HP 3000 Computer System

Systems Programming Language

Reference Manual

HEWLETT PACKARD

5303 STEVENS CREEK BLVD., SANTA CLARA, CALIFORNIA, 95050

Part No. 30000-90024
Product No. 32100A

Printed in U.S.A. 9/76
Update Incorporated 12/76

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

Page	Effective Date
Title	Jun 1976
ii	Jun 1976
iii to iv	Feb 1977
v	Jun 1976
vi	Feb 1977
vii.	Jun 1976
ix to xi	Jun 1976
1-1 to 1-14.	Jun 1976
2-1	Feb 1977
2-2	Jun 1976
2-3 to 2-4.	Feb 1977
2-5	Jun 1976
2-6 to 2-10.	Feb 1977
2-11	Jun 1976
2-12 to 2-14	Feb 1977
2-15	Jun 1976
3-1 to 3-4.	Jun 1976
3-5	Sep 1976
3-6 to 3-19.	Jun 1976
4-1 to 4-2.	Jun 1976
4-3	Sep 1976
4-4 to 4-15.	Jun 1976
4-16	Sep 1976
4-17 to 4-23	Jun 1976
4-24	Sep 1976
4-25	Jun 1976
4-26	Feb 1977
4-27 to 4-29	Jun 1976
5-1 to 5-20.	Jun 1976
6-1 to 6-7.	Jun 1976
6-8	Sep 1976
6-9	Jun 1976
6-10 to 6-12	Oct 1976
6-13 to 6-16	Jun 1976
7-1 to 7-27.	Jun 1976
8-1 to 8-7.	Jun 1976
8-8 to 8-9.	Sep 1976
8-10 to 8-46	Jun 1976
8-47	Feb 1977
8-48 to 8-51	Jun 1976
9-1 to 9-19.	Jun 1976
9-20	Feb 1977
10-1 to 10-12	Jun 1976
10-13	Sep 1976
10-14 to 10-18	Jun 1976
A-1.	Jun 1976
B-1	Jun 1976
C-1 to C-3	Jun 1976
D-1 to D-4	Jun 1976
E-1 to E-11	Jun 1976
F-1 to F-16	Jun 1976
Index-1 to Index-3.	Jun 1976

PRINTING HISTORY

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

First Edition	Jun 1976
Second Edition	Sep 1976
Update #1 Incorporated	Dec 1976
Update #2	Feb 1977

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.

This document contains proprietary information which is protected by copyright. All rights are reserved. No part of this document may be photocopied, reproduced or translated to another program language without the prior written consent of Hewlett-Packard Company

PREFACE

This publication is the reference manual for the HP 3000 Series II 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 Series II 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 instrinsic 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 — shows the syntax for SPL.

CONTENTS

Section I Page

SPL STRUCTURE

Section II Page

BASIC ELEMENTS

Section III Page

GLOBAL DATA DECLARATIONS

Section IV Page

EXPRESSIONS, ASSIGNMENT, AND SCAN STATEMENTS

Section V Page

PROGRAM CONTROL STATEMENTS

Section VI Page

MACHINE LEVEL CONSTRUCTS

CONTENTS (continued)

Section VII Page

PROCEDURES, INTRINSICS, AND SUBROUTINES

Section VIII Page

INPUT/OUTPUT

Section IX Page

COMPILER COMMANDS

Section X Page

MPE COMMANDS

CONTENTS (continued)

ILLUSTRATIONS

TABLES

SECTION I

SPL STRUCTURE

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.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	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!>>
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.

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.

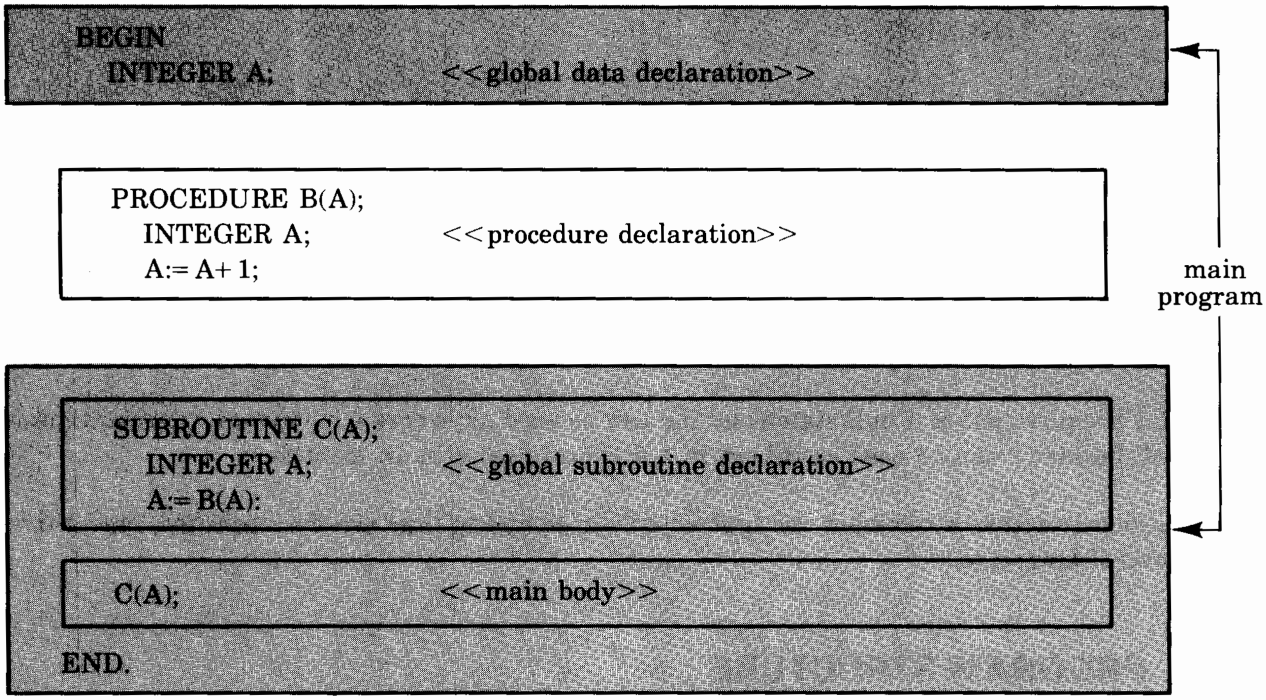
Additionally, SPL uses the stack architecture of the HP 3000 to handle both global and local variables. Global variables may be referenced anywhere in the program except in procedures where a local variable has the same identifier. Local variables are allocated memory locations upon entering a procedure 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,



```
BEGIN
  INTEGER A;              <<global data declaration>>

  PROCEDURE B(A);
    INTEGER A;            <<procedure declaration>>
    A:= A+ 1;

  SUBROUTINE C(A);
    INTEGER A;            <<global subroutine declaration>>
    A:= B(A):

  C(A);                   <<main body>>
END.
```

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.

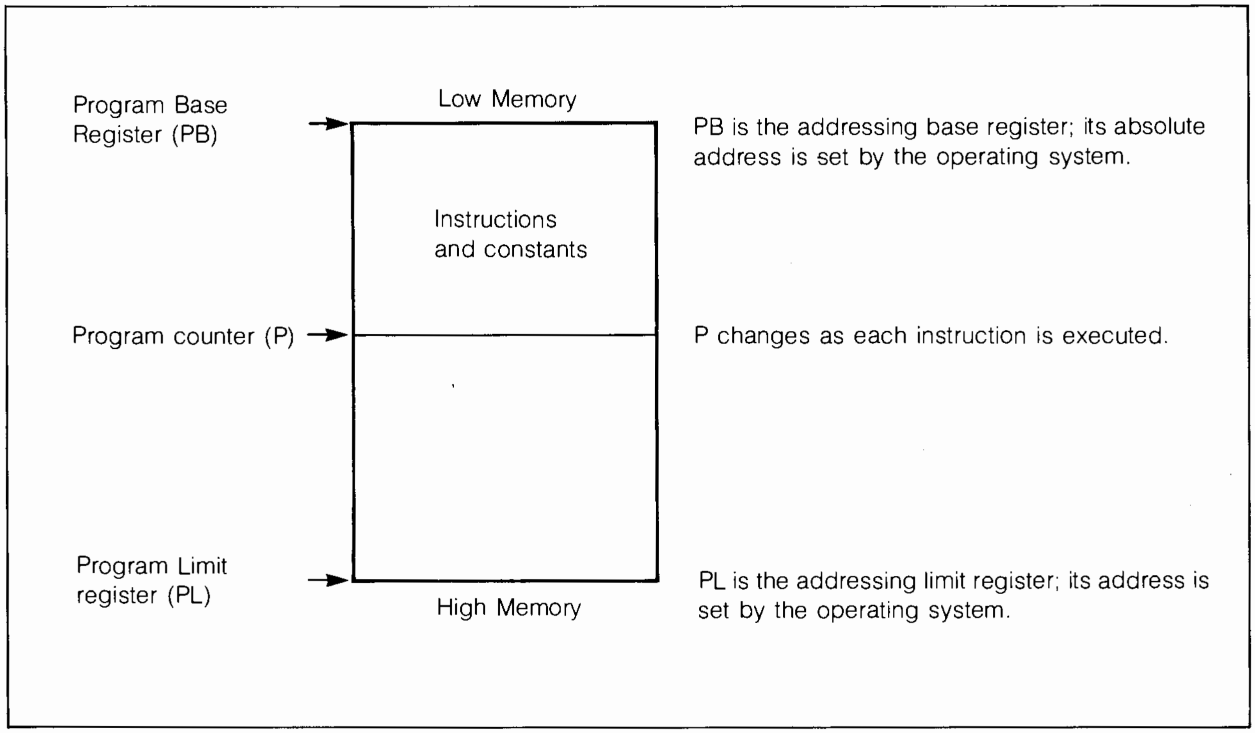
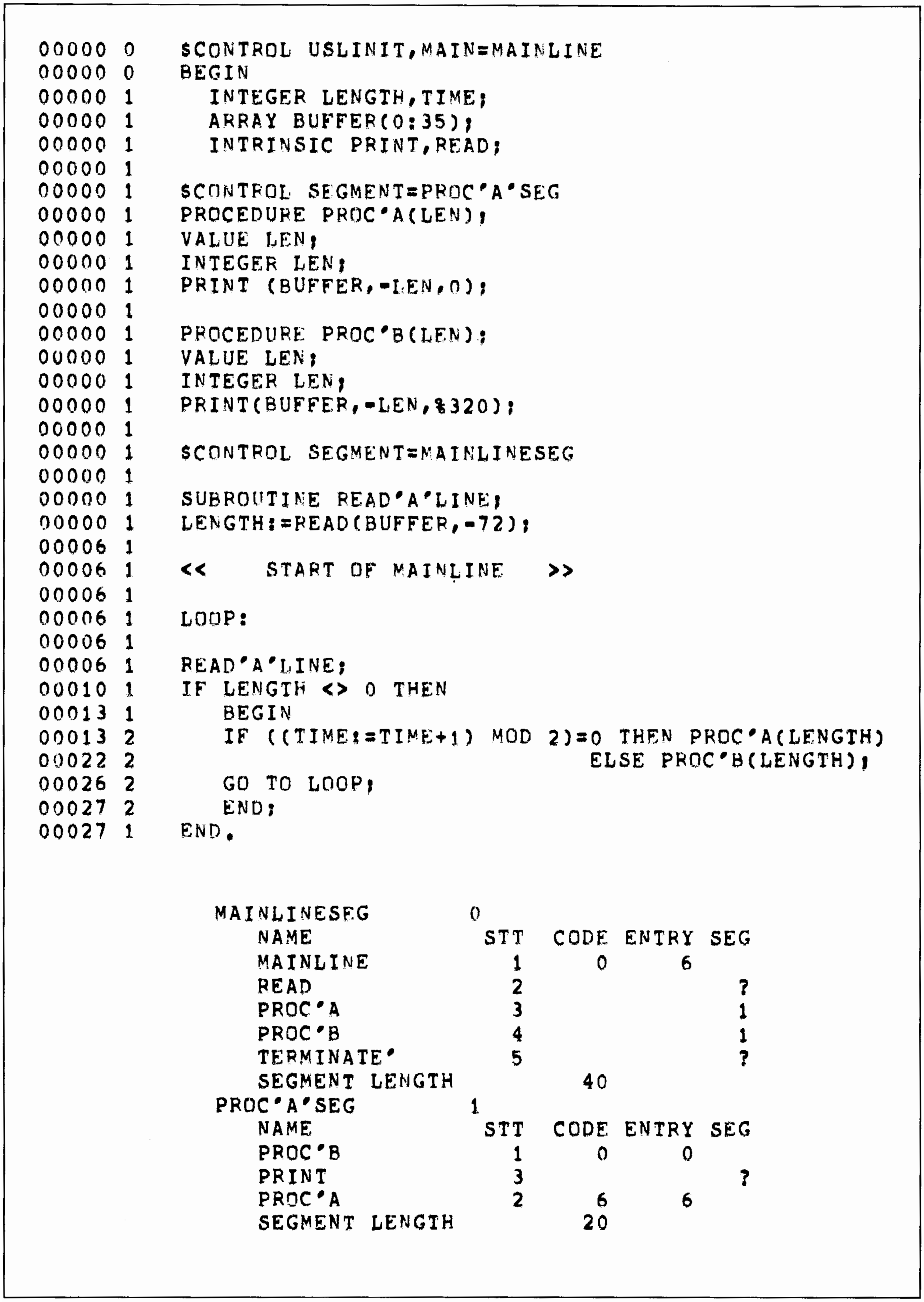

Program Base Register (PB)
Low Memory
PB is the addressing base register; its absolute address is set by the operating system.
Instructions and constants
Program counter (P)
P changes as each instruction is executed.
Program Limit register (PL)
High Memory
PL is the addressing limit register; its address is set by the operating system.

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 r ıst 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:

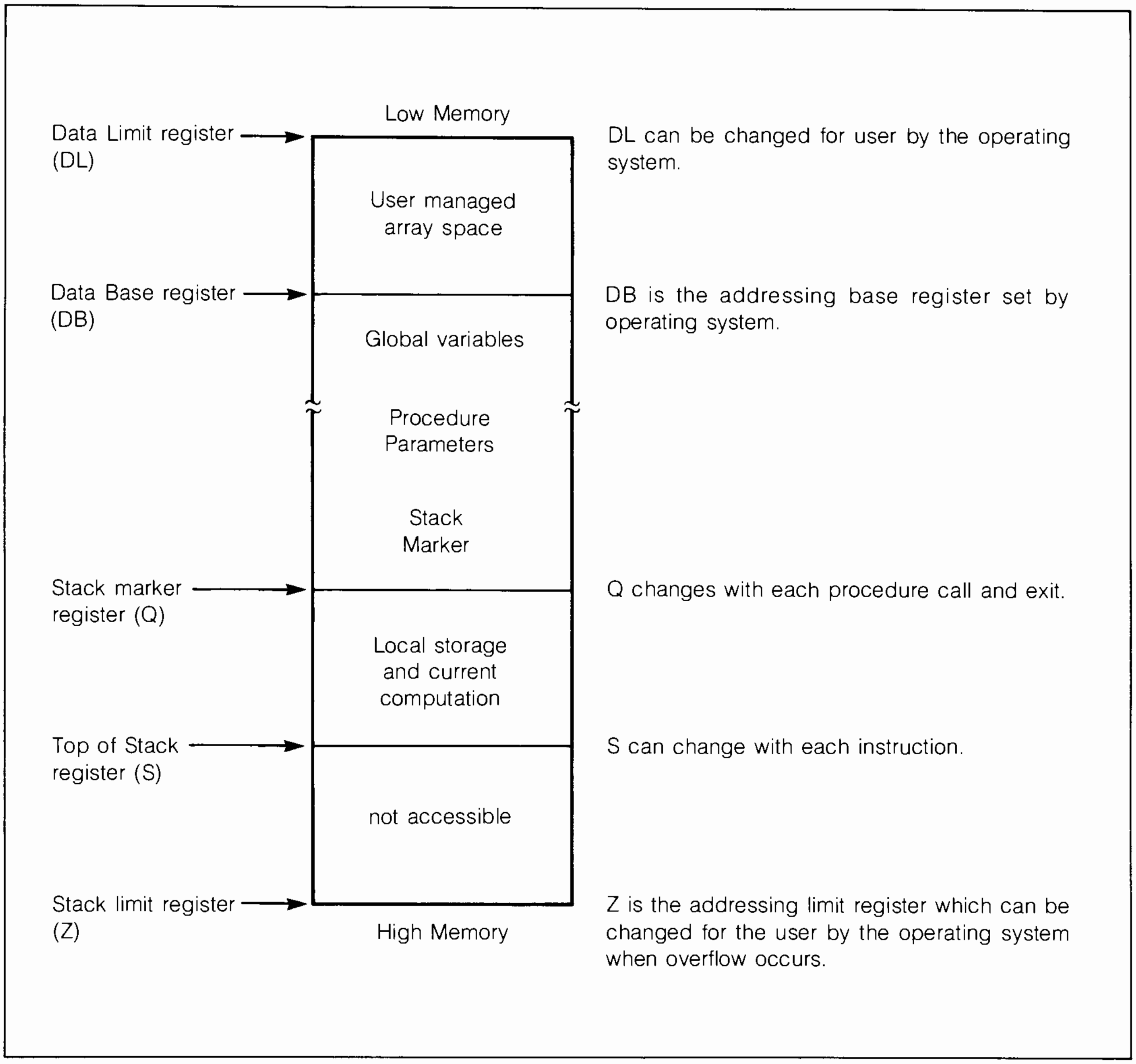
1. DB — Data Base register. Contains the absolute address of the first location of the direct address global area of the stack.

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 1-2. 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.



DL can be changed for user by the operating system.

DB is the addressing base register set by operating system.

Q changes with each procedure call and exit.

S can change with each instruction.

Z is the addressing limit register which can be changed for the user by the operating system when overflow occurs.

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

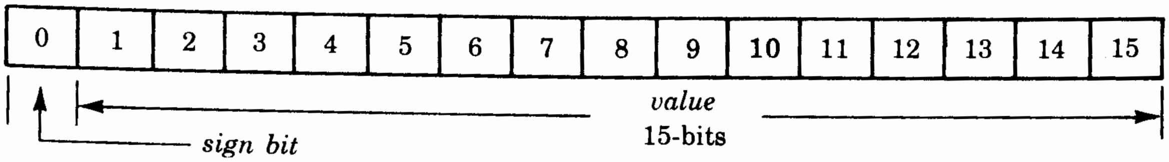
2-1. DATA STORAGE FORMATS

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

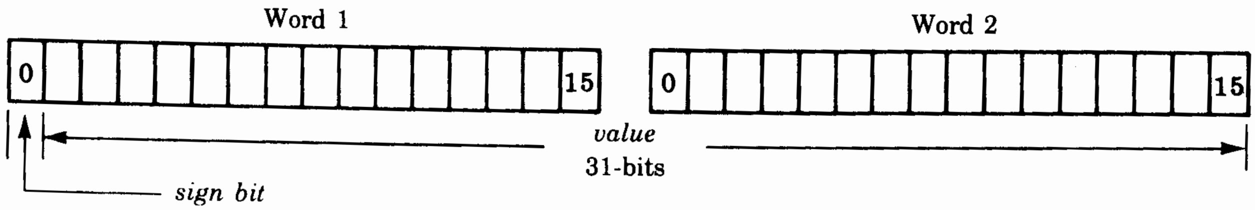
Integers are whole numbers containing no fractional part. Integer values are stored in one 16-bit computer word. The leftmost bit (bit 0) represents the arithmetic sign of the number (1=negative, 0=positive). The remaining 15 bits represent the binary value of the number. Integer numbers are represented 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:

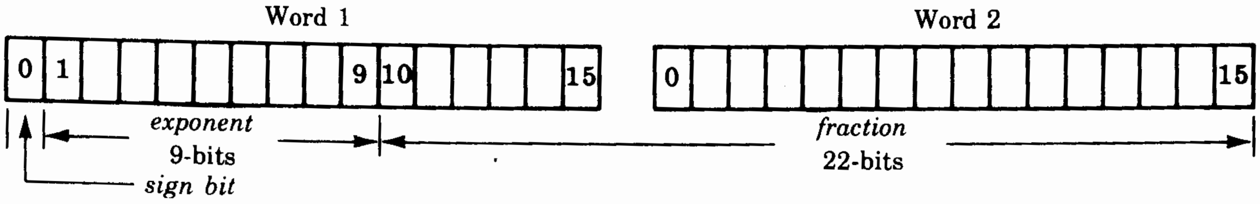
Sign(S)
Bit 0 of the first word (positive= 0, negative= 1). A value X and its negative, −X, differ only in the sign bit.

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:



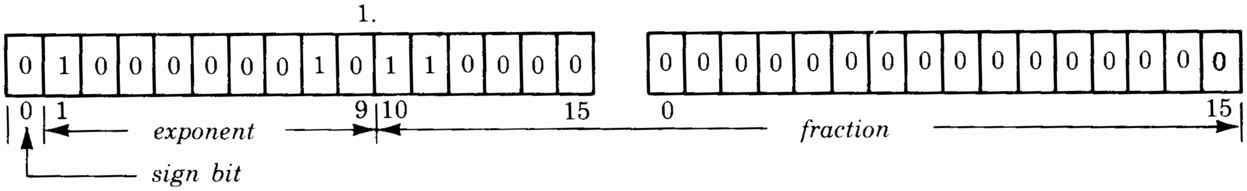
The formula for computing the decimal value of a floating-point representation is:

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

which is equivalent to:

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)

$$\begin{aligned} \text{Fraction (F)} = 1.11 \text{ (binary)} &= (1 \text{ x } 2^0) + (1 \text{ x } 2^{-1}) + (1,\text{x } 2^{-2}) \\ &= \quad 1 \quad + \quad 1/2 \quad + \quad 1/4 \\ &= 1.75 \text{ (decimal)} \end{aligned}$$

So, the decimal value of the real value is:

$$\begin{aligned} (-1)^0 \text{ x } 1.75 \text{ x } 2^{(258\ -\ 256)} &= 1 \text{ x } 1.75 \text{ x } 2^2 \\ &= 1.75 \text{ x } 4 \\ &= 7.0 \end{aligned}$$

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 $(-1)^S * (1.0 + (xxx * 2^{-38})) * 2^{(E-256)}$

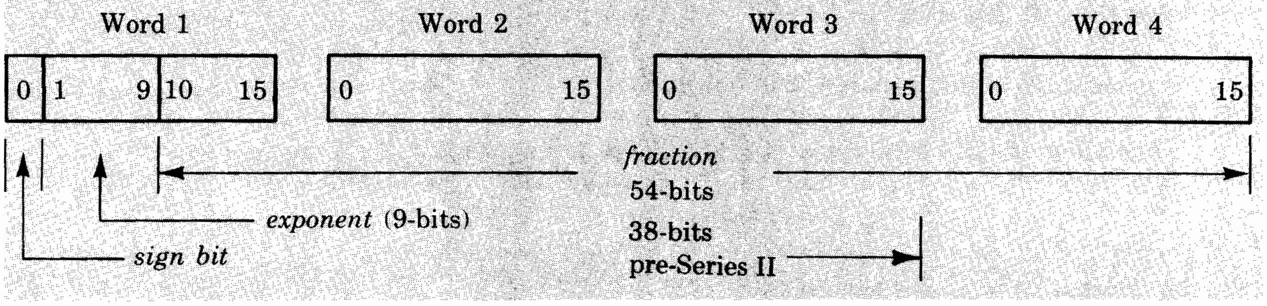
The range of the magnitude of non-zero long values is from $8.636168555094445 * 10^{-78}$ to $1.157920892373162 * 10^{77}$. Long numbers are accurate to 16.5 decimal places. The formula for computing the decimal value of a floating-point representation is:

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

which, for long values, is equivalent to:

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

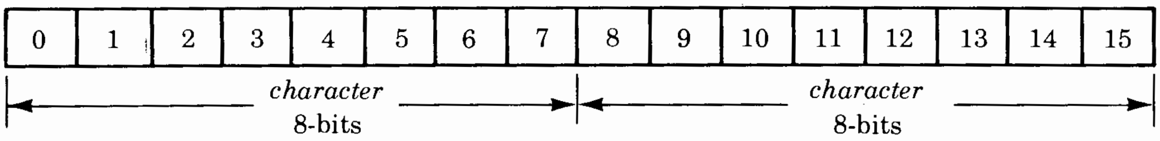
The internal representation for long numbers is:



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.

