have declarations.	Check the subroutine code and move declarations to main program or procedure.
DECLARATION OUT OF ORDER	Declarations must be ordered as: data, procedures, subroutines.	Check the order; correct.
DECLARED TWICE	An identifier has been declared twice at the same level.	Check declarations; correct.
DEFINE TOO LARGE	A DEFINE declaration has too many characters in its description.	Check declaration, reduce to 511 characters excluding extraneous blanks.
DISPLACEMENT OUT OF RANGE	The displacement is too large or has the wrong sign for the addressing mode.	Displacement varies with addressing mode: DB + 255 Q + 127; Q – 63 S – 63 P + 255; P – 255
DISPLACEMENT TOO LARGE	The displacement is too large for the addressing mode.	
EXCEEDED MAXIMUM INCLUDE DEPTH	INCLUDEs are nested to a level greater than 10.	Check your code; decrease the nesting level of INCLUDEs.
EXPECTS ALPHA	The next symbol must be an alphabetic character.	Check code; change to alphabetic character.
EXPECTS ARRAY IDENTIFIER	Only an array identifier is legal in this context.	Check code; use array identifier.

Table E-1. SPL Compiler Error Messages (Continued)

MESSAGE	MEANING	ACTION
EXPECTS ASTERISK	An asterisk is expected in this context.	Check code; use asterisk.
EXPECTS BOUNDS	An array declaration of this type requires bounds.	Check code; enter bounds.
EXPECTS CONSTANT	A constant is expected in this context; for example, as a partial word designator.	Check code; correct.
EXPECTS DOLLAR	A $ command with continuation symbol is not followed by image with $ in column 1.	Correct by entering $ at beginning of continuation line or deleting continuation symbol.
EXPECTS EQUAL	An equals sign is expected in this context.	Check code and enter = where expected.
EXPECTS FILE	Filename expected, but not found	Check your code and correct
EXPECTS IDENTIFIER REFERENCE	Identifier name not found	Check your code and correct
EXPECTS INTEGER VARIABLE	Only as integer variable is legal in this context	Check code, correct.
EXPECTS LABEL	A label must appear in this context.	Check code, correct
EXPECTS OR	OR was expected but not found.	Check your code and correct.
EXPECTS OPTION	A $ command has an illegal command or is followed by an illegal parameter.	Check command, correct.
EXPECTS POINTER	Only a pointer is legal in this context.	Check code, correct.
EXPECTS REFERENCE PARAMETER	A value parameter is passed to a procedure that expects a parameter passed by reference.	Check parameters and specifications; correct.
EXPECTS RELATIONAL	A relational operator is expected at this point.	Check code, correct by including relational operator (=,<>,<,<=,>,>=)
EXPECTS RELATIONAL OR COMMA	Either a comma or a relational operator is expected in this context.	Check code, correct by including comma or relational operator (=,<>,<,<=,>,>=) as appropriate.

Table E-1. SPL Compiler Error Messages (Continued)

MESSAGE	MEANING	ACTION
EXPECTS SYMBOL	No symbol where a symbol, such as an identifier, is expected.	Check code, include symbol.
EXPECTS UNDEFINED BOUNDS	An array declaration of this type requires an asterisk (*).	Check declaration, include *
EXPECTS VARIABLE	Only a variable is allowed in this context.	Check code, correct.
FILENAME TOO LONG	Filename is greater than 8 characters	Check your code and shorten name
ILLEGAL ADDRESS MODE	The specified address mode is not legal in this context	Address mode relative to DB, Q, S, or PB must be changed.
ILLEGAL ADDRESS STORE	An attempt has been made to store into a non-existent pointer; for example: @PTR(1):=0	Change to @PTR:=n or PTR(1)=n
ILLEGAL ASSEMBLE STATEMENT	An error occurred in an ASSEMBLE statement	Check the statement; correct
ILLEGAL ATTRIBUTE	Attribute inconsistent with identifier; e.g., LONG LABEL.	Check the specification; correct
ILLEGAL BOUNDS SPECIFICATIONS	The bounds for this array declaration are invalid	Check that bounds are *, @, or integer constant.
ILLEGAL CLASS	Symbol class (POINTER, ARRAY, etc.) incorrect in context.	Check the symbol; correct the symbol class.
ILLEGAL CONSTANT	This symbol is not a valid constant.	Check the constant, enter a valid constant.
ILLEGAL DYNAMIC BOUNDS	The dynamic bounds must be either an integer formal parameter or a global integer.	Correct as indicated.
ILLEGAL EXTERNAL VARIABLE	An error occurred in an external variable declaration or in its use.	Check the declaration and also the procedure where it is used; correct.
ILLEGAL FORMAL PARAMETER	The attributes specified for this formal parameter are not valid.	Check the parameter; correct.
ILLEGAL GLOBAL EXTERNAL VARIABLE	An error has occurred in a global or an external variable declaration.	Check declarations; correct.