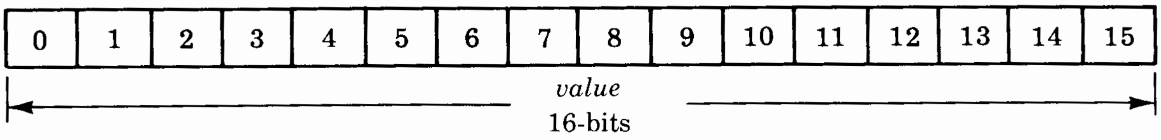
The internal representation of byte values is:



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 x 10^{78} to 1.157920892373162 x 10^{77}. The L construct is used to indicate the scaling factor, if any. For example, 2.5L–2 means 2.5 x 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).

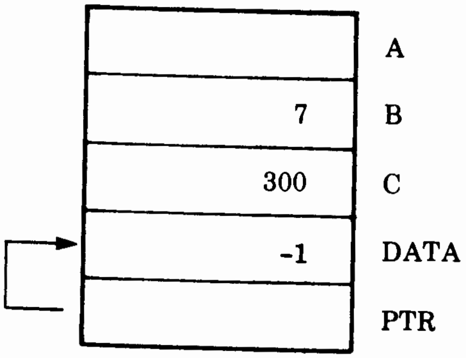
There are three contexts in which pointers can be used:

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.

For example, assume the following data declarations

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

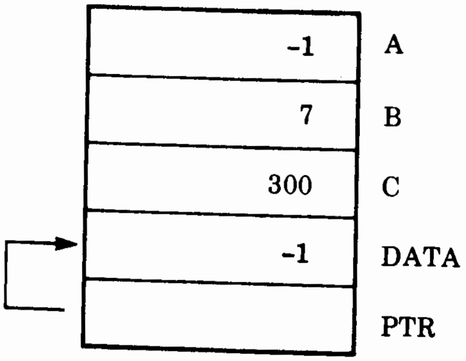
These declarations initialize the variables B, C, and DATA and set up PTR as a pointer to DATA as shown below.



Now, consider the statement

```
A:=PTR;
```

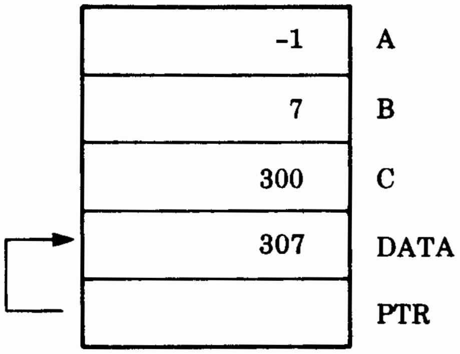
This statement assigns the object of the pointer PTR (i.e., DATA) to A.



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

```
    PTR:= B+ C;
```

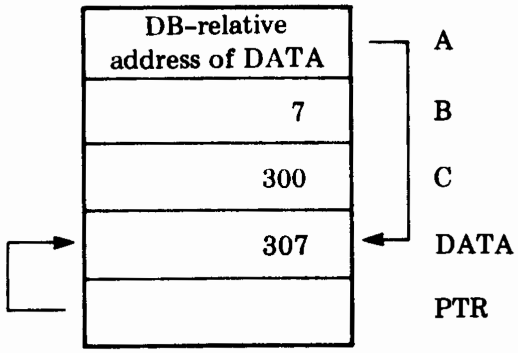
The object of the pointer PTR (i.e., DATA) is assigned the value of B+ C.



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;
```

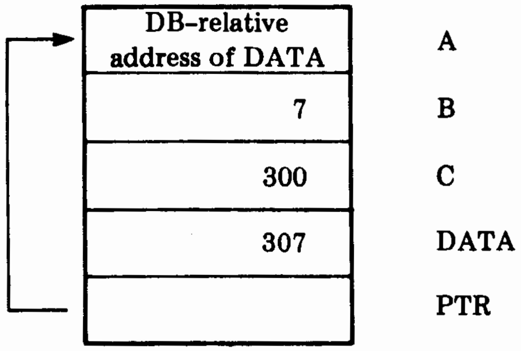
A is assigned the address contained in PTR (that is, the address of DATA).



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

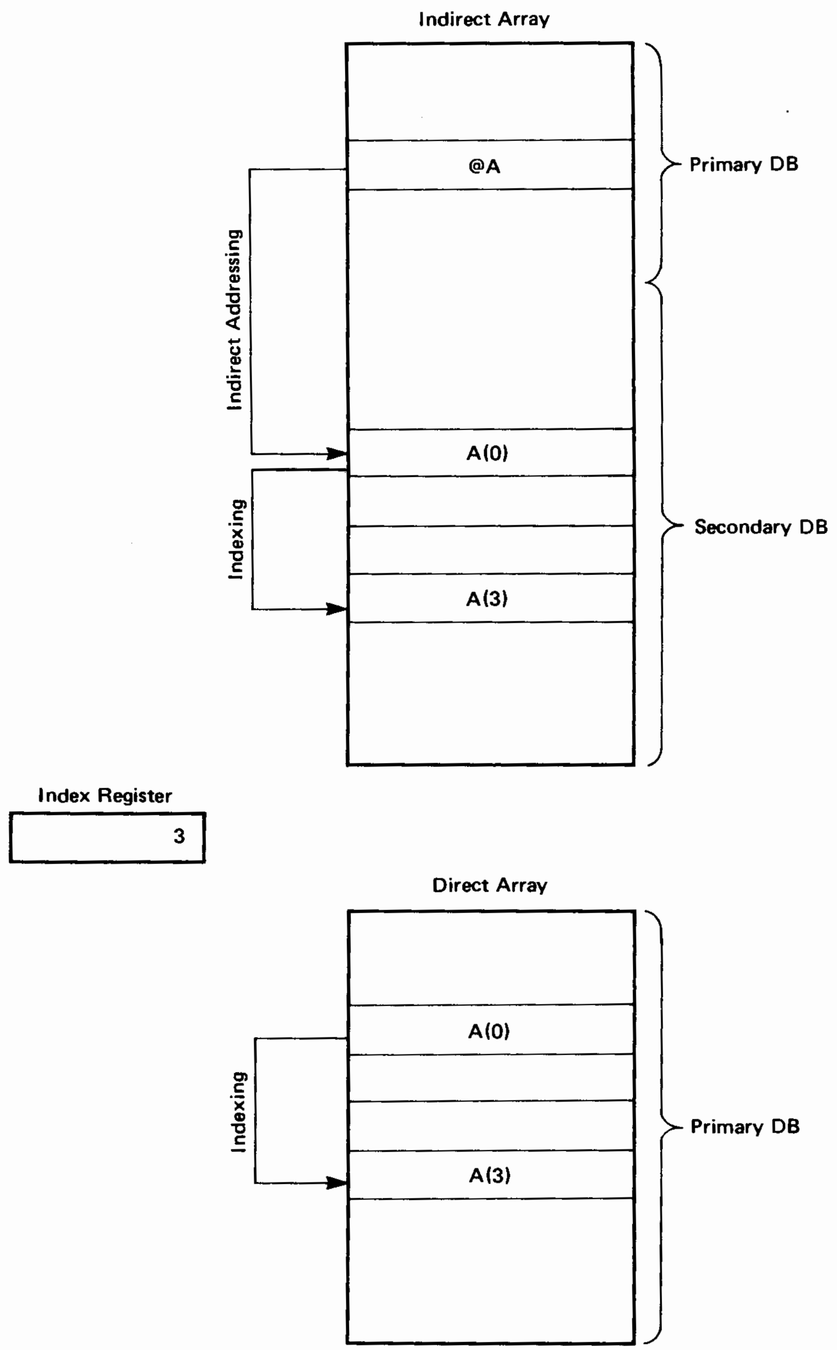


Figure 3-1. Accessing Array Elements

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,*]

$$\left\{\begin{array}{l}\textit{global-array-dec}\\ \textit{initialized-global-array-dec}\end{array}\right\};$$

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 DB, Q, or S-relative. If a base-register reference is not specfied, 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 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 DB+ 250, then the declaration

   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.

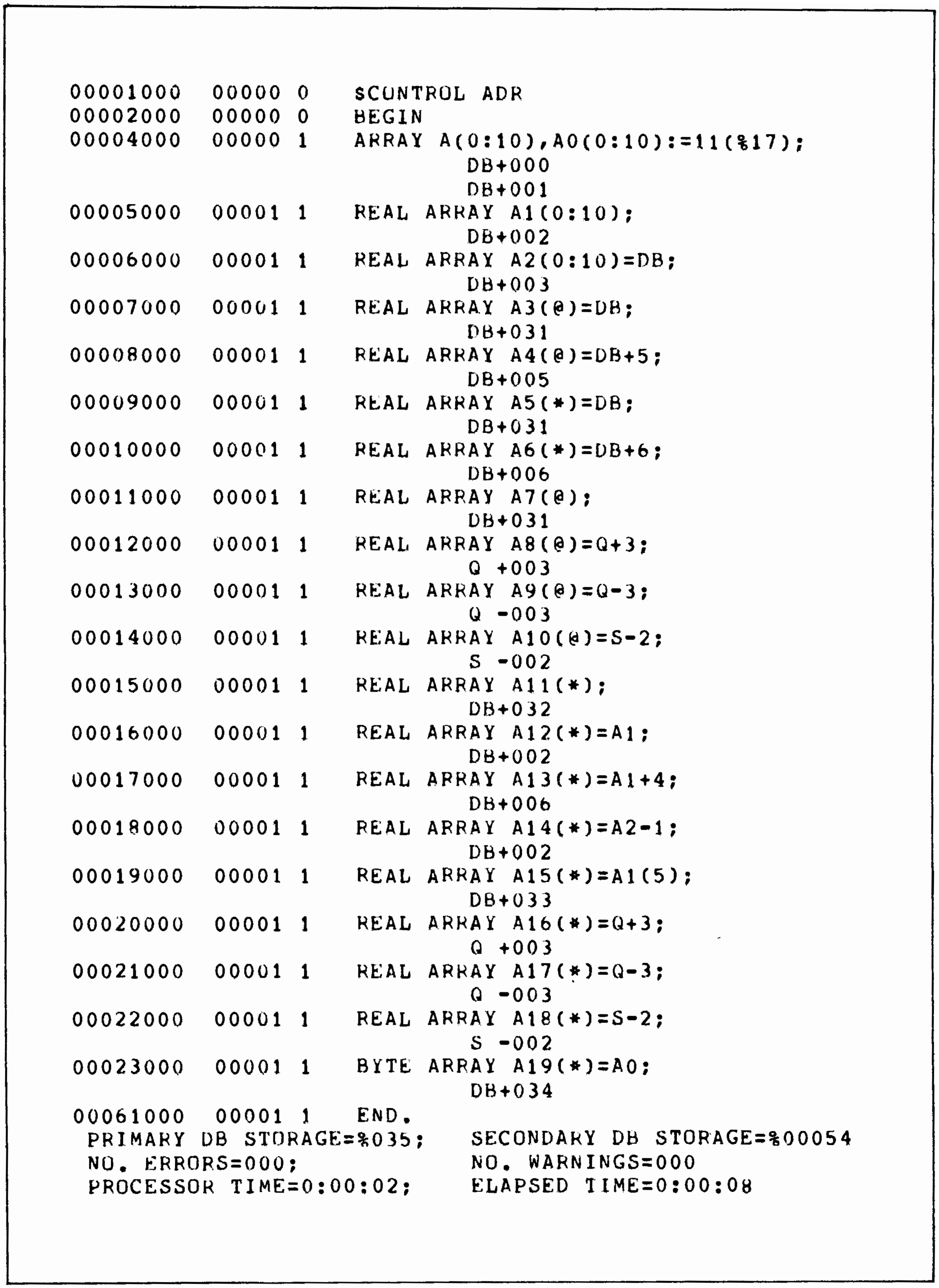
3-4. POINTER DECLARATION

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;
```

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.

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

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, not constants. The method is to follow the pointer with :=@ and a data reference (simple variable, pointer element, or array element). 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 wi!l 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>>
```

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. 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.

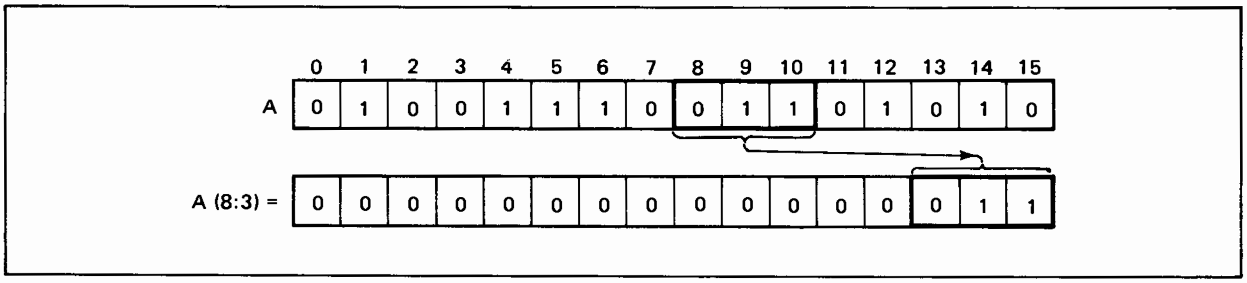
left-source-bit
specifies the bit of the source word at which the extraction begins. The *left-source-bit* is any unsigned decimal, based, composite, or equated integer constant from 0 to 15 inclusive.

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

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.*



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	0	1	0	0	1	1	1	0	0	1	1	0	1	0	1	0
A (8:3) =	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1

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.

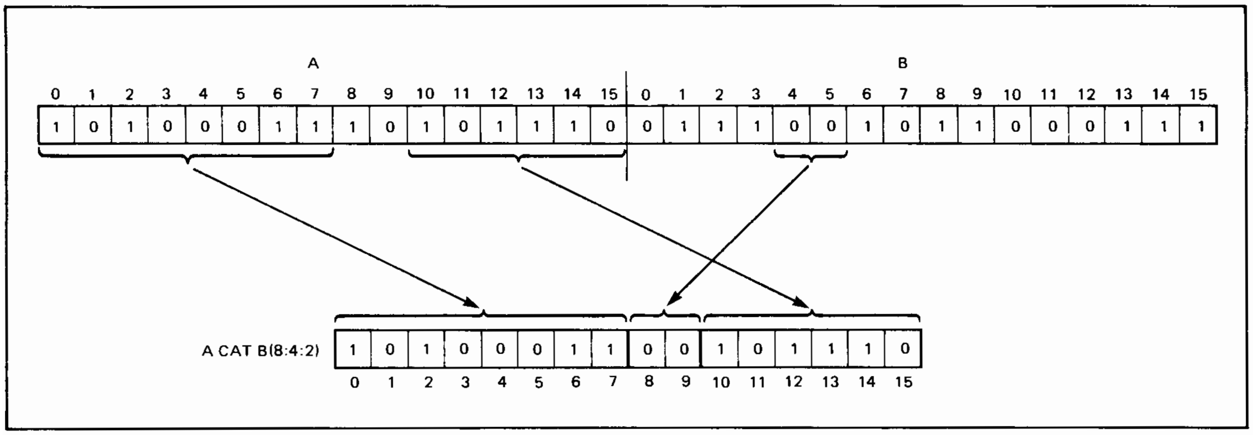


Figure 4-3. Bit Concatenation

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 in bit shifts; if you specify a triple shift on a single-word quantity, a triple shift is generated. You are responsible for maintaining compatibility. 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.

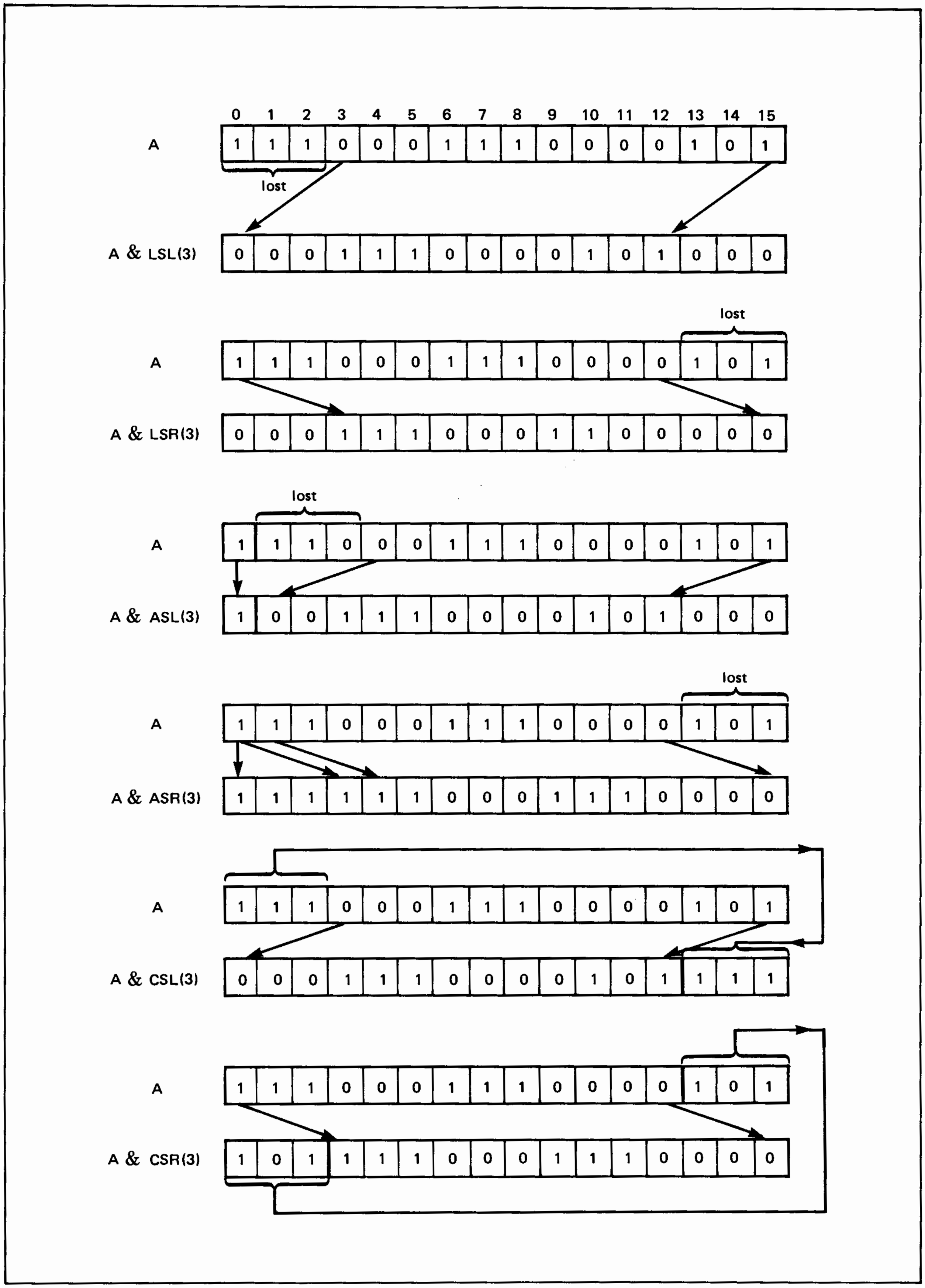

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
A 1 1 1 0 0 0 1 1 1 0 0 0 0 1 0 1
lost
A & LSL(3) 0 0 0 1 1 1 0 0 0 0 1 0 1 0 0 0
lost
A 1 1 1 0 0 0 1 1 1 0 0 0 0 1 0 1
A & LSR(3) 0 0 0 1 1 1 0 0 0 1 1 0 0 0 0 0
lost
A 1 1 1 0 0 0 1 1 1 0 0 0 0 1 0 1
A & ASL(3) 1 0 0 1 1 1 0 0 0 0 1 0 1 0 0 0
lost
A 1 1 1 0 0 0 1 1 1 0 0 0 0 1 0 1
A & ASR(3) 1 1 1 1 1 1 0 0 0 1 1 1 0 0 0 0
A 1 1 1 0 0 0 1 1 1 0 0 0 0 1 0 1
A & CSL(3) 0 0 0 1 1 1 0 0 0 0 1 0 1 1 1 1
A 1 1 1 0 0 0 1 1 1 0 0 0 0 1 0 1
A & CSR(3) 1 0 1 1 1 1 0 0 0 1 1 1 0 0 0 0

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)

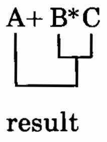
3. Multiply (*) and divide(/) for integer, real, byte, double, and long data.
   Modulo (MOD) or remainder for integer, byte, and double data.

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

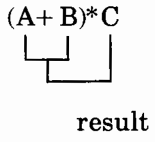
The order in which operations are performed is determined by this rank. For example,

$$\underbrace{\underbrace{A-B}+C}_{\text{result}}$$

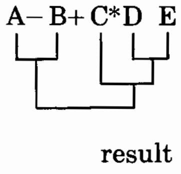
Operators of the same rank are performed from left to right.



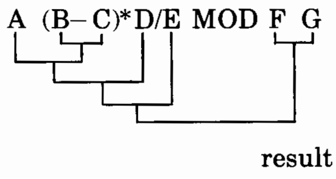
Operators of different rank are performed according to their position in the hierarchy of operators (highest rank first).



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*

integer-expressions

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 of 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 right-to-left. 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*]

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

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*])

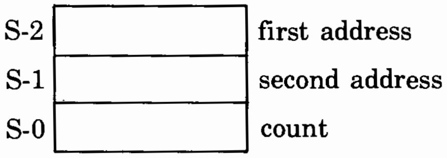
repetition-factor
specifies the number of times the constant list is used before going to the next *value-group*. The *repetition-factor* is an unsigned decimal, based, composite, or equated single-word integer constant.

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:

$$\textit{condition-term}\left[\begin{Bmatrix}\text{AND}\\\text{OR}\end{Bmatrix}\textit{condition-term}\ldots\begin{Bmatrix}\text{AND}\\\text{OR}\end{Bmatrix}\textit{condition-term}\right]$$

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.

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 (:=). 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 **, 11, 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
```

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. 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)

Move operations do not change the contents of the source.

The two forms of a move statement are:

```
                           ( source,(count)  )
                           ) * [PB],(count)  (
MOVE destination :=       <  string           >     [,stack-decrement]
                           ) (value-group)   (
                           (                 )
```

and

```
                          { source }
MOVE destination :=       { *      }     WHILE condition
```

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;
```

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*)]

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 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
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.

The *stack-decrement* is an integer constant equal to 0, 1, 2, or 3. This item specifies how many temporary values required by the move instruction are to be deleted from the stack after the move. If a *stack-decrement* is not specified, then all the temporary values are deleted.

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

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

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

S-1		destination address
S-0		source address

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

4-22. SCAN STATEMENT

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

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;
```

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.
"*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 two character string where the first character (bits 0 through 7) specifies the *terminal-character* and the second character (bits 8 through 15) specfies the *test-character*. In both cases, each byte in the 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 show below:

S-1		byte address
S-0		testword

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.

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, LOGICAL, or BYTE 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.

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*.

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.

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;
    ;
    C:=200
    END;
```

If J equals 1, control is transferred to the statement after the CASE statement.

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 is 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:

EXIT Instruction
Label address
Q

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

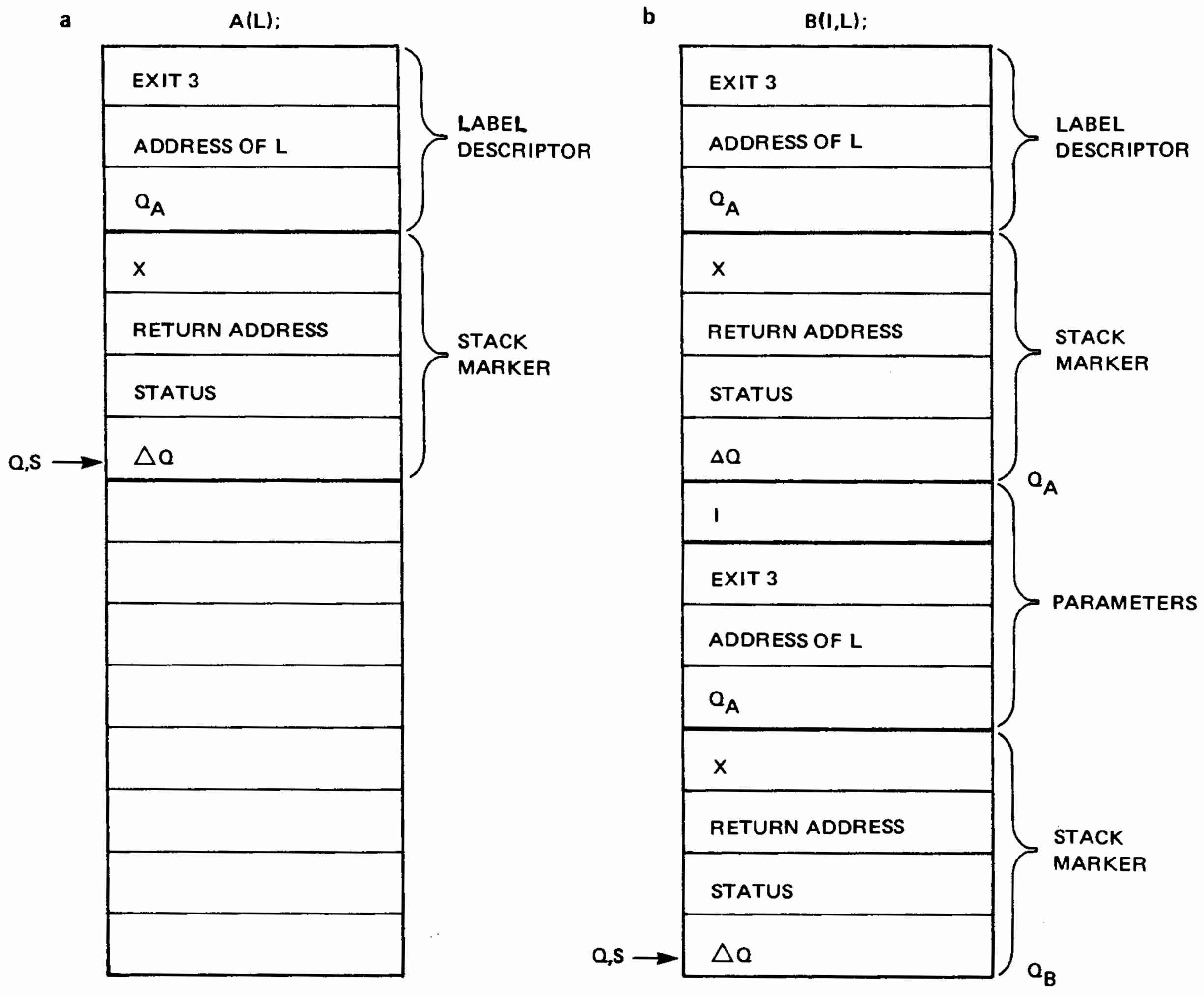


Figure 5-1. Passing a Label as a Parameter

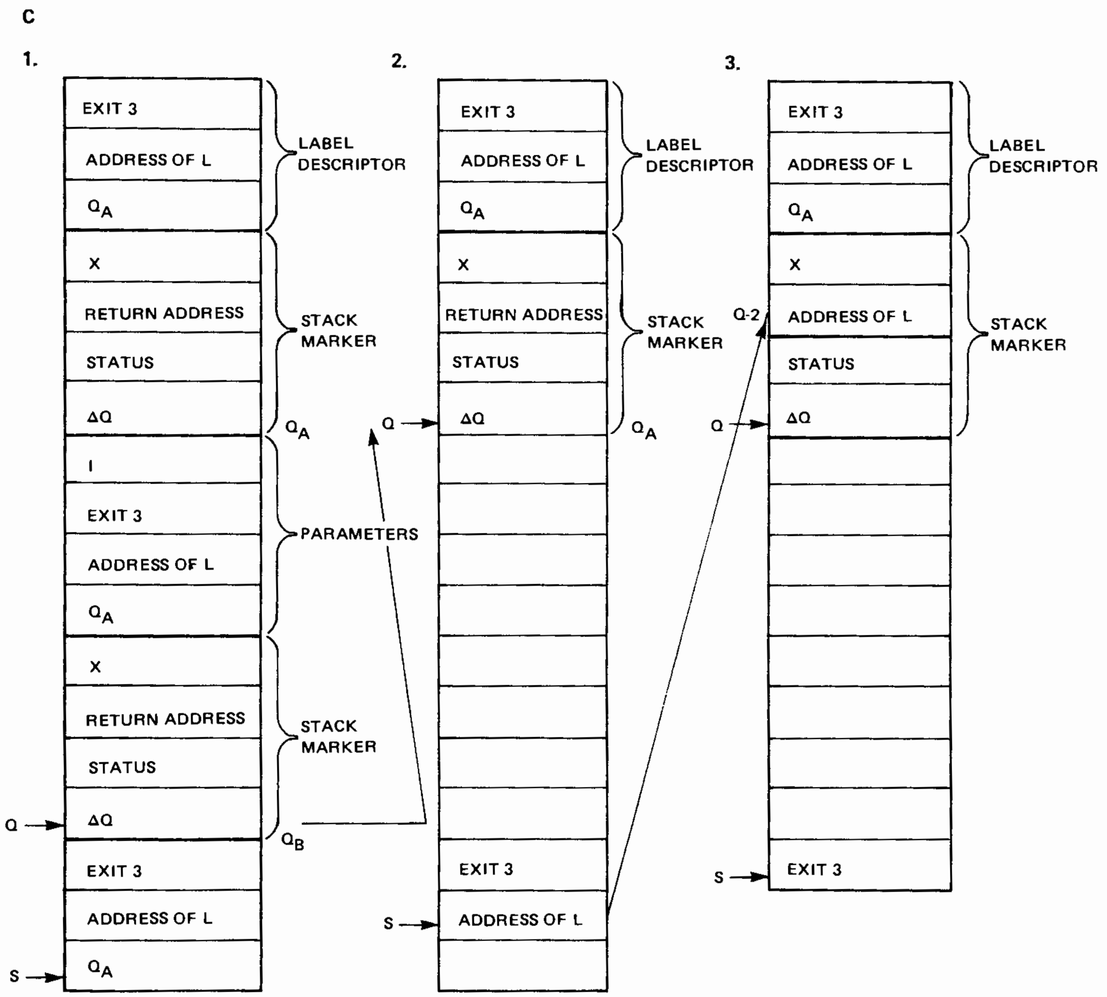


Figure 5-1. Passing a Label as a Parameter (Continued)

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. 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 transferred to 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.

Format 1

$$\text{1a}\ \left\{\begin{array}{l}\text{LOAD}\\ \text{LDX}\\ \text{LRA}\\ \text{CMPM}\\ \text{ADDM}\\ \text{SUBM}\\ \text{MPYM}\end{array}\right\}\ \left\{\begin{array}{l}\textit{label id}\\ \textit{variable id}\\ \text{DB} + \textit{usi}255\\ \text{P} + \textit{usi}255\\ \text{P} - \textit{usi}255\\ \text{Q} + \textit{usi}127\\ \text{Q} - \textit{usi}63\\ \text{S} - \textit{usi}63\end{array}\right\}\quad [,\text{I}]\ \ [,\text{X}]$$

$$\text{1b}\ \left\{\begin{array}{l}\text{LDB}\\ \text{LDD}\\ \text{STOR}\\ \text{STB}\\ \text{STD}\\ \text{INCM}\\ \text{DECM}\end{array}\right\}\ \left\{\begin{array}{l}\textit{variable id}\\ \text{DB} + \textit{usi}255\\ \text{Q} + \textit{usi}127\\ \text{Q} - \textit{usi}63\\ \text{S} - \textit{usi}63\end{array}\right\}\quad [,\text{I}]\ \ [,\text{X}]$$

$$\text{1c}\quad \text{BR}\ \left\{\begin{array}{l}\textit{label id}\\ \text{P} + \textit{usi}255\\ \text{P} - \textit{usi}255\end{array}\right\}\quad [,\text{I}]\ \ [,\text{X}]$$

$$\text{BR}\ \left\{\begin{array}{l}\text{DB} + \textit{usi}255\\ \text{Q} + \textit{usi}127\\ \text{Q} - \textit{usi}63\\ \text{S} - \textit{usi}63\end{array}\right\}\quad ,\text{I}\ \ [,\text{X}]$$

$$\text{1d}\ \left\{\begin{array}{l}\text{BL}\\ \text{BE}\\ \text{BLE}\\ \text{BG}\\ \text{BNE}\\ \text{BGE}\end{array}\right\}\ \left\{\begin{array}{l}\textit{label id}\\ \text{P} + \textit{usi}31\\ \text{P} - \textit{usi}31\end{array}\right\}\quad [,\text{I}]$$

$$\text{1e}\ \left\{\begin{array}{l}\text{TBA}\\ \text{MTBA}\\ \text{TBX}\\ \text{MTBX}\end{array}\right\}\ \left\{\begin{array}{l}\textit{label id}\\ \text{P} + \textit{usi}255\\ \text{P} - \textit{usi}255\end{array}\right\}$$

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
DELB
DDEL
XROX
INCX
DECX
ZERO
DZRO
DCMP
DADD
DSUB
MPYL
DIVL
DNEG
DXCH
CMP
ADD
SUB
MPY
DIV
NEG
TEST
STBX
DTST
DFLT
BTST
XCH
INCA
DECA
XAX
ADAX
ADXA
DEL
ZROB
LDXB
STAX
LDXA
DUP
DDUP
FLT
FCMP
FADD
FSUB
FMPY
FDIV
FNEG
CAB
LCMP
LADD
LSUB
LMPY
LDIV
NOT
OR
XOR
AND
FIXR
FIXT
INCB
DECB
XBX
ADBX
ADXB

For example,

```
ASSEMBLE(DDUP, DELB; STAX);
```

Format 3

$$3a\ \left\{\begin{array}{l}\text{IABZ}\\ \text{IXBZ}\\ \text{DXBZ}\\ \text{BCY}\\ \text{BNCY}\\ \text{CPRB}\\ \text{DABZ}\\ \text{BOV}\\ \text{BNOV}\\ \text{BRO}\\ \text{BRE}\end{array}\right\}\ \left\{\begin{array}{l}\textit{label}\\ \text{P} \pm \textit{usi}31\\ * \pm \textit{usi}31\end{array}\right\}\quad [,\text{I}]$$

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  }   usi63          [,X]
     DASR
     DLSL
     DLSR
     DCSL
     DCSR
     TBC
     TRBC
     TSBC
     TCBC
     QASL
     QASR
```

*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  }   usi255                    † = a privileged instruction for
     PSHR†                                   some registers
     LDNI
     LDXN
     CMPN
     SETR†

4b { EXF  }
     DPF  }    usi15 : usi15
```

For example,

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

Figure 6-1. Instruction Formats (Continued)

Format 5

$$\left\{\begin{array}{l}\text{RSW}\\ \text{LLSH}\dagger\\ \text{PLDA}\dagger\\ \text{PSTA}\dagger\\ \text{LSEA}\dagger\\ \text{SSEA}\dagger\\ \text{LDEA}\dagger\\ \text{SDEA}\dagger\\ \text{IXIT}\dagger\\ \text{LOCK}\dagger\\ \text{PCN}\dagger\\ \text{UNLK}\dagger\end{array}\right\}$$

† = a privileged instruction

For example,

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

Format 6

$$\left\{\begin{array}{l}\text{PAUS}\\ \text{SED}\\ \text{XCHD}\\ \text{SMSK}\\ \text{RMSK}\\ \text{XEQ}\\ \text{SIO}\\ \text{RIO}\\ \text{WIO}\\ \text{TIO}\\ \text{CIO}\\ \text{CMD}\\ \text{SIN}\\ \text{HALT}\\ \text{LST}\\ \text{PSDB}\\ \text{DISP}\\ \text{PSEB}\\ \text{SCLK}\\ \text{RCLK}\\ \text{SST}\end{array}\right\} \quad usi15$$

For example,

ASSEMBLE (XEQ 4);

All of these instructions except XEQ and RMSK are privileged.

Figure 6-1. Instruction Formats (Continued)

Format 7

$$\left\{\begin{array}{l}\text{PCAL}\\\text{SCAL}\\\text{EXIT}\\\text{SXIT}\\\text{ADXI}\\\text{SBXI}\\\text{LLBL}\\\text{LDPP}\\\text{LDPN}\\\text{ADDS}\\\text{SUBS}\\\text{ORI}\\\text{XORI}\\\text{ANDI}\end{array}\right\} \quad \mathit{usi}255$$

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

For example,

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

Format 8

8a $$\left\{\begin{array}{l}\text{MOVE}\\\text{MVB}\\\text{CMPB}\end{array}\right\} \quad \text{[PB]} \quad \left[\begin{array}{l},0\\,1\\,2\\,3\end{array}\right]$$

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

8b $$\text{MVBW} \quad \left\{\begin{array}{l}\text{A}\\\text{N}\\\text{AN}\\\text{AS}\\\text{ANS}\end{array}\right\} \quad \left[\begin{array}{l},0\\,1\\,2\end{array}\right]$$

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

8c $$\left\{\begin{array}{l}\text{MVBL}\dagger\\\text{MVLB}\dagger\\\text{SCW}\\\text{SCU}\end{array}\right\} \quad \left[\begin{array}{l},0\\,1\\,2\end{array}\right]$$

†Privileged instruction.

Figure 6-1. Instruction Formats (Continued)

If item two is missing, the stack decrement is 2. For example,

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

$$\text{8d} \left\{\begin{array}{l}\text{MABS}\dagger\\ \text{MTDS}\dagger\\ \text{MDS}\dagger\\ \text{MFDS}\dagger\end{array}\right\} \left[\begin{array}{l}0\\ 1\\ 2\\ 3\\ 4\\ 5\ \text{ for MABS and}\\ \qquad\text{MDS}\end{array}\right]$$

Format 9

CON *constant list*

This format is actually a psuedo-mnemonic for constant generation; it is not a hardware instruction.

CON stores a series of constants in the code starting at the current location. In addition to all numerical and string constants, P relative address constants can be created by listing label identifiers (this is used to create addresses for indirect references). The CON instruction itself can be labeled so that other instructions can reference the constants symbolically.

```
ASSEMBLE(
     BR P + 1, I;
     CON LABELNAME );
ASSEMBLE (TAB: CON "ABCDEFGH"; . . . . .
     LDB TAB, X; . . . . . . . . . );
```

Format 10

10a DMUL
DDIV
EADD
ESUB
EMPY
EDIV
ENEG
ECMP
DMPY

$$\text{10b} \left\{\begin{array}{l}\text{CVAD}\\ \text{CVBD}\end{array}\right\} \left[\begin{array}{l}0\\ 1\end{array}\right]$$

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

$$\text{CVDB}\begin{bmatrix}0\\1\end{bmatrix}$$

If item 2 is 0, 2 words are deleted from the stack.
If item 2 is 1 or empty, 3 words are deleted from the stack.

10d

$$\left\{\begin{array}{l}\text{ADDD}\\\text{SUBD}\\\text{MPYD}\\\text{CMPD}\\\text{SLD}\\\text{NSLD}\\\text{SRD}\end{array}\right\}\begin{bmatrix}0\\1\\2\end{bmatrix}$$

If item 2 is 0, no words are deleted from the stack.
If item 2 is 1, 2 words are deleted from the stack.
If item 2 is 2 or empty, 4 words are deleted from the stack.

10e

$$\text{CVDA}\begin{bmatrix}\begin{array}{l}0\\1\\\text{ABS}\\\text{NABS}\end{array} & \begin{bmatrix},\left\{\begin{array}{c}0\\1\end{array}\right\}\end{bmatrix}\end{bmatrix}$$

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
BCY	Branch on carry		3a
BE	Branch on equals	See BCC	
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		
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 de~~cimal 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

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
DUP	Duplicate A	2
DXBZ	Decrement X, branch if zero	3a
DXCH	Double exchange	2
DZRO	Double push zero	2
EADD	Extended-precision floating point add	10a
ECMP	Extended-precision floating point compare	10a
EDIV	Extended-precision floating point divide	10a
EMPY	Extended-precision floating point multiply	10a
ENEG	Extended-precision floating point negate	10a
ESUB	Extended-precision floating point subtract	10a
EXF	Extract field	4b
EXIT	Procedure and interrupt exit	7
FADD	Floating add	2
FCMP	Floating compare	2
FDIV	Floating divide	2
FIXR	Fix and round	2
FIXT	Fix and truncate	2
FLT	Float	2
FMPY	Floating multiply	2
FNEG	Floating negate	2
FSUB	Floating subtract	2
HALT	Halt	6
IABZ	Increment A, branch if zero	3a
INCA	Increment A	2
INCB	Increment B	2
INCM	Increment memory	1b
INCX	Increment index register	2
IXBZ	Increment X, branch if zero	3a
IXIT	Interrupt exit	5
LADD	Logical add	2

Shaded instructions not available in pre-Series II systems.

Table 6-1. Machine Instruction Mnemonics (Continued)

ALPHABETIC LISTING OF INSTRUCTIONS

MNEMONIC	FUNCTION	FORMAT
LCMP	Logical compare	2
LDB	Load byte	1b
LDD	Load double	1b
LDEA	Load double word from extended address	5
LDI	Load immediate	4a
LDIV	Logical divide	2
LDNI	Load negative immediate	4a
LDPN	Load double from program, negative	7
LDPP	Load double from program, positive	7
LDX	Load Index	1a
LDXA	Load X onto stack	2
LDXB	Load X into B	2
LDXI	Load X immediate	4a
LDXN	Load X negative immediate	4a
LLBL	Load Label	7
LLSH	Linked list search	5
LMPY	Logical multiply	2
LOAD	Load	1a
LOCK	Lock resource	5
LRA	Load relative address	1a
LSEA	Load single word from extended address	5
LSL	Logical shift left	3b
LSR	Logical shift right	3b
LST	Load from system table	6
LSUB	Logical subtract	2
MABS	Move using absolute address	8
MDS	Move using data segment	8
MFDS	Move from data segment	8
MOVE	Move words	8a
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

Shaded instructions not available in pre-Series II systems.

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
RCLK	Read clock	6
RIO	Read I/O	6
QASL	Quadruple arithmetic shift left	3b
QASR	Quadruple arithmetic shift right	3b
RMSK	Read mask	6
RSW	Read switch register	5
SBXI	Subtract immediate from X	7
SCAL	Subroutine call	7
SCAN	Scan bits	3b
SCLK	Store clock	6
SCU	Scan until	8c
SCW	Scan while	8c
SDEA	Store double word into extended address	5
SED	Set enable/disable external interrupts	6
SETR	Set registers	4a
SIN	Set interrupt	6
SIO	Start I/O	6
*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
SUB	Subtract	2
SUBD	Subtract decimal	10d
SUBI	Subtract immediate	4a
SUBM	Subtract memory	1a
SUBS	Subtract from S	7
SXIT	Subroutine exit	7
TASL	Triple arithmetic shift left	3b
TASR	Triple arithmetic shift right	3b
TBA	Test, branch, A	1e
TBC	Test bit and set condition code	3b
TBX	Test, branch, X	1e
TCBC	Test and complement bit and set CC	3b
TEST	Test TOS	2
TIO	Test I/O	6
TNSL	Triple normalizing shift left	3b
TRBC	Test and reset bit, set condition code	3b
TSBC	Test, set bit, set condition code	3b
*TSBM	Test and set bit in memory	3b
UNLK	Unlock resource	5
WIO	Write I/O	6
XAX	Exchange A and X	2
XBX	Exchange B and X	2
XCH	Exchange A and B	2

*Instructions preceded by asterisk only available in pre-Series II systems.
Shaded instructions not available in pre-Series II systems.

Table 6-1. Machine Instruction Mnemonics (Continued)

ALPHABETIC LISTING OF INSTRUCTIONS

MNEMONIC	FUNCTION	FORMAT
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.

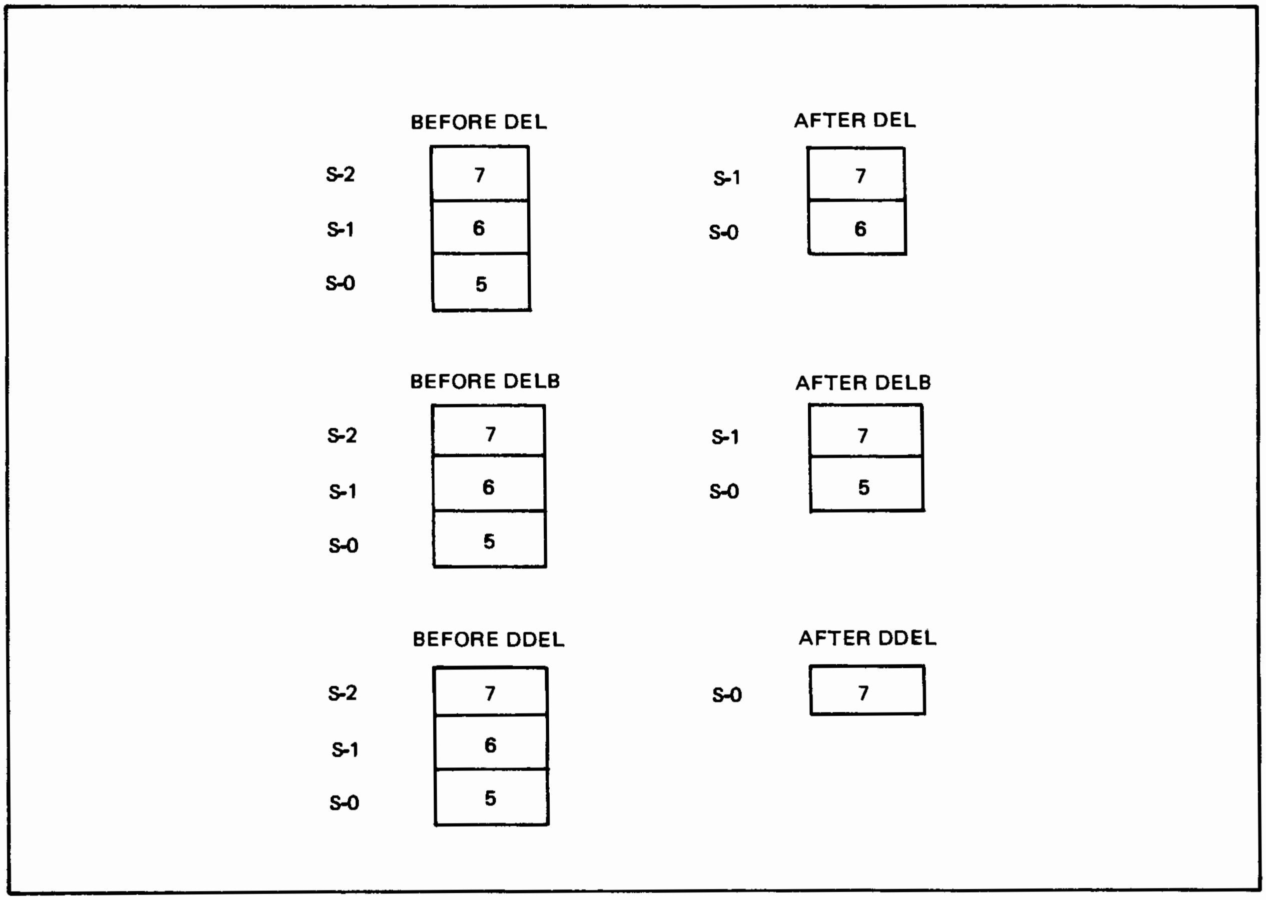


Figure 6-2. Delete Statement

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.

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. 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.

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.

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 DB area, they are not deleted when a procedure exits, but are still in existence, with their last value, when the procedure is called again. 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.
- 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,*] $\left\{ \begin{array}{l} \textit{local-array-dec} \\ \textit{constant-array-dec} \end{array} \right\}$;

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*** $\left\{\begin{matrix}(*)\\(@)\end{matrix}\right\}\left[\text{[,...,}\textit{array-name}\left\{\begin{matrix}(*)\\(@)\end{matrix}\right\}\right]$;

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.

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.

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

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, not constants. The method is to follow the pointer with :=@ and a data reference (simple variable, pointer element, or array element). 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 instruction 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;
```