Table E-1. SPL Compiler Error Messages (Continued)

MESSAGE	MEANING	ACTION
ILLEGAL IDENTIFIER REFERENCE	The reference identifier for this declaration is incorrect.	Check the declaration; reference identifier must be declared first.
ILLEGAL INITIALIZATION	The initialization list for this array is invalid.	Make sure that list contains only numeric values or strings.
ILLEGAL IF STATEMENT	This IF statement contains an error	Check the statement, correct.
ILLEGAL IN SPLIT-STACK MODE	An error was detected inside a WITH statement or with OPTION SPLIT or $SPLIT.	Check WITH and OPTION SPLIT in manual.
ILLEGAL ITEM IN EXPRESSION	The item is either not declared or is of the wrong class	Check declarations, include if necessary, otherwise correct
ILLEGAL LEFT PARENTHESIS	A left parenthesis has been used in a context where it is illegal	Remove the parenthesis
ILLEGAL MODE IN THIS CONTEXT	An address mode (relative to DB, Q, S, or PB) cannot be used in this context	Change to a mode that is legal in this context.
ILLEGAL OPERATOR	An operator is used that is not recognized by the compiler	Valid operators are: *, /, **, .+, .–, .MOD, .MODD, .=, .<, .<>, .<=, .>, .>=, LAND, LOR, XOR.
ILLEGAL OWN INITIALIZATION	The initialization list for an OWN array is invalid.	Check, correct the list to include only numbers and strings.
ILLEGAL OWN VARIABLE	An error occurred in an OWN variable declaration or in its use.	Check the OWN variable declaration and also where it is used; correct.
ILLEGAL PARAMETER	This parameter contains an illegal item.	Check the parameter; correct.
ILLEGAL S-RELATIVE ADDRESS	The displacement to S is either positive or less than – 63.	Correct the address to fall within range S-0 through S-63.
EXPECTS WHILE OR UNTIL	The reserved word WHILE or UNTIL is missing.	Check code, include WHILE or UNTIL.
EXPECTS @	The compiler expects an @ as the next symbol in this context.	Check code, include @.
ERROR IN CATENATE EXPRESSION	A catenate expression must be of the form (L:M:N) where L, M, and N are integer constants.	Check expression and correct.

Table E-1. SPL Compiler Error Messages (Continued)

MESSAGE	MEANING	ACTION
ERROR IN PARTIAL WORD DESIGNATOR	A partial word designator must be of the form (M:N) where M and N are integer constants.	Check code; correct form of partial word designator.
ERROR IN SHIFT DESIGNATOR	An illegal mnemonic follows the &.	Change mnemonic to a valid shift identifier.
ERROR IN USL FILE	USL file contains a bad entry. Compilation terminates.	Check source for errors; correct and try again.
ERROR OVERFLOW	Maximum number of errors has been generated.	Default maximum = 100 errors; change with $CONTROL command.
FORWARD PROCEDURE DECLARATION INCOMPATIBLE	Forward and actual procedure declarations do not match	Check declarations and correct.
ILLEGAL SEGMENTATION	A $CONTROL SEGMENT card is within a procedure.	Change the card to appear outside the procedure.
ILLEGAL STATEMENT BEGINNER	A statement cannot begin with this class; possibly is an undeclared variable.	Check the class, and if undeclared variable, declare it.
ILLEGAL STATEMENT TERMINATOR	A statement must be terminated by END or a semicolon.	Correct the terminator.
ILLEGAL STRING	A string is expected in this context but there are no quote marks.	Enclose the string in quotes.
ILLEGAL SYMBOL	Not an ASCII character valid for SPL.	Check and enter a valid ASCII character acceptable to SPL.
ILLEGAL TO STACK PARAMETER	Parameter must not be loaded directly to stack in this context or stack will be out of order.	Correct so that parameter is not stacked.
ILLEGAL TRACE CARD	A $TRACE card is either in the wrong position or contains an error.	Check the $TRACE card and move or correct as appropriate.
ILLEGAL TRACE IDENTIFIER	The identifier being traced is of a class that cannot be traced.	Change class to SIMPLE VARIABLE, ARRAY, POINTER, LABEL, or PROCEDURE.
ILLEGAL TYPE	A type mismatch has occurred in an arithmetic operation.	Check the types and change to matching types.