SECTION VIII INPUT/OUTPUT

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 discusses 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.*

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.

Each intrinsic description discussed in this section includes the following information:

- The intrinsic name and a brief summary of its function.

- The complete intrinsic call description as shown for the READ intrinsic:

  $$\overset{I}{\mathit{lgth}}{:=}\mathrm{READ}(\overset{A}{\boldsymbol{message}},\ \overset{IV}{\boldsymbol{expectedl}});$$

  Required parameters such as ***message*** are shown in ***bold face italics;*** optional parameters and return values such as *lgth* are shown in *reqular italics*. Superscripts are used to describe the types of parameters and whether they must be passed by value instead of by reference. The superscripts have the following meanings:

A	Array (Logical)
BA	Byte array
BP	Byte pointer
D	Double
DA	Double array
DV	Double value
I	Integer
IA	Integer array
IV	Integer value
L	Logical
LV	Logical value
O-V	OPTION VARIABLE

An O-V superscript at the end of the parameter list indicates that some or all of the parameters are optional.

- FUNCTIONAL RETURN: For those intrinsics which return a value to the calling program (function procedures), the return value is described. If the intrinsic is not a function procedure, this portion of the description is omitted.

- PARAMETERS: All parameters are described. In the intrinsic call description, required parameters are shown in bold face italics.

- CONDITION CODES: The Condition Codes are included for each intrinsic:

  CCL (Condition Code set to Less than)
  CCE (Condition Code set to Equals)
  CCG (Condition Code set to Greater than)

- 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 sytsem failures. If you are not a privileged mode user, you need not concern yourself with this restriction and you may assume that all intrinsics described in this section will not operate in split stack mode unless otherwise stated.

8-2. FOPEN

The FOPEN intrinsic opens a file.

```
   I                       BA       LV       LV     IV   BA
filenum:= FOPEN(formaldesignator,foptions,aoptions,recsize,device,

     BA        IV           IV          IV    DV
formmsg,userlabels,blockfactor,numbuffers,filesize,

     IV          IV         IV   O-V
numextents,initialloc,filecode);
```

The FOPEN intrinsic makes it possible to access a file. In the FOPEN intrinsic call, a particular file is referenced by its formal file designator, described later in this section. When the FOPEN intrinsic is executed, it returns to the user's process a file number by which the system uniquely identifies the file. This file number, rather than the file designator, then is used by subsequent intrinsics in referencing the file.

FUNCTIONAL RETURN:

This intrinsic returns an integer file number used to identify the opened file in other intrinsic calls.

PARAMETERS:

formaldesignator byte array (optional)
Contains a string of ASCII characters interpreted as a formal file designator, as defined in the *MPE Commands Reference Manual.* This string must begin with a letter, contain alphanumeric characters, slashes, or periods, and terminate with any non-alphanumeric character except a slash or a period. If the string names a system-defined file, it can begin with a dollar sign ($); if it names a user pre-defined file, it can begin with an asterisk (*). Default: A temporary file that can be read or written on, but not saved, is assigned.

foptions logical value (optional)
The foptions parameter allows you to specify six different file characteristics, by setting corresponding bit groupings in a 16-bit word. The correspondence is from right to left, beginning with bit 15. These characteristics are as follows, proceeding from the rightmost bit groups to the leftmost bit groups in the word. The bit settings are summarized in figure 8-1.

NOTE

Bit groups are denoted using the standard SPL notation. Thus, bits (14:2) indicates bits 14 and 15; bits (10:3) indicates bits 10,11, and 12.

Bits (14:2) - Domain Foption

The file domain to be searched by MPE to locate the file, indicated by these bit settings:

00 = The file is a new file, created at this point. No search is necessary.

01 = The file is an old permanent file, and the system file domain should be searched.

10 = The file is an old temporary file, and the job file domain should be searched.

11 = The file is an old file that is to be located by first searching the job file domain and then, if the file is not found, by searching the system file domain.

Bit (13:1) - ASCII/Binary *Foption*

The code (ASCII or Binary) in which a new file is to be recorded when it is written to a device that supports both codes. In the case of disc files, this also affects padding than can occur when a direct-write intrinsic call (FWRITEDIR) is issued to a record that lies beyond the current logical end-of-file indicator. In ASCII files, any dummy records between the previous end-of-file and the newly-written records are padded with blanks. In binary files, such records are padded with binary zeros. All files not on disc are treated as ASCII files.

For ASCII files, this bit is 1.

For Binary files, this bit is 0.

Bits (10:3) - Default File Designator *Foption*.

The actual file designator equated with the formal file designator specified in FOPEN, if

1. No explicit or implicit :FILE command equating the formal file designator to a different actual file designator occurs in the job or session; or
2. The Disallow File Equation *Foption* (bit 5) is specified.

The bit settings are:

000 = The actual file designator is the same as the formal file designator.

001 = The actual file designator is $STDLIST.

010 = The actual file designator is $NEWPASS.

011 = The actual file designator is $OLDPASS.

100 = The actual file designator is $STDIN.

101 = The actual file designator is $STDINX.

110 = The actual file designator is $NULL.

Bits (8:2) - Record Format *Foption*.

The format in which the records in the file are recorded, indicated by these bit settings:

00 = Fixed-length records. The file is composed of logical records of uniform length.

01 = Variable-length records. The file contains logical records of varying length. This format is restricted to records that are written in sequential order. The size of each record is recorded internally. The actual record size used is determined by multiplying the *recsize* (specified or default) by the *blockfactor*, and adding two words reserved for system use. This option is not allowed when NOBUF is specified. In such a case, the record format used is undefined-length records, discussed below.

10 = Undefined-length records. The file contains records of varying length that were not written using the variable length *foption* (01). All files not on disc or magnetic tape are treated as containing undefined-length records.

Bit (7:1) - Carriage Control *Foption*.

If selected, this specifies that you will supply a carriage control character in the calling sequence of each FWRITE call that writes records onto the file.

0 = No carriage control character expected.

1 = Carriage control character expected.

Carriage control will be defined only for character oriented, i.e., ASCII, files. This option and binary are mutually exclusive and attempts to open new files with both binary and this option will result in an access violation.

This option is a physical attribute of the file and its state cannot be modified when opening an old disc file.

A carriage control character passed through the *control* parameter of FWRITE will be recognized and acted upon only for files for which carriage control is specified in FOPEN. Embedded control will be treated strictly as data on files for which no carriage control is specified, and will not invoke spacing for such files. You may specify spacing action on files for which carriage control has been specified, either by embedding the control in the record, indicated with a *control* parameter of one in the call to FWRITE, or by sending the control code directly via the *control* parameter of FWRITE.

A carriage control character sent to a file on which the control cannot be executed directly, for example, line spacing to a disc or tape file, will result in having the control character embedded as the first byte of the record. Thus, the first byte of each record in a disc file having a carriage control character will be control information. Control sent to other types of files results in transmission of the control to the driver.

The control codes %400 through %403 will be remapped to %100 through %103 so that they will fit into one byte and thus can be embedded. Records written to the line printer with one of the above controls should not contain information other than control information. In order that the FWRITE of such a control record acts the same as the equivalent call to FCONTROL, the printer driver will execute the control portion of the record and will not transfer any other data included in the record.

For the purpose of computing record size, carriage control information will be considered by the file system to be part of the data record. As such, specifying the carriage control option will add one byte to the record size at the time the file is created. For example, a specification of REC= – 132,1,F,ASCII;CCTL will result in a *recsize* of 133 characters.

You always may read up to and including the *recsize* as returned by the FGETINFO intrinsic. On writes of files for which carriage control is specified, however, the data transferred is limited to *recsize*-1 unless a control of one is passed indicating the data record is prefixed with embedded control.

Bit (6:1) - Reserved for MPE. Should be set to zero.

Bit (5:1) - Disallow File Equation.

This option ignores any corresponding :FILE command, so that the specifications in the FOPEN call take effect (unless preempted by those in the file label, for disc files).

0 = Allow :FILE.

1 = Disallow :FILE.

Bits (0:5) Reserved for MPE. Should be set to zero.

Default: All bits are set to zero.

BITS	(0:5)	(5:1)	(6:1)	(7:1)	(8:2)	(10:3)	(13:1)	(14:2)
FIELD	(RESERVED)	DISALLOW :FILE	(RESERVED)	CARRIAGE CONTROL	RECORD FORMAT	DEFAULT DESIGNATOR	ASCII/ BINARY	DOMAIN
MEANING		1 = No :FILE 0 = :FILE		0 = NOCCTL 1 = CCTL	00 = Fixed 01 = Variable 10 = Undefined	000 = filename 001 = $STDLIST 010 = $NEWPASS 011 = $OLDPASS 100 = $STDIN 101 = $STDINX 110 = $NULL	0 = Binary 1 = ASCII	00 = New file 01 = Old System File 10 = Temporary File 11 = Old User File

Figure 8-1. Foptions Bit Summary

aoptions logical value (optional)

The *aoptions* parameter permits you to specify up to five different access options established by bit groupings in a 16-bit word. These access options are described below. The bit settings are summarized in figure 8-2.

Bits (12:4) - Access Type *Aoptions*.

The type of access allowed users of this file:

0000 = Read access only. The FWRITE, FUPDATE, and FWRITEDIR intrinsic calls cannot reference this file.

0001 = Write access only. Any data written in the file prior to the current FOPEN request is deleted. The FREAD, FREADSEEK, FUPDATE, and FREADDIR intrinsic calls cannot reference this file.

0010 = Write-access only, but previous data in the file is *not* deleted. The FREAD, FREADSEEK, FUPDATE, and FREADDIR intrinsics cannot reference this file.

0011 = Append access only. The FREAD, FREADDIR, FREADSEEK, FUPDATE, FSPACE, FPOINT, and FWRITEDIR intrinsic calls cannot reference this file. This option is not valid for files containing variable-length records.

0100 = Input/output access. Any file intrinsic except FUPDATE can be issued for this file.

0101 = Update access. All file intrinsics, including FUPDATE, can be issued for this file.

0110 = Execute access. Allows users with privileged mode capability input/output access to any loaded file.

Bit (11:1) - Multirecord *Aoption.*

Signifies that individual read or write requests are not confined to record boundaries. Thus, if the number of words or bytes to be transferred (specified in the *tcount* parameter of the read or write request) exceeds the size of the physical record (i.e., block) referenced, the remaining words or bytes are taken from subsequent successive records until the number specified by *tcount* have been transferred. This option is available only if the inhibit buffering *aoption* described below, is selected also.

0 = Non-multirecord mode.

1 = Multirecord mode.

Bit (10:1) - Dynamic Locking *Aoption.*

Indicates that you want to use the FLOCK and FUNLOCK intrinsics to dynamically permit or restrict concurrent access to the file by other processes at certain times. The user process can continue this temporary locking/unlocking until it closes the file. Dynamic locking/unlocking is made possible through a Resource Identification Number (RIN) assigned to the file and temporarily acquired by the FOPEN intrinsic. The calling process must use the RIN in cooperation with other processes also using it to guarantee the integrity of the file, as discussed in the *MPE Intrinsics Reference Manual.* Non-cooperating processes are allowed concurrent access at all times, unless other provisions prohibit this.

0 = Disallow dynamic locking/unlocking.

1 = Allow dynamic locking/unlocking.

A file may be multiple accessed only if all FOPEN requests for the file specify dynamic locking, or if none of them do. An FOPEN request that disagrees with the current access, if any, will fail.

Bits (8:2) - Exclusive *Aoption.*

This *aoption* specifies whether you have continuous exclusive access to this file, from the time it is opened to the time it is closed. This option often is used when performing some critical operation, such as updating the file.

01 = Exclusive access. After this file is opened, prohibits another FOPEN request, whether issued by this or another process, until this process issues the FCLOSE request or terminates. If any process already is accessing this file when this FOPEN call is issued, a CCL error code is returned to the calling process. If another FOPEN call is issued for this file while the exclusive access *aoption* is in effect, an error code is returned to that calling process. The exclusive *aoption* can be requested only by users allowed the file locking access mode by the security provisions for the file.

10 = Semi-exclusive access. After the file is opened, prohibits concurrent output access to this file through another FOPEN request, whether issued by this or another process, until this process issues the FCLOSE request or terminates. A subsequent request for the

input/output or update *aoption* access type will obtain read only access. Other types of read access, however, are allowed. If any process already has output access to the file when this FOPEN call is issued, a CCL error code is returned to the calling process. If another FOPEN call that violates the read-only restriction is issued while the semi-exclusive *aoption* is in effect, that call fails and an error code is returned to the calling process. The semi-exclusive access can be requested only by users allowed the file-locking access mode by the security provisions for the file.

11 = Share access. After the file is opened, permits concurrent access to this file by any process, in any access mode, subject to other basic MPE security provisions in effect.

00 = Default value. If the read access only access type aoption is selected, share access (11) takes effect. Otherwise, exclusive access (01) takes effect.

Bit (7:1) - Inhibit Buffering *Aoption.*

When selected, this *aoption* inhibits automatic buffering by MPE and allows input/output to take place directly between the user's stack or extra data segment and the applicable hardware device.

0 = Allow normal buffering.

1 = Inhibit buffering.

NOBUF access is oriented to the transfer of physical blocks rather than logical records.

With NOBUF access, you have responsibility for blocking and deblocking of records in the file (Refer to the *MPE Intrinsics Reference Manual*). To be consistent with files built using buffered I/O, records should begin on word boundaries, and when the information content of the record is less than the defined record length, the record should be padded with blanks by you if the file is ASCII or with zeros if the file is binary.

The *recsize* and block size for files manipulated under NOBUF access will follow the same rules as those files that are created using buffering. The default *blockfactor* for a file created under NOBUF is one.

When a NOBUF file is opened without multirecord access, the amount of data transferred per read or write is limited to a maximum of one block.

The end-of-file, next record pointer, and record transfer count are maintained in terms of logical records for all files. The number of logical records affected by each transfer will be determined from the size of the transfer.

Transfers always begin on a block boundary. Those transfers which do not transfer whole blocks leave the next record pointer set to the first record in the next block. The end-of-file pointer always points at the last record in the file.

For files opened with NOBUF access, the FREADDIR, FWRITEDIR, and FPOINT intrinsics will consider the *recnum* parameter as a block number.

Bit (6:1) - Multi-Access Mode *Aoption.*

When selected, this *aoption* provides the accessor with a means of sharing access to the file, including file system buffers. Thus, the accessor can "sequentially" access records within the file in conjunction with other such accessors in the same job. This option is not allowed if the file is accessed exclusively, if NOBUF is selected, or if the multirecord option is requested.

Bit (5:1) - Reserved for MPE. Should be set to zero.

Bit (4:1) - No-Wait I/O.

The selection of this *aoption* allows you to initiate an I/O request and to have control returned before the completion of the I/O. The IOWAIT intrinsic must be called after each I/O request to confirm the completion of the I/O. Also, multirecord access is not available.

Bits (0:3) - Reserved for MPE. Should be set to zero.

Default: All bits are set to zero.

BITS	(0:3)	(4:1)	(5:1)	(6:1)	(7:1)	(8:2)	(10:1)	(11:1)	(12:4)
FIELD	(RES.)	NO WAIT I/O	(RES.)	MULTI ACCESS	INHIBIT BUFFERING	EXCLUSIVE ACCESS	DYNAMIC LOCKING	MULTI-RECORD ACCESS	ACCESS TYPE
MEANING		1 = No-Wait 0 = Non No-Wait		1 = Multi access 0 = Non-Multi access	1 = NOBUF 0 = BUF	01 = Exclusive 10 = Semi-exclusive 11 = Share 00 = Default	0 = No Dynamic Lock 1 = Dynamic Lock	1 = Multi-record 0 = No multi-record	0000 = Read only 0001 = Write only 0010 = Write (save) only 0011 = Append only 0100 = Read/write 0101 = Update 0110 = Execute

Figure 8-2. Aoptions Bit Summary

recsize integer value (optional)

An integer indicating the size of the logical records in the file. If a positive number, this represents words; bytes are represented by a negative number. If the file is a newly-created file, this value is recorded permanently in the file label. If the records in the file are of variable length, this value indicates the maximum logical record length allowed.

Binary files are word oriented. A record size specifying an odd byte count for a binary file is rounded up by FOPEN to the next highest even number.

ASCII files may be created with logical records which are an odd number of bytes in length. Within each block, however, records begin on word boundaries.

For either ASCII or binary files with fixed or undefined length records, the record size is rounded up to the nearest word boundary. For example, a *recsize* specified as −106 for an ASCII file will be 106 characters (53 words) in length. A *recsize* of −113 for a binary file will be 114 characters (57 words) in length. The rounded sizes should be used in computations for *blockfactor* or block size.

Default: For unit-record devices, the default value is the physical record width of the associated device. For all other devices the default value is 128 words.

device byte array (optional)

Contains a string of ASCII characters terminating with any non-alphanumeric character except a slash or period, designating the device on which the file is to reside. The string may represent a device class name up to eight alphanumeric characters beginning with a letter or a logical device number consisting of a three-byte numeric string. Device class names and logical device numbers are defined and assigned to devices during system configuration. See the *MPE General Information Manual* for a discussion of device class names and logical device numbers.

If the file is a newly-created disc file and the device specification is a device class, then all extents of the file will be restricted to members of the class. Similarly, if the device specification is a logical device number, then all extents will be restricted to the specified logical device.

Default: DISC.

formmsg byte array (optional)

Contains a forms message that can be used for such purposes as telling the console operator what type of paper to use in the line printer. This message must be displayed to the operator and verified before this file can be printed on a line printer. The message itself is a string of ASCII characters terminated by a period. The maximum number of characters allowed in the array is 49, which includes the period terminating character. Arrays with more than 49 characters are truncated by MPE.

Default: No forms message is available.

userlabels integer value (optional)

An integer specifying the number of user-label records to be written for this file.

Default: The default number of user-label records is zero.

blockfactor integer value (optional)

An integer containing the size of each buffer to be established for the file, specified as a number equal to the logical records per block. For fixed-length records, *blockfactor* is the actual number of records in a block. For variable-length records, *blockfactor* is interpreted as a multiplier used to compute the block size (maximum *recsize* × *blocfactor*). For undefined-length records, *blockfactor* is always one logical record per block. The *blockfactor* value specified by you may be overridden by MPE. The valid range for *blockfactor* is from 1 to 255. Specification of a negative or zero value results in the default *blockfactor* setting. Values greater than 255 are defaulted to *blockfactor* modulo 256.

Default: 1.

numbuffers integer value (optional)

A 16-bit word whose bits specify the following:

Bits (11:5) - Number of buffers.

Specifies the number of buffers to be allocated to the file. This parameter is not used for files representing interactive terminals, since a system-managed buffering method is always used in such cases. If omitted, set to zero, or set to a negative number, the default value of 2 is set by MPE.