Table E-1. SPL Compiler Error Messages (Continued)

MESSAGE	MEANING	ACTION
ILLEGAL TYPE TRANSFER	The type of the operand may not be converted to the type of the object in SPL.	Check the statement and correct to avoid type mismatch.
ILLEGAL USE OF PB BYTE ARRAY	Byte cannot be loaded from a PB byte array since the load byte instruction is not PB-relative.	Correct code so attempt is not made to load byte from PB byte array.
ILLEGAL VARIABLE	Form of variable is not valid.	Check variable and insure that it starts with letter.
ILLEGAL X ON OR OFF	Parameter on $IF command is invalid; may be X0 through X9 = ON or OFF only.	Check $IF parameter and correct.
ILLEGAL X REGISTER REFERENCE	Either the type or the class of the variable referencing the X register is illegal.	Change type and/or class to that of a one-word variable
INDEX NOT ALLOWED	An attempt was made to index a simple variable.	Change declaration to array or remove index.
INITIALIZATION OUT OF RANGE	An array has been initialized with a list that is larger than the array size.	Either change the array size or decrease the list.
INTEGER OVERFLOW	A constant expression resulted in an integer overflow.	Check constants used in expressions for a resulting value greater than 32767 or less than −32767.
INVALID BRANCH EMITTED	Compiler has emitted a bad branch in ASSEMBLE statement; probably label out of range.	Check label range; change to indirect branch.
INVALID BYTE INITIALIZATION	The initialization list of a byte array is incorrect.	Check byte array and its initialization list; correct.
INVALID COMMENT	Comment has been used in an illegal context.	Check code; either move or remove comment.
INVALID EXPONENT PARAMETER	An exponent expression contains an error.	Check the expression; correct.
INVALID NUMBER	Either the field is not numeric or the number is out of range in this context.	Check field and range of number; correct.
INVALID OPERATOR MNEMONIC	The mnemonic in ASSEMBLE statement not identifiable.	Check code for invalid instruction mnemonic; correct.

Table E-1. SPL Compiler Error Messages (Continued)

MESSAGE	MEANING	ACTION
INVALID SDEC	Stack decrement (SDEC) field in statement such as MOVE or SCAN is out of range.	Check range for this SDEC constant and correct.
INVALID SUBSCRIPT	An index must be an integer expression.	Check expression used as index; correct.
LABEL IN ASSEMBLE STATEMENT MUST OCCUR	A label referenced in an ASSEMBLE statement cannot be found.	Check statement; either include label or remove reference.
LOCAL DECLARATION OVERFLOW	Too many local declarations; up to 127 words allowed.	Check and remove extra declarations.
LOCAL INITIALIZATION MUST BE PB	A local array can be initialized only in PB mode.	Check array declaration; change mode to PB, or make array global.
LOGICAL COMPARE EMITTED	Issued when a logical compare always gives the same result.	Warning that compare such as L>=0 is always true, L<0 always false if L is logical variable.
MAY NOT GO TO ENTRY	A GO TO statement may not transfer to an entry label.	Check GO TO; change label.
MAY NOT TRACE EXTERNAL LABEL	Trace can only be made on label in program unit being compiled.	Check TRACE; change label to one in current program unit.
MAXIMUM REPEAT FACTOR 8191	The largest repeat factor allowed in an initialization list is 8191.	Check initialization list; lower repeat factor.
MISSING ASSIGNMENT OPERATOR	An assignment operator must appear in this context.	Check code; include assignment operator.
MISSING BEGIN	The compiler expects a BEGIN as the next symbol.	Check code; include BEGIN.
MISSING CCF	This ASSEMBLE instruction requires a CCF specification.	Check code; include CCF specification.
MISSING COLON	A colon (:) must appear in this context.	Check code; include colon.
MISSING COMMA	A comma (,) is expected in this context.	Check code; include comma.
MISSING DO	A DO must appear in this context.	Check code; include DO.
MISSING ELSE	An ELSE must appear in this context.	Check code; include ELSE.