Bits (4:7) - Number of copies.

For spooled output devices, specifies the number of copies of the entire file to be produced by the spooling facility. This can be specified for a file already FOPENed (for example, $STDLIST), in which case the highest value supplied before the last FCLOSE will take effect. The copies will not appear contiguously if the console operator intervenes or if a file of higher *outputpriority* becomes READY before the last copy is complete. This parameter is ignored for non-spooled output devices. The default value is 1.

Bits (0:4) - Output priority.

Specifies the *outputpriority* to be attached to this file. This priority is used to determine the order in which files will be produced when several are waiting for the same device. This parameter must be a number between 1 (lowest priority) and 13 (highest priority), inclusive. If this value is less than the current output fence set by the console operator, the file will be deferred from printing/punching until the operator raises the *outputpriority* of the file or lowers the output fence. This parameter can be specified for a file already FOPENed (for example, $STDLIST), in which case the highest value supplied before the last FCLOSE will take effect. This parameter is ignored for non-spooled devices. The default value is 8.

Default: The default values of all bit groupings are taken.

filesize double value (optional)

A double-word integer specifying the maximum file capacity in terms of physical records for files containing variable-length and undefined-length records, and logical records for files containing fixed-length records. A zero or negative value results in the default *filesize* setting. The maximum capacity is over one million (2^{21}) sectors. The number of sectors in a file is found by the formula shown in the *MPE Intrinsics Reference Manual.*

Default: 1023 physical records.

numextents integer value (optional)

An integer specifying the number of extents (integral number of contiguously-located disc sectors) that can be dynamically allocated to the file as logical records are written to it. The size of each extent is determined by the *filesize* parameter value divided by the *numextents* parameter value. If specified, *numextents* must be an integer from 1 to 32. A zero or negative value results in the default setting.

Default: 8 extents.

NOTE

> Extents are allocated on any disc in the *device* class specified in the device parameter when the file was created. If it is necessary to insure that all extents of a file are on a particular disc, a single disc device class or a logical *device* number must be used in the device parameter.

initialloc integer value (optional)
An integer specifying the number of extents to be allocated to the file when it is opened. This must be an integer from 1 to 32. If an attempt to allocate the requested disc space fails, the FOPEN intrinsic returns an error condition code to the calling program.

Default: 1 extent.

filecode integer value (optional)
An integer recorded in the file label and made available for general use to anyone accesssing the file through the FGETINFO intrinsic. This parameter is used for new files only. For this parameter, any user can specify a positive integer ranging from 0 to 1023. If your process is running in privileged mode, you can specify a negative integer for *filecode* when initially opening a file. Then, any future accesses of the file must be requested in privileged mode and also must specify the correct *filecode*. Certain positive integers beyond 1023 have particular HP-defined meanings, as follows:

1024	A USL file.
1025	A BASIC data file.
1026	A BASIC program file.
1027	A BASIC fast program file.
1028	A relocatable library (RL) file.
1029	A program file.
1030	A STAR file.
1031	A segmented library (SL) file.
1040	A Cross Loader ASCII file (SAVE).
1041	A Cross Loader relocated binary file.
1042	A Cross Loader ASCII file (DISPLAY).
1050	An EDIT KEEPQ file (non-COBOL).
1051	An EDIT KEEPQ file (COBOL).
1052	An EDIT TEXT file (COBOL).
1060	An RJE punch file.
1070	A QUERY procedure file.

Default: 0.

CONDITION CODES:

CCE Request granted. The file is open.

CCG Not returned by this intrinsic.

CCL Request denied. This may be because another process already has exclusive or semi-exclusive access for this file, or an initial allocation of disc space cannot be made due to lack of disc space. The file number value returned by FOPEN if the file is not opened successfully will be zero.

8-3. OPENING A NEW DISC FILE

Figure 8-3 contains an SPL program which opens two files: a card reader file and a new disc file.

The second FOPEN call in figure 8-3

```
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.

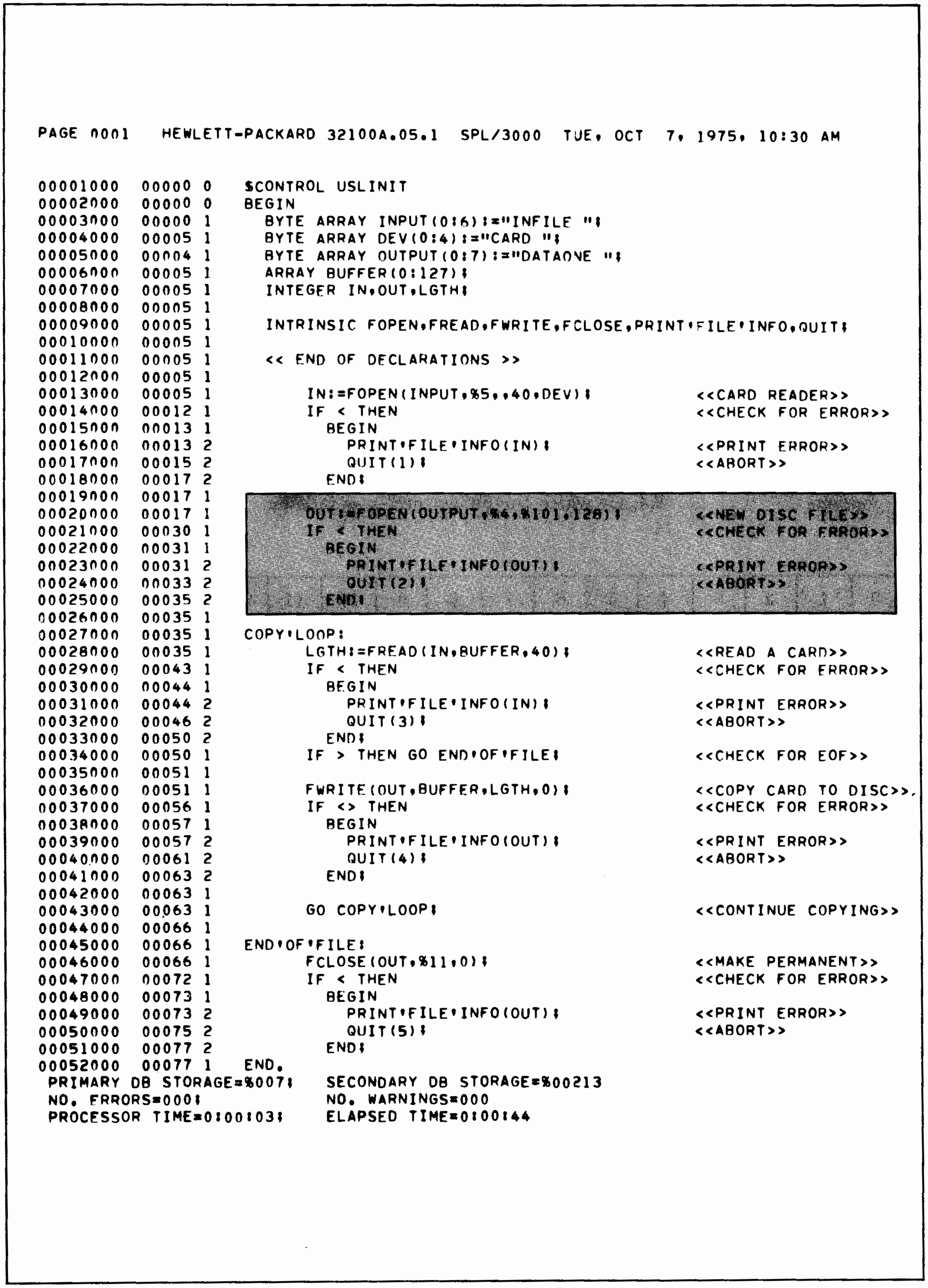
aoptions %101, for which the bit pattern is as follows:

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 8-3. 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.

8-4. READ

```
  I               A         IV
lgth:= READ(message,expetedl);
```

The READ intrinsic reads a string of ASCII characters from a job/session input device into an array in your program. This intrinsic is similar to issuing an FREAD intrinsic call against the file $STDIN. The READ intrinsic is limited in its usefulness, however, in that the full capability of the file system is not available to a user of this intrinsic. For example, :FILE commands are not allowed and certain file intrinsics cannot be used because the *filenum* parameter, obtained from the FOPEN intrinsic, is not available to normal users of the READ intrinsic.

Basic line editing such as cancellation of lines and backspacing are performed automatically by the input/output driver. If the input device is a terminal and it is in full-duplex mode and the echo facility is on, or if the terminal is in half-duplex mode, the characters read are printed.

FUNCTIONAL RETURN:

This intrinsic returns a positive value representing the length of the ASCII string which was read. If *expetedl* is positive, this length specifies words; if *expetedl* is negative, length specifies bytes.

PARAMETERS:

message *array* (required)
The array into which the ASCII characters are read.

expectedl *integer value* (required)
An integer specifying the maximum length of the array message. If *expectedl* is positive, this specifies the length in words; if *expectedl* is negative, this specifies the length in bytes. When the record is read, the first *expectedl* characters are input. If *expectedl* equals or exceeds the size of the physical record, the entire record is transmitted.

CONDITION CODES:

CCE	Request granted.
CCG	A record with a colon in the first column, signalling the end of data, or a hardware end-of-file was encountered.
CCL	Request denied because a physical input/output error occurred. Further error analysis through the FCHECK intrinsic is not possible.

8-5. READX

```
  I                A      IV
lgth:=READX(message,expectedl);
```

The READX intrinsic reads a string of ASCII characters from a job/session input device into an array in your program. This intrinsic is similar to issuing an FREAD intrinsic call against the file $STDINX. The READX intrinsic is limited in its usefulness, however, in that the full capability of the file system is not available to a user of this intrinsic. For example, :FILE commands are not allowed and certain file intrinsics cannot be used because the *filenum* parameter, obtained from the FOPEN intrinsic, is not available to normal users of the READX intrinsic.

Basic line editing such as cancellation of lines and backspacing are performed automatically by the input/output driver. If the input device is a terminal and it is in full-duplex mode and the echo facility is on, or if the terminal is in half-duplex mode, the characters read are printed.

FUNCTIONAL RETURN:

This intrinsic returns a positive value representing the length of the ASCII string which was read. If *expectedl* is positve, this length specifies words; if *expectedl* is negative, length specifies bytes.

PARAMETERS:

message *array* (required)
The array into which the ASCII characters are read.

expectedl *integer value* (required)
An integer specifying the maximum length of the array *message*. If *expectedl* is positive, this specifies the length in words; if *expectedl* is negative, this specifies the length in bytes. When the record is read, the first *expectedl* characters are input. If *expectedl* equals or exceeds the size of the physical record, the entire record is transmitted.

CONDITION CODES:

CCE	Request granted.
CCG	An :EOD, :EOF, or in a job, :EOJ, :JOB, or :DATA command was encountered.
CCL	Request denied because a physical input/output error occurred. Further error analysis through the FCHECK intrinsic is not possible.

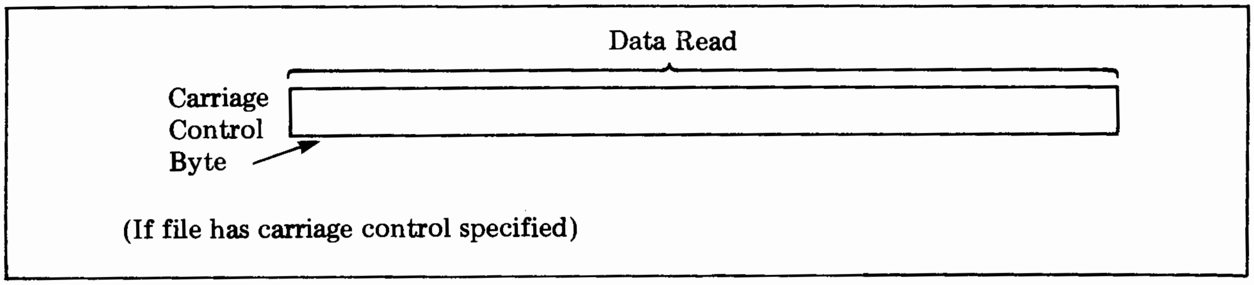
8-6. FREAD

```
  I           IV     A     IV
lgth:=FREAD(filenum,target,tcount);
```

The FREAD intrinsic reads a logical record, or a portion of such a record, from a file on any device to the user's stack.

When the logical end-of-data is encountered during reading, the CCG condition code is returned to the user process. On magnetic tape, the end-of-data can be denoted by a physical indicator such as a tape mark. On disc, the end-of-data occurs when the last logical record of the file is passed. In this case, the CCG condition code is returned and no record has been read. If the file is embedded in an input source containing MPE commands, the end-of-data is indicated when an :EOD command is encountered, but the :EOD command itself is not returned to the user. The end-of-data is indicated by a hardware end-of-file, including :EOF:, or on $STDIN by an MPE command, or on $STDINX by :EOD. In addition, on the standard input device for a job, as opposed to a session, :JOB, :EOJ, or :DATA indicate end-of-data.

When an old file containing carriage-control characters, supplied through the *control* parameter of the FWRITE intrinsic, is read, and the carriage-control *foption* parameter of the FOPEN intrinsic, or the CCTL parameter of the :FILE command is specified, the carriage-control byte is read as follows:



It is possible to skip portions of records inadvertently if the *multirecord aoption* of FOPEN is set and the *tcount* parameter specified is greater than one logical record. For example, if you read all of record 11 and half of record 12 in a file, the logical record pointer will be set to the beginning of record 13 after the FREAD intrinsic executes. Thus, the second half of record 12 may be skipped.

FUNCTIONAL RETURN:

The FREAD intrinsic returns a positive integer value showing the length of the information transferred. If the *tcount* parameter in the FREAD call was positive, the positive value returned represents a word count; if the *tcount* parameter was negative, the positive value returned represents a byte count.

PARAMETERS:

filenum *integer value* (required)
A word identifier supplying the file number of the file to be read.

target *array* (required)

An array to which the record is to be transferred. This array should be large enough to hold all of the information to be transferred.

tcount *integer value* (required)

An integer specifying the number of words or bytes to be transferred. If this value is positive, it signifies the length in words; if it is negative, it signifies the length in bytes; if it is zero, no transfer occurs. If *tcount* is less than the size of the record, only the first *tcount* words or bytes are read from the record.

If *tcount* is larger than the size of the physical record (i.e., block), and the *multirecord* aoption was not specified in FOPEN, transfer is limited to the length of the physical record. If the *multirecord aoption* was specified in FOPEN, the remaining words or bytes specified in *tcount* will be read from succeeding records.

CONDITION CODES:

CCE	The information was read.
CCG	The logical end-of-data was encountered during reading.
CCL	The information was not read because an error occurred, a terminal read was terminated by a special character as specified in the FCONTROL intrinsic, or a tape error was recovered and the FSETMODE option was enabled.

SPLIT STACK CALLS ARE PERMITTED.

8-7. READING A FILE IN SEQUENTIAL ORDER

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.

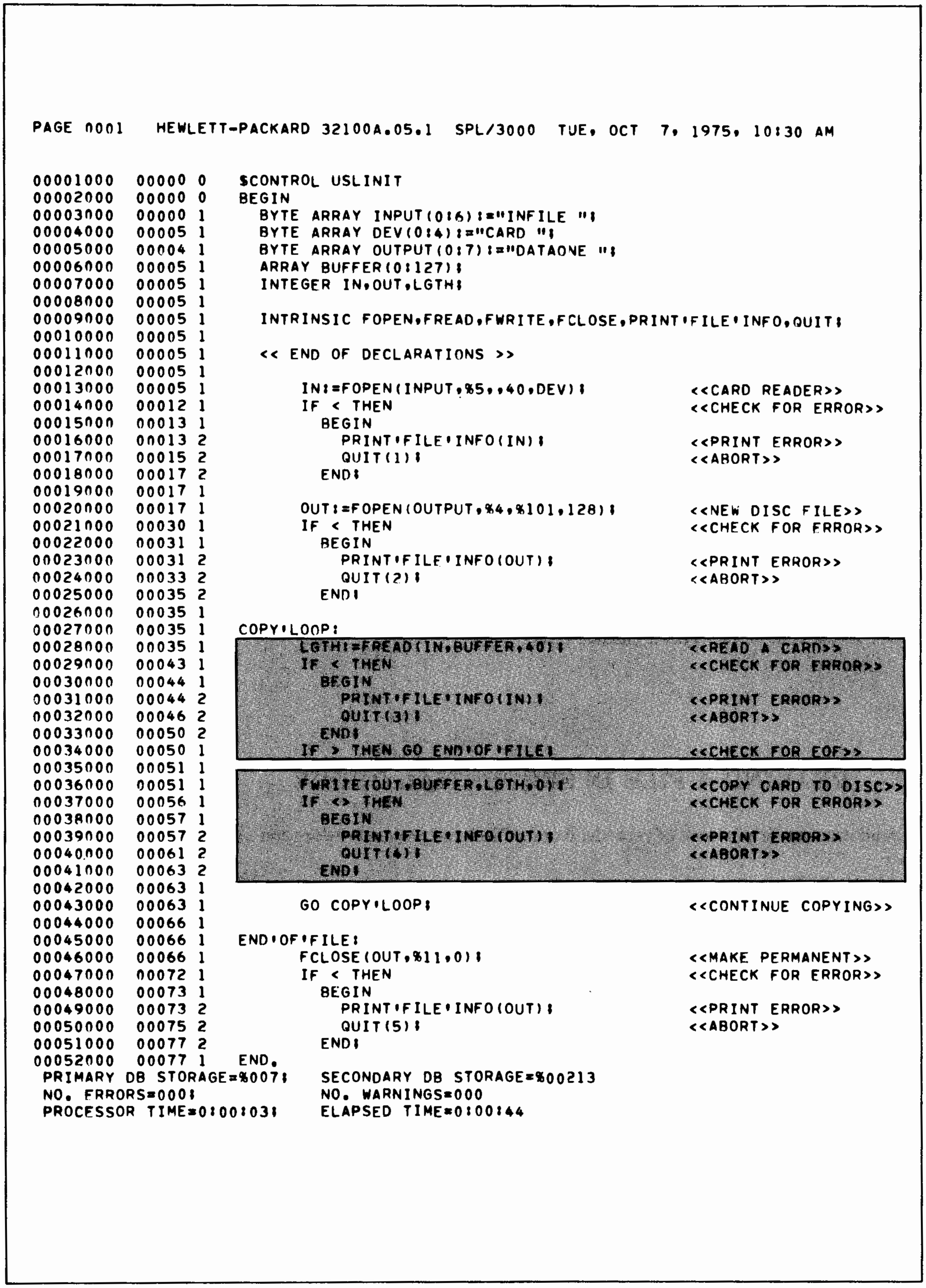
NOTE

> The logical record pointer is a number kept by MPE to indicate the next sequential record to be accessed in a file.

The program shown in figure 8-4 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.



```
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 8-4. FREAD Intrinsic Example

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.

8-8. FREADDIR

```
           IV      A      IV     DV
FREADDIR(filenum,target,tcount,recnum);
```

The FREADDIR intrinsic reads a specific logical record, or a portion of such a record, from a disc file to the user's data stack. This intrinsic differs from the FREAD intrinsic in that the FREAD intrinsic reads only the record pointed to by the logical record pointer. The FREADDIR intrinsic may be issued only for disc files composed of fixed-length or undefined-length records.

After the FREADDIR intrinsic is executed, the logical record pointer is set to the beginning of the next logical record, or first logical record of the next block for NOBUF files, in the file.

It is possible to skip portions of records inadvertently if the *multirecord aoption* of FOPEN is set and the *tcount* parameter specified is greater than one logical record. For example, if you read all of record 11 and half of record 12 in a file, the logical record pointer will be set to the beginning of record 13 after the FREADDIR intrinsic executes. Thus the second half of record 12 may be skipped.

PARAMETERS:

filenum *integer value* (required)
A word identifier supplying the file number of the file to be read.

target *array* (required)
An array to which the record is to be transferred. This array should be large enough to hold all of the information to be transferred.

tcount *integer value* (required)
An integer specifying the number of words or bytes to be transferred. If this value is positive, it signifies words; if negative, it signifies bytes; and if it is zero, no transfer occurs. If *tcount* is less than the size of the record, only the first *tcount* words or bytes are read from the record.

If *tcount* is larger than the size of the logical record and the *multirecord aoption* was not specified in FOPEN, the transfer is limited to the length of the logical record. If the *multirecord aoption* was specified in FOPEN, the remaining words or bytes specified in tcount will be read from succeeding records.

recnum double value (required)
A double-word integer indicating the relative number, in the file, of the logical record to be read. The first record is indicated by 0D (double word zero).

CONDITION CODES:

CCE	The specified information was read.
CCG	The logical end-of-data was encountered during reading.
CCL	The information was not read because an error occurred.

SPLIT STACK CALLS ARE PERMITTED.

8-9. PRINT

```
              A     IV      IV
PRINT(message,length,control);
```

You can write a string of ASCII characters from an array to the job/session listing device by using the PRINT intrinsic. This intrinsic is similar to issuing an FWRITE intrinsic call against the file $STDLIST. The PRINT intrinsic is limited in its usefulness, however, in that the full capability of the file system is not available to a user of this intrinsic. For example, :FILE commands are not allowed and certain file intrinsics cannot be used because the *filenum* parameter, obtained from the FOPEN intrinsic, is not available to normal users of the PRINT intrinsic.

PARAMETERS:

message *array* (required)
Contains the character string to be output.

NOTE

You can avoid annoying warning messages in the compiled output by equivalencing a byte array to a logical array for the message parameter.

length *integer value* (required)
An integer denoting the length of the character string to be transmitted. If *length* is positive, it specifies the length in words; if *length* is negative, it specifies the length in bytes.

control *integer value* (required)
An integer representing a carriage-control code as shown in figure 8-5.

CONDITION CODES:

CCE	Request granted.
CCG	End-of-data encountered.
CCL	Request denied because of input/output error. Further error analysis through the FCHECK intrinsic is not possible.

Octal Code	ASCII Symbol	Carriage Action
%40	“ ”	*Single-space.
%60	“0”	*Double-space.
%61	“1”	Page-eject (form-feed).
%53	“+”	No space, return (next printing at column 1).
%2nn (where n is any digit from 0 through 7)		Space nn lines. (No automatic page eject.)
%300		Page-eject (Tape Channel 1).
%301		Skip to bottom of form (Tape Channel 2).
%302		Single-spacing (with automatic page eject). (Tape Channel 3.)
%303		Single-space on next odd-numbered line (with automatic page eject). (Tape Channel 4.)
%304		Triple-space (with automatic page eject). (Tape Channel 5.)
%305		Space 1/2 page (with automatic page eject). (Tape Channel 6.)
%306		Space 1/4 page (with automatic page eject). (Tape Channel 7.)
%307		Space 1/6 page (with automatic page eject). (Tape Channel 8.)
%310		Space to bottom of form. (Tape Channel 9.)
%311		Skip to Tape Channel 10. (User option.)
%312		Skip to Tape Channel 11. (User option.)
%313		Skip to Tape Channel 12. (User option.)
%320		No space, no return. (Next printing physically follows this.)
%0-%37 %41-%52 %54-%57 %62-%77 %314-%317 %321-%377		**Same as %40
%400 or %100		Set post-space movement option; this first prints, then spaces. If previous option set was pre-space movement option, the driver outputs a line (and suppresses spacing) to clear the buffer.
%401 or %101		Set pre-space movement option; this first spaces, then prints.
%402 or %102		Set single-space option, with automatic page eject (60 lines per page).
%403 or %103		Set single-space option *without* automatic page eject (66 lines per page).

* Spacing with or without automatic page eject can be selected.

**Future MPE/3000 requirements may necessitate redefinition of these octal codes.

Figure 8-5. Carriage Control Directives

8-10. FWRITE

```
              IV      A       IV      LV
FWRITE(filenum,target,tcount,control);
```

The FWRITE intrinsic writes a logical record, or a portion of such a record, from the user's stack to a file on any device.

When information is written to a fixed-length record, and the NOBUF aoption was not specified in FOPEN, any unused portion of the record will be padded with binary zeros for a binary file or ASCII blanks for an ASCII file.

When the FWRITE intrinsic is executed, the logical record pointer is set to the record immediately following the record just written, or the first logical record in the next block for NOBUF files.

When an FWRITE call writes a record beyond the current logical end-of-file indicator, this indicator is advanced to a farther location. If the physical bounds of the file are reached, the CCL condition code is returned.

PARAMETERS:

filenum *integer value* (required)
A word identifier supplying the file number of the file to be written on.

target *array* (required)
Contains the record to be written.

tcount *integer value* (required)
An integer specifying the number of words or bytes to be written to the record. If this value is positive, it signifies words; if it is negative, it signifies bytes; if it is zero, no transfer occurs. If tcount is less than the recsize parameter associated with the record, only the first tcount words or bytes are written.

If *tcount* is larger than the *recsize* value, and the *multirecord aoption* was not specified in FOPEN, the FWRITE request is refused and condition code CCL is returned. If the *multirecord aoption* was specified in FOPEN, the excess words or bytes are written to succeeding records. For files for which carriage control is specified, the actual data transferred is limited to *recsize* minus one byte.

control *logical value* (required)
A logical value representing a carriage control code, effective if the file is transferred to a line printer or terminal (including a spooled file whose ultimate destination is a line printer or a terminal). This parameter is effective only for files opened with carriage control specified. The options are:

0 = Print the full record transferred, using single spacing. This results in a maximum of 132 characters per printed line.

1 = Use the first character of the data written to signify space control, and suppress this character on the printed output. This results in a maximum of 132 characters of data per printed line. Permissible control characters are shown in figure 8-5.

Any octal code from figure 8-5 can be used to determine space control and print the full record transferred. This results in a maximum of 132 characters per printed line. No data will be transferred if the octal code is 100 through 103 or 400 through 403, or if the octal code is 1 and the embedded control is octal 100 or 103.

If the *control* parameter is not 0 or 1, and *tcount* is 0, only the space control is executed — no data is transferred.

The effect of the FWRITE *control* parameter in combination with the FOPEN carriage control *foption* (or overriding :FILE command CCTL/NOCCTL parameter) upon the data written is summarized in figure 8-6.

FOPEN OR :FILE	**FWRITE Control Parameter**		
	= 0	**= 1**	**= Greater than 1**
Carriage Control Foption Specified or CCTL	Byte 1 0 record Data output contains 132 characters; the first byte contains 0.	record Data output contains 132 characters; the carriage control character in the first byte is suppressed.	Byte 1 con-trol record Data output contains 132 characters; the first character is a carriage-control character specified by the FWRITE *control* parameter.
Carriage Control Foption *not* specified or NOCCTL	record Data output contains 132 characters.	record Data output contains 132 characters	record Data output contains 132 characters.

EFFECT ON DATA OUTPUT

Figure 8-6. Carriage Control Summary

You determine whether the carriage control directive takes effect before printing (pre-space movement) or after printing (post-space movement), through the FCONTROL intrinsic.

All of the carriage control codes listed in figure 8-5 may be used as the value of the *param parameter* in FCONTROL (when *controlcode* = 1), regardless of whether the file is opened with

CCTL or NOCCTL. When the file is opened with CCTL, these carriage control codes may be used in either of the following ways via FWRITE:

a. As the value of the *control* parameter.

b. When *control* = 1, as the first byte of the *target* array.

The default carriage control code is post spacing with automatic page eject. This applies to all HP-supported subsystems except FORTRAN which is prespacing with automatic page eject.

CONDITION CODES:

CCE	Request granted.
CCG	The physical bounds of the file prevented further writing; all disc extents are filled.
CCL	Request denied because an error occurred, such as *tcount* exceeding the size of the record in non-multirecord mode; or the FSETMODE option is enabled to signify recovered tape errors; or the end-of-tape marker was sensed.

SPLIT STACK CALLS ARE PERMITTED.

8-11. WRITING RECORDS INTO A FILE IN SEQUENTIAL ORDER

To write records, or portions of records, from your buffer to a file in sequential order, you use the FWRITE intrinsic.

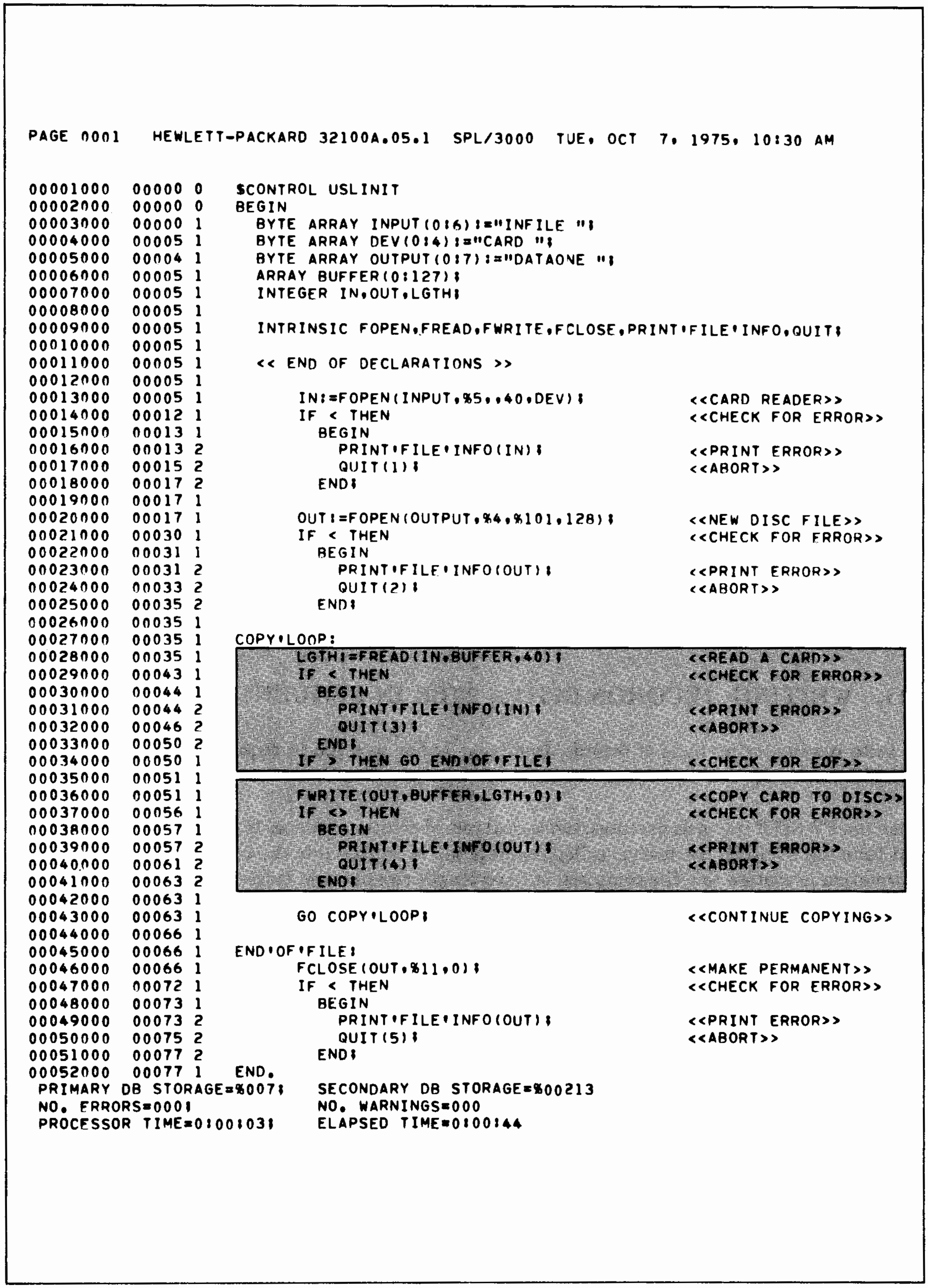
When the FWRITE intrinsic executes, the logical record pointer advances to the next record. Then, the next time the FWRITE intrinsic is called, information is written into the next record position. When information is written to a file composed of fixed-length records (and buffering is not specified 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 8-7

```
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.



```
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 8-7. FWRITE Intrinsic Example

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.

8-12. FWRITEDIR

```
           IV      A     IV      DV
FWRITEDIR(filenum,target,tcount,recnum);
```

The FWRITEDIR intrinsic writes a specific logical record, or a portion of such a record, from the user's stack to a disc file. This intrinsic differs from the FWRITE intrinsic in that the FWRITE intrinsic writes only the record pointed to by the logical record pointer. The FWRITEDIR intrinsic may be used only for disc files composed of fixed or undefined-length records.

When information is written to a fixed-length record and NOBUF was not specified in the FOPEN call that opened the file, any unused portion of the record will be padded with binary zeros for a binary file, or ASCII blanks for an ASCII file.

When the FWRITEDIR intrinsic is executed, the logical record pointer is set to the record immediately following the record just written, or the first logical record of the next block for NOBUF files.

When an FWRITEDIR call writes a record beyond the current logical end-of-file indicator, the indicator is advanced to a farther location. This can result in the creation of dummy records to pad the records between the previous end-of-file and the newly-written record. These dummy records are filled with binary zeros for a binary file, or with ASCII blanks for an ASCII file.

When the physical bounds of the file prevent further writing, because all allowable extents are filled, the end-of-file condition (CCG) is returned to the user's program.

PARAMETERS:

filenum *integer value* (required)
A word identifier specifying the file number of the file to be written on.

target *array* (required)
Contains the record to be written. This array should be large enough to hold all the information to be transferred.

tcount *integer value* (required)
An integer specifying the number of words or bytes to be written to the record. If this value is positive, it signifies words; if it is negative, it signifies bytes; if it is zero, no transfer occurs. If *tcount* is less than the *recsize* parameter associated with the record, only the first *tcount* words or bytes are written.

If *tcount* is larger than the size of the logical record and the *multirecord aoption* was not specified in FOPEN, the transfer is limited to the length of the logical record. If the *multirecord aoption* was specified in FOPEN, the remaining words or bytes are written to succeeeding records.

recnum *double value* (required)
A double integer indicating the relative number of the logical record, or block number for NOBUF files, to be written. The first record is indicated by 0D.

CONDITION CODES:

CCE	Request granted.
CCG	The physical end-of-file was encountered.
CCL	Request denied because an error occurred.

SPLIT STACK CALLS ARE PERMITTED

8-13. FUPDATE

```
           IV       A     IV
FUPDATE(filenum,target,tcount);
```

The FUPDATE intrinsic updates a logical record in a disc file. This intrinsic affects the logical record (or block for NOBUF files) last referenced by an intrinsic call for the file specified. FUPDATE moves the specified information from the user's stack into this record. The file containing this record must have been opened with the update aoption specified in the FOPEN call, and must not have variable-length records.

PARAMETERS:

filenum *integer value* (required)
A word identifier supplying the file number of the file to be updated.

target *array* (required)
Contains the record to be written in the updating.

tcount *integer value* (required)
An integer specifying the number of words or bytes to be written to the record. If this value is positive, it signifies words; if it is negative, it signifies bytes; if it is zero, no transfer occurs. If *tcount* is less than the *recsize* parameter associated with the record, only the first *tcount* bytes or words are written. For buffered files, tcount is limited to the block size. FUPDATE cannot perform multirecord updates.

CONDITION CODES:

CCE	Request granted.
CCG	An end-of-file condition was encountered during updating.
CCL	Request denied because of an error, such as the file not residing on disc, or *tcount* exceeding the size of the record when multirecord mode is not in effect.

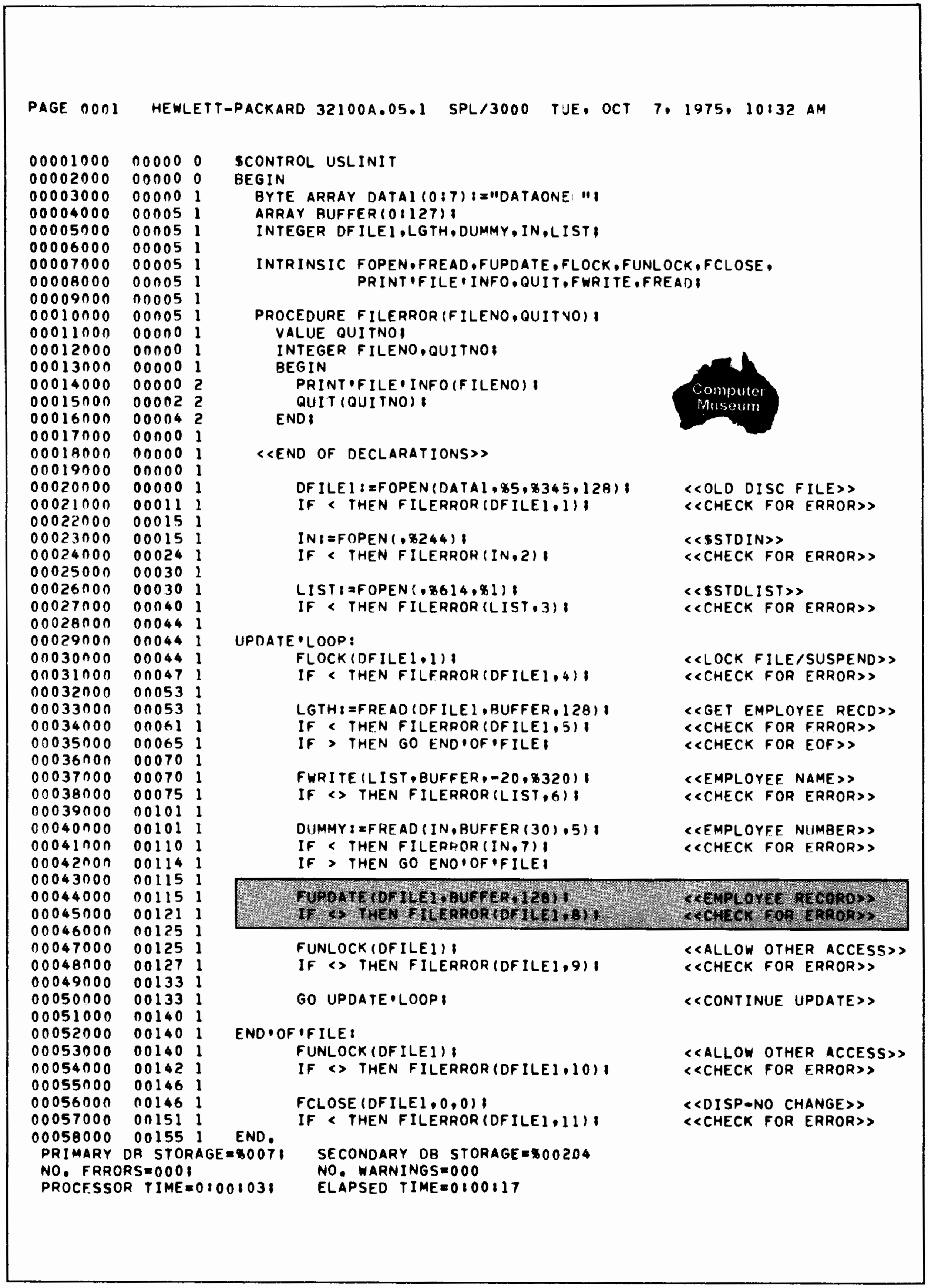
SPLIT STACK CALLS ARE PERMITTED.

8-14. UPDATING A FILE

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.



```
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 FILFRROR(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 FRROR>>
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);         <<EMPLOYFE NUMBER>>
00041000  00110 1         IF < THEN FILERROR(IN,7);              <<CHECK FOR ERROR>>
00042000  00114 1         IF > THEN GO ENO'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. FRRORS=000;                 NO. WARNINGS=000
  PROCFSSOR TIME=0:00:03;         ELAPSED TIME=0:00:17
```

Figure 8-8. FUPDATE Intrinsic Example

Figure 8-8 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.

8-15. FCLOSE

```
         IV        IV        IV
FCLOSE(filenum,disposition,seccode);
```

The FCLOSE intrinsic terminates access to a file. This intrinsic applies to files on all devices. FCLOSE deletes the buffers and control blocks through which the user process accessed the file. It also deallocates the device on which the file resides and it may change the disposition of the file. If you do not issue FCLOSE calls for all files opened by your process, such calls are issued automatically by MPE when the process terminates. When a file on magnetic tape is saved, the tape is rewound. All magnetic tape files are left offline after an FCLOSE to indicate to the operator that they may be removed (i.e., the magnetic tape drive has been deallocated).

PARAMETERS:

filenum *integer value* (required)
A word identifier supplying the file number of the file to be closed.

disposition *integer value* (required)
Indicates the disposition of the file, significant only for files on disc and magnetic tape. This disposition can be overridden by a corresponding parameter in a :FILE command entered prior to program execution. The disposition options are defined by two bit fields, as follows:

Bits (13:3) - Domain Disposition

0 = No change. The disposition code remains as it was before the file was opened. Thus, if the file is new, it is deleted by FCLOSE; otherwise, the file is assigned to the domain to which it belonged previously.

1 = Permanent file. The file is saved in the system file domain. If the file is a new or old temporary file on disc, an entry is created for it in the system file directory. An error code is returned if a file of the same name already exists in the directory. If the file is an old permanent file on disc, this disposition value has no effect. If the file is stored on magnetic tape, that tape is rewound and unloaded.

2 = Temporary job file (rewound). The file is retained in the user's temporary (job/session) file domain and can thus be requested by any process within the job/session. The uniqueness of the file name is checked. If a file of the same name already exists, an error code is returned. If the file resides on magnetic tape, the tape is rewound but not unloaded.

3 = Temporary job file (not rewound). This option has the same effect as disposition code 2, except that tape files are not rewound.

4 = Released file. The file is deleted from the system.

NOTE

Although the basic functions covering magnetic tape files are covered above in dispositions 0 through 4, it is recommended that you read the discussion of magnetic tape files in the *MPE Intrinsics Reference Manual* for special considerations not mentioned here.

Default value for this field is 0 (no change)

Bit (12:1) - Disc Space Disposition

1 = Returns to the system any disc space allocated beyond the end-of-file indicator.

0 = Does not return any disc space allocated beyond the end-of-file indicator.

The default value for this field is 0 (no return).

When a file is opened by the FOPEN intrinsic, a file count (maintained by the system) is incremented by one. When the file is FCLOSEd, the file count is decremented by one. If more than one FOPEN is in effect for a particular file, its disposition is saved but not affected by the FCLOSE call until the file count is decremented to zero. Then the effective (saved) disposition is the smallest non-zero disposition parameter specified among all FCLOSE calls issued against the file. For example, a file XYZ is opened three successive times by a process. The first FCLOSE disposition is 1, the second FCLOSE disposition is %4, and the third (and last) FCLOSE disposition is %12. The final disposition on the file XYZ will be disposition 1 (permanent file and no return of disc space).

Bits (0:12) are reserved for MPE and should be set to zero.

seccode *integer value* (required)

Denotes the type of security initially applied to the file, significant only for new permanent files. The options are:

0 = Unrestricted access — the file can be accessed by any user, unless prohibited by current MPE provisions.

1 = Private file creator security — the file can be accessed only by its creator.

The default value is 0.

CONDITION CODES:

CCE	The file was closed successfully.
CCG	Not returned by this intrinsic.
CCL	The file was not closed, perhaps because an incorrect *filenum* was specified, or because another file with the same name and disposition exists in the system.

SPLIT STACK CALLS ARE PERMITTED.

8-16. FCHECK

O-V

FCHECK(***filenum***,*errorcode*,*tlog*,*blknum*,*numrecs*);

When a file intrinsic returns a condition code indicating a physical input/output error, additional details may be obtained by using the FCHECK intrinsic call. This intrinsic applies to files on any device.

FCHECK accpets zero as a legal *filenum* parameter value. When zero is specified, the information returned in *errorcode* reflects the status of the last call to FOPEN. When an FOPEN fails, there is obviously no file number which can be referenced in *filenum*. Therefore, when an FOPEN fails, a *filenum* of zero can be used in the FCHECK intrinsic call to obtain the *errorcode* only. If the *tlog, blknum,* or *numrecs* parameters are specified, a zero value will be returned to these parameters. If a *filenum* of zero is used for a file which has been previously FOPENed, but not yet FCLOSEd, the returned *errorcode* will be meaningless.

PARAMETERS:

filenum *integer value* (required)
A word identifier supplying the file number of the file for which error information is to be returned.

errorcode integer (optional)
A word to which is returned a 16-bit code, specifying the type of error that occurred. If the previous operation was successful, all 16 bits are set to zero. The errorcodes are shown in table 8-1.

Default: The errorcode is not returned.

tlog integer (optional)
A word to which is returned the transmission log value recorded when an erroneous data transfer occurs. This word specifies the number of words not read or written (those left over) as the result of the input/output error.

Default: The transmission log value is not returned.

blknum double (optional)
A double word to which is returned the relative number of the block involved in the error.

Default: The block number is not returned.

numrecs integer (otional)
A word to which is returned the number of logical records in the bad block.

Default: The number of logical records is not returned.

CONDITION CODES:

CCE Request granted.

CCG Not returned by this intrinsic.

CCL Request denied because filenum was invalid or a bounds violation occurred while processing this request and errorcode is 73.

SPLIT STACK CALLS ARE PERMITTED.

CODE (DECIMAL)	MEANING
0	End of file.
1	Illegal DB register setting (typically, a request in split-stack mode when it is illegal).
8	Illegal parameter value.
20	Invalid operation.
21	Data parity error.
22	Software time-out.
23	End of tape.
24	Unit not ready.
25	No write ring on tape.
26	Transmission error.
27	Input/output time-out.
28	Timing error or data overrun.
29	Start input/output (SIO) failure.
30	Unit failure.
31	End of line (special character terminator).
32	Software abort of input/output operation.
33	Data lost.
34	Unit not on line.
35	Data set not ready.
36	Invalid disc address.
37	Invalid memory address.
38	Tape parity error.