Table E-1. SPL Compiler Error Messages (Continued)

MESSAGE	MEANING	ACTION
MISSING EXPONENT	A valid exponent must follow a caret (^).	Check code; enter valid exponent.
MISSING FORMAL PARAMETER	A specification is made for a non-existent formal parameter.	Check code; include formal parameter or delete specification.
MISSING LEFT PARENTHESIS	A left parenthesis is expected in this context.	Check code; include left parenthesis.
MISSING OF	A CASE statement does not contain the word OF.	Check CASE statement; include OF.
MISSING RIGHT BRACKET	A right bracket is only acceptable symbol at this point.	Check code and include right bracket.
MISSING RIGHT PARENTHESIS	A right parenthesis is expected at this point	Check code; include right parenthesis.
MISSING SEMICOLON	A semicolon (;) or other separator is required in this context	Check code; include semicolon.
MISSING SLASH	A slash is the only acceptable symbol at this point.	Check code; include slash.
MISSING SPECIFICATION	There is no specification for a formal parameter.	Check code; include specification for formal parameter.
MISSING SUBPROGRAM	A procedure specified in a $CONTROL SUBPROGRAM command cannot be found.	Check code; correct name in command or include procedure.
MISSING THEN	A THEN must appear in this context.	Check code; include word THEN.
MISSING UNTIL	An UNTIL must appear in this context.	Check code; include word UNTIL.
MULTIPLE FORWARD DECLARATION	There is more than one forward declaration for this procedure.	Check declarations; remove redundant forward declaration.
MULTIPLE SPECIFICATIONS	A formal parameter is specified more than once.	Check code; remove extra formal parameter.
MUST BE DB	Only DB-relative addressing is allowed in this context.	Check address; correct to DB-relative.
MUST BE DB OR Q	Only DB-relative or Q-relative addressing allowed in this context.	Check address; correct to DB-relative or Q-relative.
MUST BE DOUBLE OR LOGICAL	Only a double-word or logical variable is allowed in this context.	Check variable; change to double or logical.

Table E-1. SPL Compiler Error Messages (Continued)

MESSAGE	MEANING	ACTION
MUST BE INTEGER TYPE	The only valid type for this construct is integer.	Check code; use integer.
MUST BE INTEGER, LOGICAL OR BYTE	A one-word quantity is expected in this context.	Check code; correct to use one-word quantity.
MUST BE LOCAL	Action allowed only for local is being performed on global variable.	Check code; correct variable.
MUST BE TYPE BYTE	Symbol must be type byte in this context.	Check symbol; correct if illegal or change to type byte.
MUST BE TYPE LOGICAL	Only a logical variable can appear in a Boolean expression.	Check expression; change to logical variable.
MUST BE TYPE PROCEDURE	In this context, procedure must be typed.	Check code; change to typed procedure.
MUST BE VALUE FORMAL PARAMETER	A reference parameter is not legal in this context.	Check parameter; change to formal parameter.
NESTED PROCEDURE NOT ALLOWED	A procedure declaration is within another procedure.	Check code; remove procedure declaration for other procedure.
NESTED REPEAT FACTOR	Repeat factor inside a repeat factor is not allowed	Check code.
NOT END OF COMMENT	Two greater-than symbols are separated by one or more blanks.	If intended as comment, remove blanks so symbols are adjacent (>>).
NOT INTRINSIC FILE	A file specified as an intrinsic file in INTRINSIC statement is not an intrinsic file.	Check file name; change to name of intrinsic file.
NOT ON INTRINSIC FILE	Procedure referenced in an INTRINSIC declaration is not on the intrinsic file.	Check procedure name and intrinsic file; change name or include intrinsic in file.
OUT OF RANGE BRANCH	An ASSEMBLE statement contains branch that is beyond range of direct branch.	Check statement; change range of branch or use indirect addressing.
PARAMETER NOT ALLOWED	Interrupt procedure that should have no parameters has a parameter.	Check procedure; remove parameter.
PARAMETER NUMBER INCOMPATIBLE	A procedure call has an incorrect number of parameters.	Check procedure; change number of parameters accordingly.

Table E-1. SPL Compiler Error Messages (Continued)

MESSAGE	MEANING	ACTION
PARAMETER OUT OF RANGE	This parameter exceeds the maximum allowable displacement for this address mode.	Displacements may be: DB+255, Q+127, Q−63, S−63, P+255, P−255.
PARAMETER OVERFLOW	There are more than 31 parameters in this procedure.	Reduce number of parameters to 31 or fewer.
PARTIAL WORD ILLEGAL HERE	A partial word designator is not allowed in multiple store.	Break into several store statements to allow bit deposit.
PRIMARY DB OVERFLOW	A variable cannot be assigned with a DB-relative address greater than 255, or total is greater than 907 words.	Correct to address within accepted bounds possibly by removing declarations.
PRIMARY Q OVERFLOW	Variable cannot be assigned with Q-relative address greater than 127.	Correct assignment to address within acceptable bounds.
PROCEDURE TOO LARGE	The number of instructions in this procedure exceeds the limit.	Decrease number of instructions in procedure or increase segment size.
RECURSIVE DEFINE	Invoking this DEFINE statement would result in infinite loop.	Check text of DEFINE statement for identifier being defined.
RESERVED SYMBOL REDEFINED	Cannot define a constant or reserved word.	Check definition; omit reserved word or symbol.
SDEC TOO LARGE	Stack decrement in an ASSEMBLE statement is larger than largest allowed value	Check statement; reduce stack decrement to acceptable value for context.
SECONDARY DB OVERFLOW	There are too many declarations in the outer block	Check code, and reduce the number of declarations.
SEMICOLON NOT ALLOWED	A semicolon (;) cannot be used in this context.	Remove semicolon.
SEQUENCE ERROR	Input files contain images that are out of order.	Check input files; correct order.
SIZE INCOMPATIBILITY	Parameter passed to a procedure has wrong number of words.	Check parameter size in procedure, and correct call.
SORT TABLE OVERFLOW	Table used to sort map output is full (over 1162 procedures/symbols, 1912 globals)	Symbol table map cannot be produced.

Table E-1. SPL Compiler Error Messages (Continued)

MESSAGE	MEANING	ACTION
STRING TOO LARGE	This string exceeds 128 characters.	Reduce string size to acceptable limit.
SYMBOL TABLE ERROR	Some entries in the symbol table are no longer valid.	Symbol table map cannot be produced.
SYMBOL TABLE OVERFLOW	The compiler limit for the number of symbols has been exceeded.	Reduce number of symbols in program and recompile
STACK OVERFLOW MAY BE IRRECOVERABLE	If stack overflow occurs and Q and S set in same instruction, process may terminate	Separate into two instructions: e.g., SET (Q), SET (S), not SET (Q,S).
SUBPROGRAM TABLE OVERFLOW	Overflow in table where subprogram names to be compiled are stored.	Reduce number or size of names to total of 252 characters plus 1 extra for each name
SUBPROGRAM & USLINIT	This compilation specifies both subprogram and USLINIT, resulting in no outer block	Compile an outer block before preparing the program file
TOO MANY USL HEADERS	Too many procedure calls inside code block.	Reduce the number of procedure calls.
TRACE HEADER TOO LARGE	Too many symbols being traced resulting in table overflow.	Reduce number of symbols to be traced.
TYPE INCOMPATIBILITY	In arithmetic statement, two operands of different type are combined	Change one or both operands so that they are the same type (REAL, LONG, etc.)
TYPE PROCEDURE STORE OUT OF RANGE	A procedure name can appear on the left-hand side of a replacement operator (:=) only within the scope of the procedure with the same name.	Check procedure name; correct name or remove statement.
UNDECLARED IDENTIFIER	An identifier used in a statement has not been declared in a declaration.	Declare identifier or change identifier name to a declared identifier.
USL FILE OVERFLOW	The USL file is full.	Build larger USL file; recompile.
@ NOT ALLOWED	An @ is not legal in this context.	Remove @

CALLING SPL FROM OTHER LANGUAGES

APPENDIX F

There are a number of things to consider when writing SPL procedures that are to be called from other languages. Not all languages pass parameters in the same way and some have restrictions as to their ability to call function procedures, OPTION VARIABLE, and so forth. This note summarizes these restriction for BASIC, COBOL, COBOL II, and FORTRAN.