Table 8-1. FCHECK Error Codes

CODE (DECIMAL)	MEANING
39	Recovered tape error.
40	Operation inconsistent with access type.
41	Operation inconsistent with record type.
42	Operation inconsistent with device type.
43	The *tcount* parameter value exceeded the *recsize* parameter value in this intrinsic, but the *multirecord access aoption* was not specified in the currently-effective FOPEN intrinsic.
44	The FUPDATE intrinsic was called, but the file was positioned at record zero. (FUPDATE must reference the last record read, but no previous record was read.)
45	Privileged file violation.
46	Insufficient disc space.
47	Input/output error occurs on a file label.
48	Invalid operation due to multiple file access.
49	Unimplemented function.
50	The account referenced does not exist.
51	The group referenced does not exist.
52	The file referenced does not exist in the system file domain.
53	The file referenced does not exist in the job temporary file domain.
54	The file reference is invalid.
55	The device referenced is not available.
56	The device specification is invalid or undefined.
57	Virtual memory is not sufficient for the file specified.
58	The file was not passed (typically, a request for $OLDPASS when there is no $OLDPASS).
59	Standard label violation.
60	Global RIN not available.
61	Group disc file space exceeded.

Table 8-1. FCHECK Error Codes (Continued)

CODE (DECIMAL)	MEANING
62	Account disc file space exceeded.
63	Non-sharable device (ND) capability required but not assigned.
64	Multiple RIN (MR) capability required but not assigned.
66	Plotter limit switch reached.
67	Paper tape error.
68	System internal error.
69	Miscellaneous (ATTACHIO) input/output error.
71	Too many files opened for process.
72	Invalid file number.
73	Bounds check violation.
77	NO-WAIT input/output operation is pending.
78	There is no NO-WAIT input/output for any file.
79	There is no NO-WAIT input/output for file specified.
80	Configured maximum number of spoolfile sectors would be exceeded by this output request.
81	No SPOOL class defined in system.
82	Insufficient space in SPOOL class to honor this input/output request.
83	Extent size exceeds maximum allowable.
84	The next extent in this spoolfile resides on a device which is unavailable to the system (i.e., the device is =DOWN).
85	Operation inconsistent with spooling; e.g., attempt to read hardware status.
86	Spool process internal error.
89	Power failure.
90	The callling process requested exclusive access to a file to which another process has access.
91	The calliing process requested access to a file to which another process has exclusive access.

Table 8-1. FCHECK Error Codes

CODE (DECIMAL)	MEANING
92	Lockword violation.
93	Security violation.
94	Creator conflict in use of FRENAME intrinsic (user is not the creator).
95	"BROKEN" terminal read.
96	Miscellaneous disc input/output error (device may require HP Customer Engineer attention).
97	CONTROL Y processing requested but no CONTROL Y PIN exists.
98	Input/output read time has overflowed.
99	Magnetic tape error. Beginning of tape (BOT) found while requesting a backspace record (BSR) or a backspace file (BSF).
100	Duplicate file name in the system file directory.
101	Duplicate file name in the job temporary file directory.
102	Directory input/output error.
103	System directory overflow.
104	Job temporary directory overflow.
106	Extent size exceeds maximum allowable.
107	Offset to data is greater than 255 words.
108	Inaccessible file due to a bad file label.
109	Illegal carriage control option.
110	The intrinsic attempted to save a system file in the job temporary file directory.

Table 8-1. FCHECK Error Codes (Continued)

8-17. FCONTROL

```
            IV          IV    L
FCONTROL(filenum,controlcode,param);
```

The FCONTROL intrinsic performs various control operations on a file or on the device on which the file resides. These operations include:

- Supplying a printer or terminal carriage-control directive.
- Verifying input/output.
- Reading the hardware status word pertaining to the device on which the file resides.
- Setting a terminal's time-out interval.
- Rewinding the file.
- Writing an end-of-file indicator.
- Skipping forward or backward to a tape mark.

The FCONTROL intrinsic applies to files on disc, tape, terminal, or line printer.

NOTE

> The FCONTROL intrinsic also can be used to perform various terminal functions, such as changing the terminal speed or enabling parity checking. See the *MPE Intrinsics Reference Manual* for descriptions of these functions.

PARAMETERS:

NOTE

> The parameters described here pertain to the FCONTROL intrinsic as it is used to perform control operations on a file or on the device on which the file resides. Descriptions of FCONTROL parameters when the intrinsic is used to change terminal characteristics are described in the *MPE Intrinsics Reference Manual.*

filenum *integer value* (required)
A word identifier supplying the file number of the file for which the control operation is to be performed.

controlcode *integer value* (required)
An integer identifying the operation to be performed:

0 = General Device Control. The *param* parameter is transmitted to the appropriate device driver, and the value returned is transmitted to the user through the *param* parameter.

1 = Line Control. A request to send the value specified in the *param* parameter to the terminal or line printer driver as a carriage-control directive. Use line controls provided by FWRITE when directing to a disc or a spooled file.

2 = Complete Input/Output. This insures that requested input/output has been physically completed. Valid only for buffered files. Posts the block being transferred whether full or not.

3 = Read Hardware Status Word. This operation will return in *param* the status word from the device on which the file resides. The returned value is the status of the device from the previous input/output operation, including FOPEN of the file.

4 = Set Time-Out Interval. This code indicates that a time-out interval is to be applied to input from the terminal. If input is requested from the terminal but is not received in this interval, the FREAD request terminates prematurely with condition code CCL. The interval itself is specified, in seconds, in a word in the user's stack, indicated by *param*. If this interval is zero, any previously established interval is cancelled, and no time-out occurs. *Controlcode* 4 is ignored if the addressed file is not being read from the terminal.

5 = Rewind File. This repositions the file at its beginning, so that the next record read or written is the first record read or written is the first record in the file. This code is valid only for files on disc and magnetic tape.

6 = Write End-of-File. This operation is used to denote the end of a file on disc or magnetic tape.

7 = Space Forward to Tape Mark. This moves the tape forward until a tape mark is encountered.

8 = Space Backward to Tape Mark. This moves the tape backward until a tape mark is encountered.

9 = Rewind and Unload Tape File. This repositions the tape file at its beginning and places the tape offline.

NOTE

Control codes 0, 1, and 3 will be rejected for spooled devicefiles. Control codes 5 through 9 (magnetic tape control) will be rejected for spooled :DATA tapes.

Although the basic functions covering magnetic files are covered above, it is recommended that you read the discussion of magnetic tape files in the *MPE Intrinsics Reference Manual* for special considerations not covered here.

param *logical* (required)

If *controlcode* is 1, *param* denotes a word containing the value to be transmitted to the terminal or line printer driver as a carriage control or mode control directive. The carriage control directive is selected from figure 8-5.

The mode control determines whether any carriage control directive transmitted through the FWRITE intrinsic takes effect before printing (pre-space movement) or after printing (post-space movement). The mode control directive is selected from the octal codes %400 or %401 in figure 8-5.

If *param* contains a mode control directive, then a value is returned to *param* that shows the mode setting of the device as it was before the call to FCONTROL, as follows:

Value	**Meaning**
0	Post-Spacing
1	Pre-Spacing

If *controlcode* is 4, *param* denotes a word in the user's stack that contains the time-out interval, in seconds, to be applied to input from the terminal.

If *controlcode* is 2, 5, 6, 7, 8, or 9, param is any variable or word identifier. This parameter is needed by FCONTROL to satisfy the internal requirements of the intrinsic. It serves no other purpose, however, and is not modified by the intrinsic.

CONDITION CODES:

CCE	Request granted.
CCG	Not returned by this intrinsic.
CCL	Request denied because an error occurred.

SPLIT STACK CALLS ARE PERMITTED.

8-18. FSPACE

```
              IV            IV
FSPACE(filenum,displacement);
```

You can space forward or backward on a fixed-length or undefined-length file by using the FSPACE intrinsic. This results in resetting the logical record pointer. The FSPACE intrinsic applies to files on disc and magnetic tape devices only.

The FSPACE intrinsic cannot be used with variable-length record files or with spooled files on disc. An attempt to use this intrinsic on such files results in a CCL error condition code and the logical record pointer is left at its current position.

See the *MPE Intrinsics Reference Manual* for special considerations for magnetic tape files.

PARAMETERS:

filenum *integer value* (required)
A word identifier supplying the file number of the file on which spacing is to be done.

displacement *integer value* (required)
A signed integer indicating the number of logical records for buffered disc files, or blocks for NOBUF files and all tape files, to be spaced over, relative to the current position of the logical record pointer. A positive value signifies forward spacing, a negative value signifies backward spacing. The maximum positive value is 32767, the maximum negative value is −32768. For positve values, the sign is optional.

CONDITION CODES:

CCE	Request granted.
CCG	A logical end-of-file indicator was encountered during spacing. For disc files, the logical record pointer has not been changed. For magnetic tape files, the logical record pointer will be pointing to the logical end-of-file. The magnetic tape, however, will be positioned such that it will be one record past the file mark on the tape.
CCL	Request denied because an error occurred; the file resides on a device that prohibits spacing.

SPLIT STACK CALLS ARE PERMITTED.

8-19. 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 discussed in the following paragraphs:

- 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.

For handling floating point numbers, refer to the EXTIN' and INEXT' procedures in the *Compiler Library Reference Manual.*

8-20. ASCII

```
   I              LV   IV     BA
numchar:= ASCII(word,base,string);
```

Any 16-bit binary number can be converted to a different base and represented as a numeric character ASCII string by using the ASCII intrinsic call.

FUNCTIONAL RETURN:

This intrinsic returns the number of characters in the resulting string.

PARAMETERS:

word *logical value* (required)
The number to be converted to an ASCII string.

base *integer value* (required)
An integer indicating octal or decimal conversion.

8 = octal

10 = decimal (left justified)

−10 = decimal (right justified)

If any other number is entered in this parameter, the intrinsic causes the process to abort.

string *byte array* (required)
A byte array into which the converted value is placed. This array must be long enough to contain the result. No result, however, exceeds six characters. For octal conversion (Base = 8), six characters including leading zeros, are always returned in *string,* showing the octal representation of word. In octal conversions, the length returned by ASCII is the number of significant (right-justified)characters in *string* (excluding leading zeros). If word = 0, the length returned by ASCII is 1.

For decimal conversions, *word* is considered as a 16-bit, 2's complement integer ranging from −32768 to +32767. If the value of *word* is negative, the first byte of *string* contains a minus sign. If *word* = 0, only one zero character is returned in *string*. The length returned by ASCII is the total number of characters in string (excluding the sign). If *word* = 0, the length returned by ASCII is 1.

For decimal left-justified conversions (*base* = 10), leading zeros are removed and the numeric ASCII result is left justified in *string*.

For decimal right justified conversions (*base* = −10), the result is right justified with *string* defining the rightmost byte of the field. Leading bytes in *string* are not changed.

CONDITION CODES:

The condition code remains unchanged.

8-21. DASCII

```
    I             DV   IV    BA
numchar:=DASCII(word,base,string);
```

A 32-bit double-word binary number can be converted to a different base and represented as a numeric character ASCII string with the DASCII intrinsic call.

FUNCTIONAL RETURN:

This intrinsic returns the number of characters in the resulting string.

PARAMETERS:

dword *double value* (required)
A double-word value indicating the number to be converted to ASCII code.

base *integer value* (required)
An integer indicating octal or decimal conversion.

8 = octal

10 = decimal (left justified)

If any other number is entered in this parameter, the intrinsic causes the user process to abort.

string *byte array* (required)
The byte array into which the converted value is placed. This array must be long enough to contain the result. No result, however, exceeds 11 characters.

For octal conversion (*base* = 8), 11 characters, including leading zeros, are always returned in *string*, showing the octal representation of *dword*. The length returned by DASCII is the number of significant (right justified) characters in *string*, excluding leading zeros. If *dword* = 0, the length returned by DASCII is 1.

For decimal conversions (*base* = 10), *dword* is considered as a 32-bit, 2's complement integer ranging from −2,147,483,648 to +2,147,483,647. Leading zeros are removed and the numeric DASCII result is left justified in *string*. If the value of *dword* is negative, the first byte of the *string* returned contains a minus sign. If *dword* = 0, only one zero character is returned to *string*. *String* can contain up to 11 characters, including the sign. If *dword* = 0, the length returned by DASCII is 1.

CONDITION CODES:

The condition code remains unchanged.

8-22. BINARY

```
   L               BA    IV
bineqv:=BINARY(string,length);
```

The BINARY intrinsic converts a number from an ASCII string to a binary word.

FUNCTIONAL RETURN:

This intrinsic returns the binary equivalent of the numeric string.

PARAMETERS:

string *byte array* (required)
Contains the octal or signed decimal number (ASCII characters) to be converted. If the character string in this array begins with a percent sign (%), it is treated as an octal value. If the string begins with a plus sign, minus sign, or a number, it is treated as a decimal value.

length *integer value* (required)
An integer representing the length (number of bytes) in the byte array containing the ASCII-coded value. If the value of *length* is 0, the intrinsic returns 0 to the calling process. If the value of *length* is less than 0, the intrinsic causes the user process to abort.

CONDITION CODES:

CCE	Successful conversion. A one-word binary value is returned to the user's process.
CCG	A word overflow, possibly resulting from too many characters (*string* number too large), occurred in the word returned.
CCL	An illegal character was encountered in the byte array specified by *string*. For example, the digits 8 or 9 specified in an octal value.

8-23. DBINARY

```
    D                BA    IV
dval:=DBINARY(string,length);
```

The DBINARY intrinsic performs double-integer ASCII to binary conversion.

FUNCTIONAL RETURN:

This intrinsic returns the converted double-word value.

PARAMETERS:

string *byte array* (required)
Contains the octal or signed decimal number as ASCII characters to be converted. If the character string in this array begins with a percent sign (%), it is treated as an octal value. If the string begins with a plus sign, minus sign, or number, it is treated as a decimal value.

length *integer value* (required)
An integer representing the length (number of bytes) in the string containing the ASCII-coded value. If the value of *length* is 0, the intrinsic returns 0 to the calling process. If the value of *length* is less than 0, the intrinsic causes the user process to abort.

CONDTION CODES:

CCE	Successful conversion. A double-word binary value is returned to the program.
CCG	A word overflow, possibly resulting from too many characters (*string* number too large), occurred in the word returned.
CCL	An illegal character was encountered in *string*. For example, the digits 8 or 9 specified in an octal value.

8-24. 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

9-1. 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.
$TRACE	Specifies identifiers to be traced at run-time.
$COPYRIGHT	Specifies copyright information to be copied to the list, USL, and program files.

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, CODE, NOCODE, LINES=*nnnn,* ERRORS =*nnn,* USLINIT, 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.

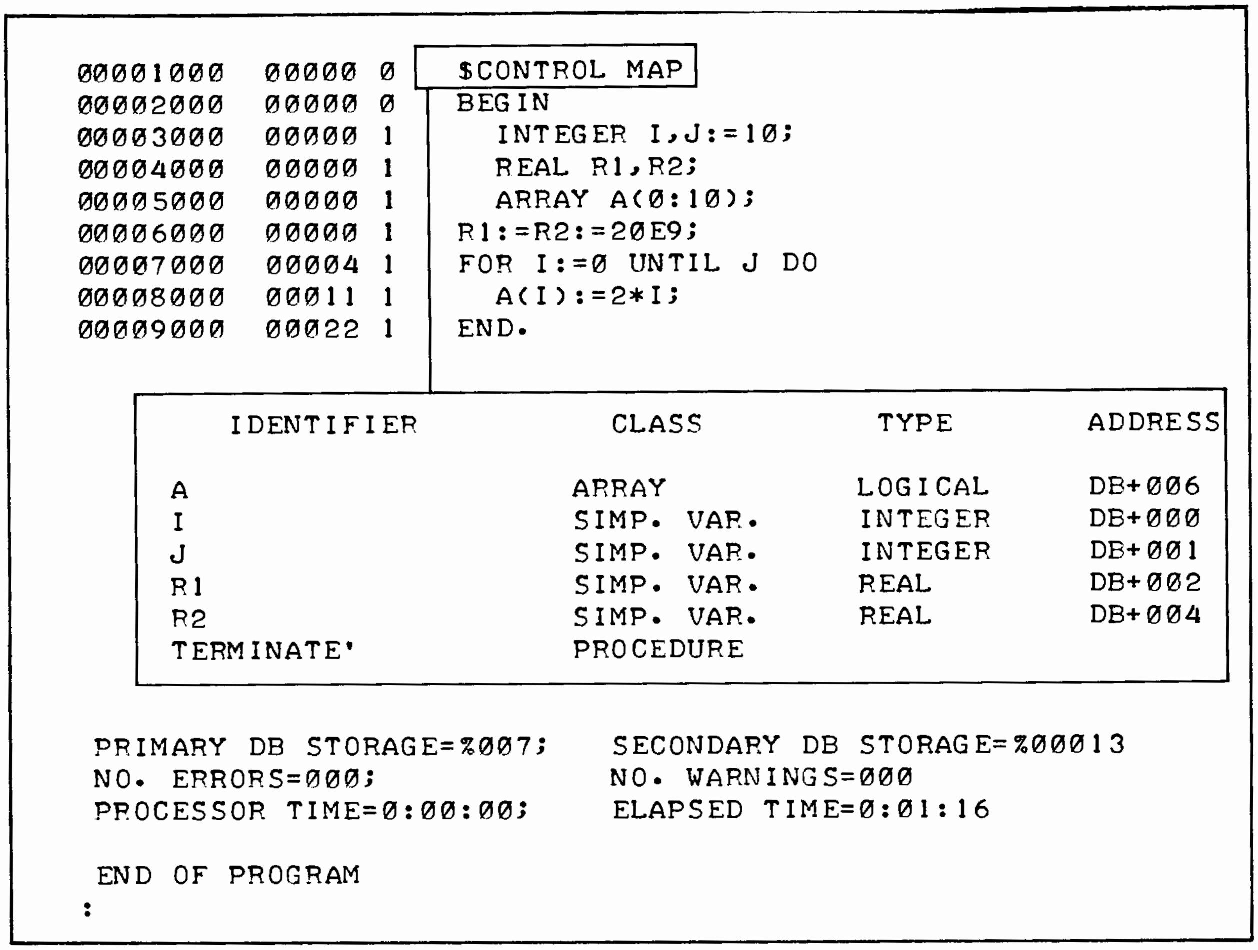
NOTE

> 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. 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.



```
ØØØØ1ØØØ  ØØØØØ Ø   $CONTROL MAP
ØØØØ2ØØØ  ØØØØØ Ø   BEGIN
ØØØØ3ØØØ  ØØØØØ 1     INTEGER I,J:=1Ø;
ØØØØ4ØØØ  ØØØØØ 1     REAL R1,R2;
ØØØØ5ØØØ  ØØØØØ 1     ARRAY A(Ø:1Ø);
ØØØØ6ØØØ  ØØØØØ 1   R1:=R2:=2ØE9;
ØØØØ7ØØØ  ØØØØ4 1   FOR I:=Ø UNTIL J DO
ØØØØ8ØØØ  ØØØ11 1     A(I):=2*I;
ØØØØ9ØØØ  ØØØ22 1   END.

      IDENTIFIER              CLASS          TYPE        ADDRESS

   A                       ARRAY          LOGICAL      DB+ØØ6
   I                       SIMP. VAR.     INTEGER      DB+ØØØ
   J                       SIMP. VAR.     INTEGER      DB+ØØ1
   R1                      SIMP. VAR.     REAL         DB+ØØ2
   R2                      SIMP. VAR.     REAL         DB+ØØ4
   TERMINATE'              PROCEDURE

 PRIMARY DB STORAGE=%ØØ7;    SECONDARY DB STORAGE=%ØØØ13
 NO. ERRORS=ØØØ;             NO. WARNINGS=ØØØ
 PROCESSOR TIME=Ø:ØØ:ØØ;     ELAPSED TIME=Ø:Ø1:16

 END OF PROGRAM
:
```

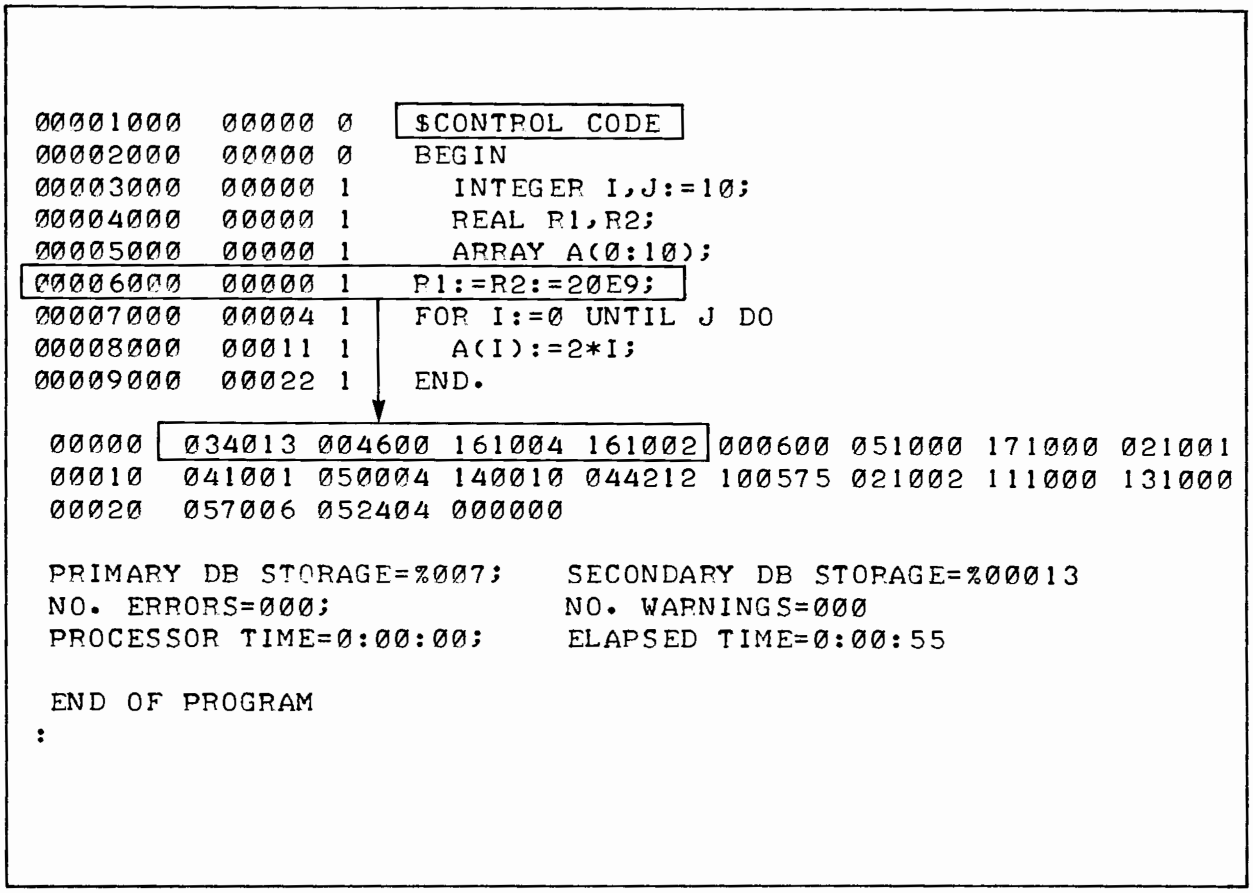
Figure 9-1. Symbol Map

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 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.



```
00001000  00000 0   $CONTROL CODE
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.

 00000  034013 004600 161004 161002 000600 051000 171000 021001
 00010  041001 050004 140010 044212 100575 021002 111000 131000
 00020  057006 052404 000000

 PRIMARY DB STORAGE=%007;     SECONDARY DB STORAGE=%00013
 NO. ERRORS=000;              NO. WARNINGS=000
 PROCESSOR TIME=0:00:00;      ELAPSED TIME=0:00:55

 END OF PROGRAM
:
```