There are two ways to pass a parameter to a procedure: by REFERENCE and by VALUE. Passing a parameter by reference means that the 16-bit ADDRESS of the variable is passed on the stack; the called procedure refers to this parameter via indirect memory reference instructions (LOAD Q-n, I and STOR Q-n, I). Passing a parameter by value means that the actual contents of the variable (1, 2, or 4 words) are passed on the stack; the called procedure refers to this parameter via direct memory reference instructions (LOAD Q-n and STOR Q-n). As a result, if the called procedure modifies a call-by-reference parameter, the caller's variable is modified; for call-by-value parameters, only the "temporary" copy in Q-minus storage is changed (the caller's version retains its old value).

OPTION VARIABLE is a facility that provides the ability to call a procedure with a varying number of parameters. The called procedure will expect a "bit mask" in Q-4 (and Q-5 if there are more than 16 parameters) with bits set indicating which parameters are present. Parameters are always passed in the same Q-minus addresses; the Q-minus locations for parameters which are omitted have undefined values. It is up to the called procedure to examine the bit mask and to access only those parameters which are passed on any particular call.

A function procedure is one which returns a value in place of its name; it therefore can be called from an expression and the value that it returns will be used in the expression. This value is stored in the stack just before (lower address) the parameters to the procedure. It is the responsibility of the caller to dispose of or use the return value properly. An example of such a procedure is the BINARY intrinsic.

Because the various languages have differing capabilities for dealing with the various aspects of procedure calls, the SPL coder needs to be aware of what each language does. Below are summarized the things that need to be considered for each language.

COBOL

- All parameters are passed as WORD addresses (call-by-reference). There is one exception: you can pass the MPE file number for a file opened with the OPEN verb by passing the FD-name to a procedure; this is passed as a 16-bit integer by value.

- COBOL has no way of coping with the return value of a function procedure; an extra value will be left on the stack which will disrupt program execution. Do not call function procedures from COBOL.

- There is no way for COBOL to generate the bit msk required by OPTION VARIABLE procedures, so these cannot be called either. Since it is impossible to pass a parameter from COBOL by value, you can't generate the bit mask yourself.

- The following illustrates how the COBOL data types map to SPL data types:

COMPUTATIONAL	
1-4 digits	INTEGER
5-9 digits	DOUBLE
COMPUTATIONAL-3	SPL has no PACKED DECIMAL capability; you must access this as a byte array and generate the machine instructions yourself. Note that COBOL passes a WORD address for this; you will need to use an equivalenced byte array.
DISPLAY	Passed as LOGICAL (array). You will usually want to equivalence a byte array to the passed parameter and access the data this way.

Note that COBOL has no equivalent of REAL or LONG.

FORTRAN

- FORTRAN passes all parameters by reference unless the parameter is enclosed in backslashes, in which case it is passed by value. You may use a constant or expression in a call; if it is not enclosed in backslashes, a temporary cell is created and the address of the cell is passed.

- FORTRAN may call function procedures normally (external function).

- If you are calling an OPTION VARIABLE procedure, you must calculate the bit mask required and pass it as a constant by value as the LAST (or last two) parameter(s). See below for form of the bit mask.

- The following illustrates how FORTRAN data types map to SPL data types:

INTEGER/INTEGER*2	INTEGER
INTEGER*4	DOUBLE
REAL	REAL
DOUBLE PRECISION	LONG
CHARACTER*n	BYTE ARRAY

- When calling an intrinsic, you should name the intrinsic in a SYSTEM INTRINSIC statement. Then FORTRAN will take care of the OPTION VARIABLE mask, passing of parameters by reference or value, and so on.

BASIC

- BASIC passes all parameters by reference. There is no way to override this; if you pass a constant or expression, a temporary cell is created and the address of the cell is passed.

- BASIC, like COBOL, can't handle the return value from a function procedure. Likewise, it has no ability to generate an OPTION VARIABLE bit mask. Because all parameters are call-by-reference, you cannot generate a proper bit mask.

- BASIC passes a parameter type descriptor just in front of (lower memory address) the first parameter. The called procedure may use this or ignore it — see the BASIC Interpreter reference manual for details. This descriptor does not interfere with the normal addresses of the parameters.

- The following illustrates how BASIC data types map to SPL data types:

REAL/undeclared	REAL
LONG	LONG
INTEGER	INTEGER
String (x$)	BYTE ARRAY

Please keep in mind that the default constant in BASIC is type-REAL. To pass an integer, you must either store the value into an integer variable and pass the variable or use the following construct:

```
DEF INTEGER FNI(N)=N
        . . .
CALL proc(FNI(4))
```

This will pass the 4 as an integer instead of a real number.

Arrays and strings have physical and logical length information stored in the −2 and −1 elements of the array. (See the Basic Interpreter Reference Manual.) The point to note here is that if you change the length of a string or array, you must update the logical length so that BASIC knows what you did. Two-dimensional arrays and string arrays have length information at the beginning of each major dimension or string element.

(See below for a discussion on converting byte addresses to word addresses.)

COBOL II

- Much like FORTRAN, COBOL II passes all parameters by reference unless the parameter is enclosed in backslashes, in which case it is passed by value.

- All parameters are passed as WORD addresses unless an @ is used in front of the parameter name, in which case a BYTE address is passed.

- If you are calling a function procedure, an extension to the CALL statement (the GIVING clause, as in CALL proc USING parm GIVING value) allows you to pick up the return value; you MUST use this construct if you are calling a function procedure (even if you have no use for the return value) so that the stack is decremented properly.

- As with FORTRAN, you can generate the bit mask for OPTION VARIABLE procedures by passing it by value as the last parameter(s).

- COBOL II allows you to call intrinsics via the CALL INTRINSIC statement, relieving you of worrying about value v. reference, byte addressing, the OPTION VARIABLE mask, and so forth.

- The data types are precisely the same as for COBOL, above.

OPTION VARIABLE mask

The OPTION VARIABLE MASK IS ONE WORD AT Q-4 (or two words at Q-5 and Q-4 if there are more than 16 parameters) that describes which parameters are present. The RIGHTMOST bit (bit 15 in HP3000 nomenclature) corresponds to the rightmost (last) parameter; bit 14 refers to the next-to-last, and so forth on back to the first parameter. A 1 bit means the parameter is present; 0 means that the parameter was omitted and it should not be accessed.

For example, suppose we have the following procedure head:

```
PROCEDURE upshift(string,length,result);
VALUE length;
BYTE ARRAY string;
INTEGER length,result;
OPTION VARIABLE;
```

and we wish to call this from FORTRAN. What would be the proper CALL statement? Since there are three parameters, the last three bits of the mask would be used. If all parameters were included, the call would look like this:

```
CALL UPSHIFT(CHARSTRING, LEN ,IRESULT, %7L )
```

If, for example, the last parameter (RESULT) were omitted, the call would be:

```
CALL UPSHIFT(CHARSTRING, LEN , 0 , %6L )
```

The zero as the third parameter is required as a place holder.

Byte to word address conversion

It is sometimes desirable (or necessary) to convert a passed byte address to a word address (so that the array can be passed to the file system intrinsics, for example). You will find that if you attempt to equivalence a word array back to a passed byte array you will get a warning "ARITHMETIC RIGHT SHIFT EMITTED." What this is saying is that the SPL compiler is emitting an ASR 1 instruction to convert the byte address to a word address, and you are being warned because this is not always the

correct thing to do. The reason for this is that it is possible to have byte addresses that point to the DB-minus area (in fact, BASIC does this all the time) but it is impossible to tell if an address is in the DB-minus area or is simply a very large DB-plus byte address without looking at the registers. Here is a foolproof procedure that will generate the proper word address given any byte address provided that the byte address is not odd.

```
INTEGER PROCEDURE wordadr(byteadr);
ARRAY;
BYTE byteadr;
BEGIN
  INTEGER S0=S; <<Address of S>>
  tos:=tos:=@byteadr & LSR(1); <<Logical divide by 2>>
  IF tos>@S0 then tos.(0:1):=1; <<If in DB-minus, fix sign>>
  wordadr:=tos
END; <<wordadr>>
```

Sample call:

```
PROCEDURE sample(string);
BYTE ARRAY string;
BEGIN
  POINTER stringp; <<Word pointer>>
  @stringp;=wordadr(string);
            . . .
```

INDEX

READER COMMENT SHEET

HP 3000 Computer System

System Programming Language Reference Manual

30000-90024 Feb 1984

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, arrangement, and readability?	Yes ☐ No ☐	(If no, explain or suggest improvements under Comments, below.)

Comments:

FROM:

Name _____

Company _____

Address _____

_____

_____

FOLD FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 1070 CUPERTINO,CALIFORNIA

POSTAGE WILL BE PAID BY ADDRESSEE

Publications Manager
Hewlett-Packard Company
Computer Language Lab
19420 Homestead Road
Cupertino, California 95014

FOLD FOLD