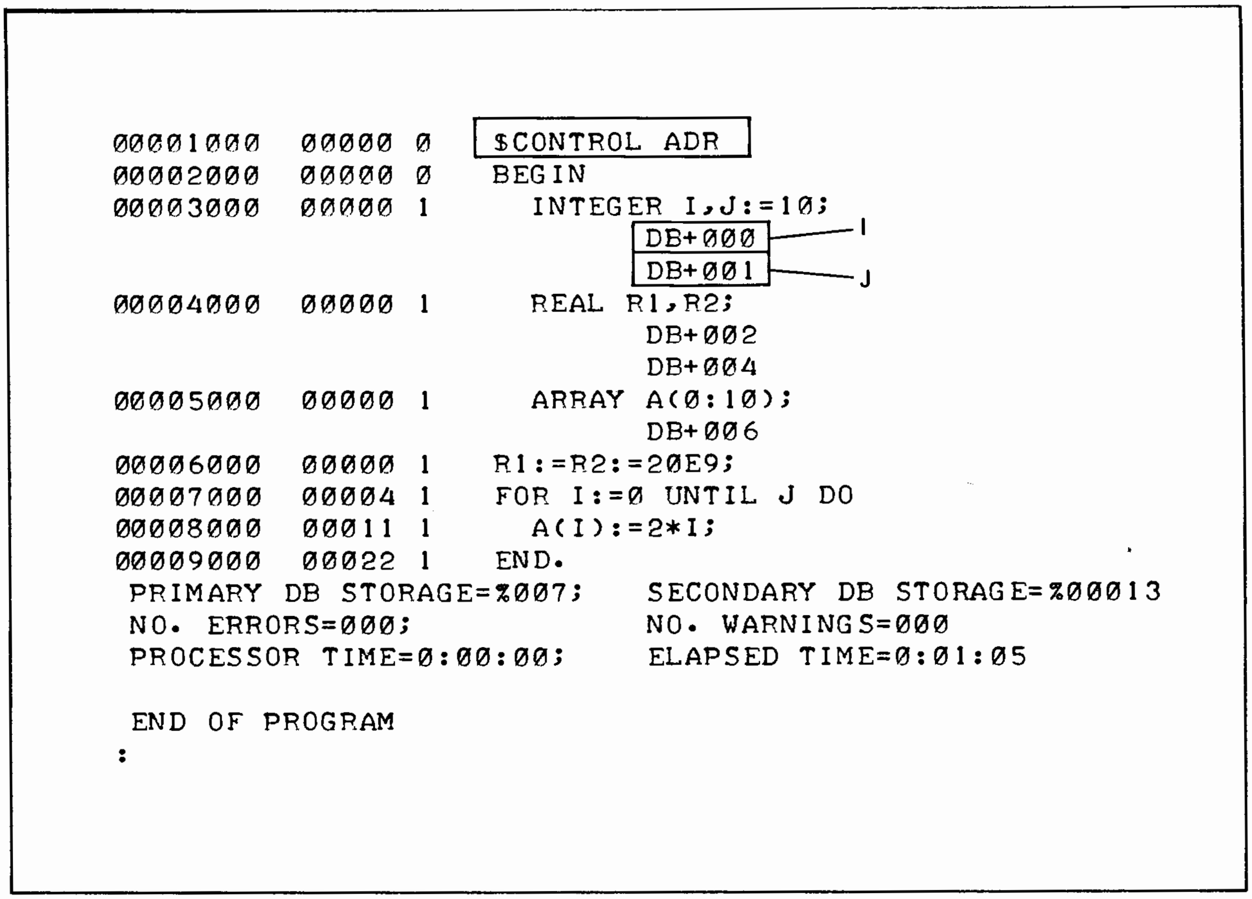
Figure 9-2. $CONTROL CODE Output

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.

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.



```
00001000  00000 0   $CONTROL ADR
00002000  00000 0   BEGIN
00003000  00000 1     INTEGER I,J:=10;
                            DB+000
                            DB+001
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

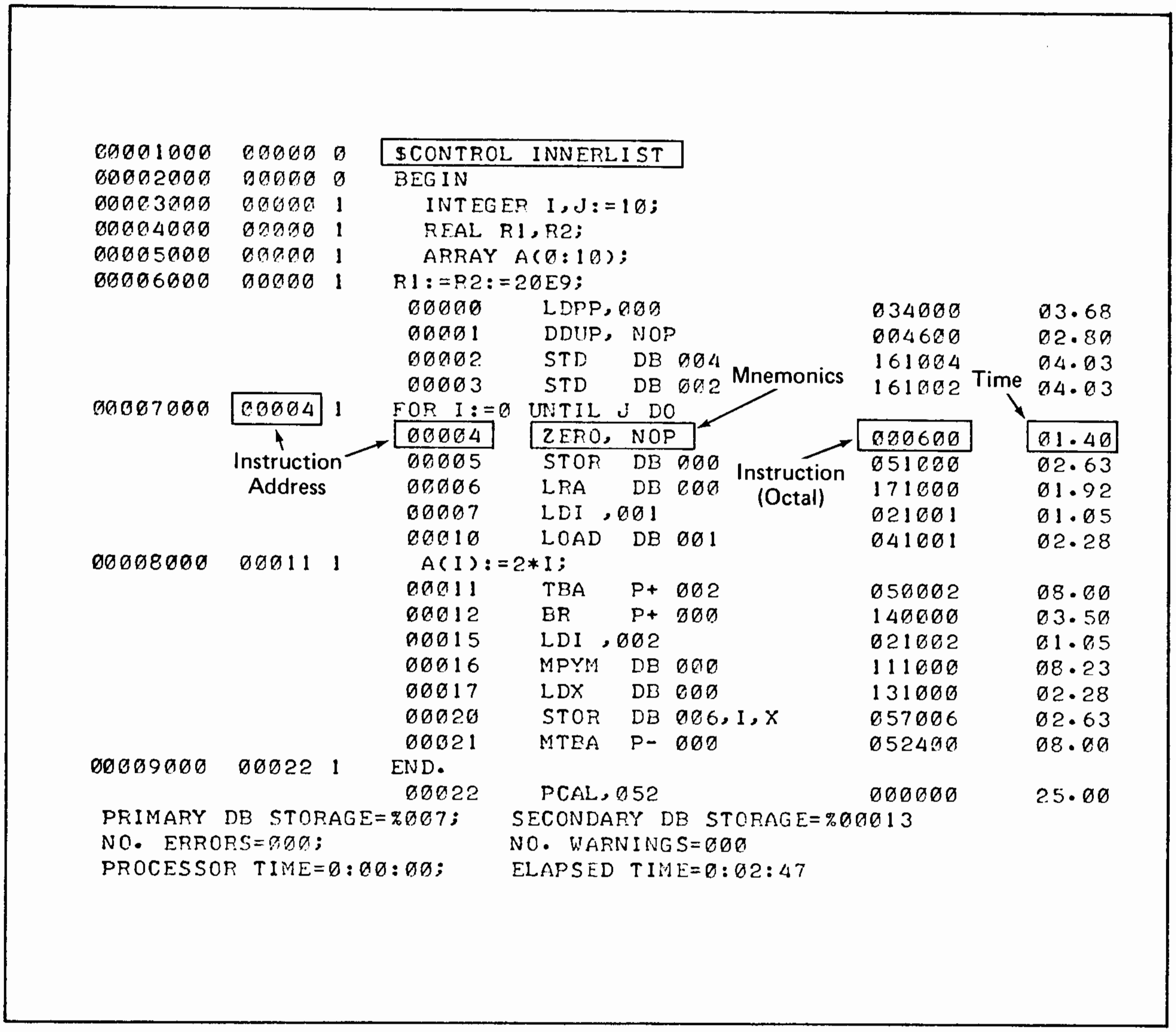
INNERLIST
After each statement line, the mnemonics for unoptimized code generated by the compiler are sent to the *listfile* if LIST is in effect. In addition to the mnemonic, the octal value and approximate execution time in microseconds of each instruction are shown. This option is turned off by NOLIST. Figure 9-4 shows a sample INNERLIST output.

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.

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.

SUBPROGRAM [*(procedure-name*[*] [,...,*procedure-name*[*]]*)*]
Places the compiler is 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.



```
00001000  00000 0  $CONTROL INNERLIST
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;
                     00000      LDPP,000                034000     03.68
                     00001      DDUP, NOP               004600     02.80
                     00002      STD   DB 004            161004     04.03
                     00003      STD   DB 002            161002     04.03
00007000  00004 1   FOR I:=0 UNTIL J DO
                     00004      ZERO, NOP               000600     01.40
                     00005      STOR  DB 000            051000     02.63
                     00006      LRA   DB 000            171000     01.92
                     00007      LDI  ,001               021001     01.05
                     00010      LOAD  DB 001            041001     02.28
00008000  00011 1     A(I):=2*I;
                     00011      TBA   P+ 002            050002     08.00
                     00012      BR    P+ 000            140000     03.50
                     00015      LDI  ,002               021002     01.05
                     00016      MPYM  DB 000            111000     08.23
                     00017      LDX   DB 000            131000     02.28
                     00020      STOR  DB 006,I,X        057006     02.63
                     00021      MTBA  P- 000            052400     08.00
00009000  00022 1   END.
                     00022      PCAL,052                000000     25.00
 PRIMARY DB STORAGE=%007;     SECONDARY DB STORAGE=%00013
 NO. ERRORS=000;              NO. WARNINGS=000
 PROCESSOR TIME=0:00:00;      ELAPSED TIME=0:02:47
```

Figure 9-4. $CONTROL INNERLIST Output

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:

$$\$[\$]\text{IF}\left[\text{Xn=}\begin{Bmatrix}\text{OFF}\\ \text{ON}\end{Bmatrix}\right]$$

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:

$$\$[\$]\text{SET}\left[\text{Xn=}\begin{Bmatrix}\text{OFF}\\ \text{ON}\end{Bmatrix}\left[\text{,Xn=}\begin{Bmatrix}\text{OFF}\\ \text{ON}\end{Bmatrix}\right]\ldots\right]$$

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.

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. $TRACE COMMAND

The $TRACE command specifies identifiers within the outer block or procedures to be traced at run-time. The tracing is implemented through calls to the SYMBOL TRACE subsystem. This subsystem allows references to variables, arrays, pointers, labels, and procedures to be monitored with appropriate printout and breakpoints. For further details, refer to the TRACE/3000 Reference Manual.

The form of the $TRACE command is:

$[$]TRACE [*program-unit*] ;*identifier*[,*identifier*]...

EXAMPLES:

```
$TRACE PROC1;A,B,C
$$TRACE ;A1,B1,C1
```

where

program-unit
specifies the procedure where the identifiers will be traced. If a *program-unit* is not specified, the main program or outer block is used.

identifier
specifies the item to be traced. The *identifier* is a *simple-variable, array-name, pointer-name, label,* or *procedure-name*.

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*]...]

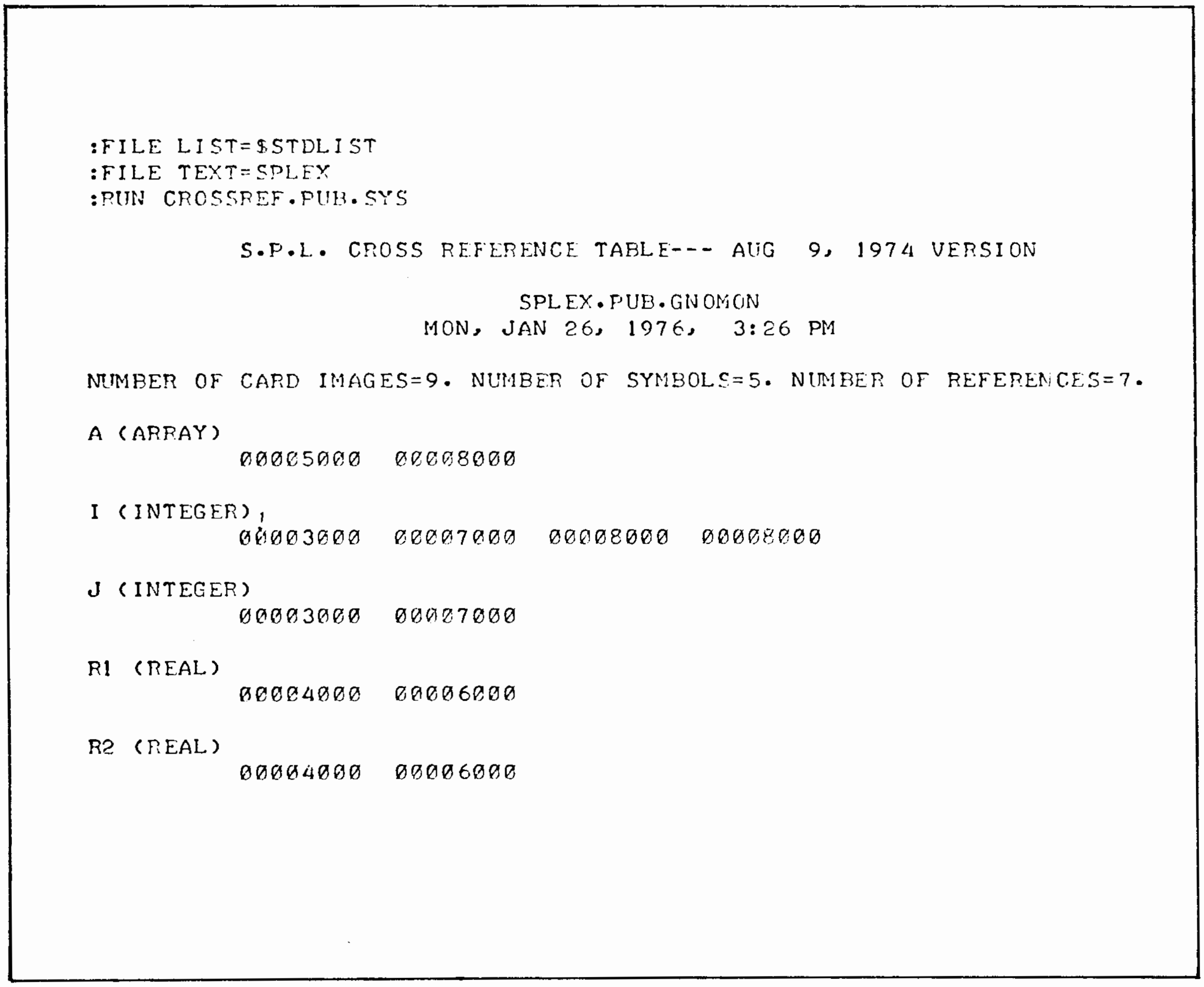
EXAMPLE:

```
$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.



```
: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

*The CROSSREF program is available through the HP 3000 Contributed Library package offered by HP General Systems Division. Contact your local HP Sales Office for more information.

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 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*]]]]

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*).

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)*

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:

11	12	13	14	15
newfile	*masterfile*	*uslfile*	*listfile*	*textfile*

For example, to invoke the compiler with the *textfile* and *listfile* present, you would use the command:

```
:RUN SPL.PUB.SYS;PARM=3
```

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*

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*]]]]

EXAMPLES:

```
:SPLPREP MYSOURCE,MYPROG,*LP
:SPLPREP MYSOURCE,,,MAST
```

where

textfile, listfile, masterfile, newfile
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*]]]

EXAMPLES:

```
:SPLGO MYSOURCE,*LP
:SPLGO MYSOURCE,,MAST
```

where

textfile, listfile, masterfile, newfile
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*.

APPENDIX

A

ASCII CHARACTER SET

BYTE POSITION			
CHAR	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

BYTE POSITION			
CHAR	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

APPENDIX B

RESERVED WORDS

The following symbols have special meaning in SPL/3000 and thus, cannot be used as identifiers:

ABSOLUTE
ALPHA
AND
ARRAY
ASSEMBLE
BEGIN
BYTE
CARRY
CASE
CAT
CHECK
COMMENT
DABZ
DDEL
DEFINE
DEL
DELB
DO
DOUBLE
DXBZ
ELSE
END
ENTRY
EQUATE
EXTERNAL
FALSE
FIXR
FIXT
FOR
FORWARD
GLOBAL
GO
GOTO
IABZ
IF
INTEGER
INTERNAL
INTERRUPT
INTRINSIC
IXBZ
LABEL
LAND
LOGICAL
LONG
LOR
MOD
MODD
MOVE
NOCARRY
NOT
NOVERFLOW
NUMERIC
OF
OPTION
OR
OVERFLOW
OWN
POINTER
PRIVILEGED
PROCEDURE
PUSH
REAL
RETURN
SCAN
SET
SPECIAL
STEP
SUBROUTINE
SWITCH
THEN
TO
TOS
TRUE
UNCALLABLE
UNTIL
VALUE
VARIABLE
WHILE
XOR

BUILDING AN INTRINSIC FILE

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:

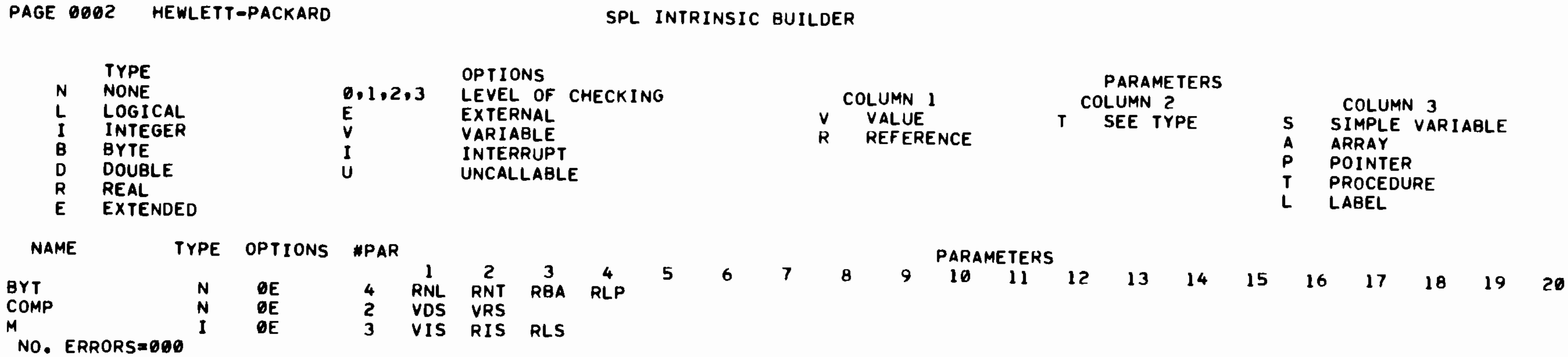
$PURGE	Removes all entries from the intrinsic file.
$REMOVE	Removes all entries which follow this command, until a $BUILD. Input has the same format as for adding entries.
$BUILD	Adds all subsequent input entries to the intrinsic file. $BUILD is required only if $REMOVE is used.

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

Table D-1. Summary of MPE Intrinsics

INTRINSIC NAME	PURPOSE	CAPABILITY REQUIRED
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
CLOCK	Returns the actual time.	Standard
COMMAND	Executes an MPE command programmatically.	Standard
CREATE	Creates a process.	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
DBINARY	Converts a number from ASCII code to a double-word binary value.	Standard
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
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

Table D-1. Summary of MPE Intrinsics (Continued)

INTRINSIC NAME	PURPOSE	CAPABILITY REQUIRED
FGETINFO	Requests access and status information about a file.	Standard
FLOCK	Dynamically locks a file.	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
FREADDIR	Reads a logical record from a direct-access file to the user's data stack.	Standard
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's file label.	Standard
GETDSEG	Creates an extra data segment.	Data-Segment Management
GETJCW	Fetches contents of job control word.	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

Table D-1. Summary of MPE Intrinsics (Continued)

INTRINSIC NAME	PURPOSE	CAPABILITY REQUIRED
GETPROCID	Requests PIN of a son process.	Process Handling
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
IOWAIT	Initiates completion operations for an I/O request.	Standard
KILL	Deletes a process.	Process Handling
LOADPROC	Dynamically loads a library procedure.	Standard
LOCKGLORIN	Locks a global RIN.	Standard
LOCKLOCRIN	Locks a local RIN.	Standard
MAIL	Tests mailbox status.	Process Handling
MYCOMMAND	Parses (delineates and defines parameters) for user-supplied command image.	Standard
PAUSE	Suspends calling process for a specified number of seconds.	Standard
PRINT	Prints character string on job/session list device.	Standard
PRINTOP	Prints a character string on the Operator's Console.	Standard
PRINTOPREPLY	Prints character string on 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
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
RESETCONTROL	Resets terminal to accept CONTROL-Y signal.	Standard
SEARCH	Searches an array for a specified entry or name.	Standard

Table D-1. Summary of MPE Intrinsics (Continued)

INTRINSIC NAME	PURPOSE	CAPABILITY REQUIRED
SENDMAIL	Sends mail to another process.	Process Handling
SETJCW	Sets bits in job control word.	Standard
SUSPEND	Suspends a process.	Process Handling
SWITCHDB	Switches DB-register pointer.	Privileged Mode
TERMINATE	Terminates a process.	Standard
TIMER	Returns system timer bit count.	Standard
UNLOADPROC	Dynamically unloads a library procedure.	Standard
UNLOCKGLORIN	Unlocks a global RIN.	Standard
UNLOCKLOCRIN	Unlocks a local RIN.	Standard
WHO	Returns user attributes.	Standard
XARITRAP	Arms or disarms the software arithmetic trap.	Standard
XCONTRAP	Arms or disarms the CONTROL-Y trap.	Standard
XLIBTRAP	Arms or disarms the software library trap.	Standard
XSYSTRAP	Arms or disarms the system trap.	Standard
ZSIZE	Changes size of Z to DB area.	Standard

APPENDIX E

COMPILER ERROR MESSAGES

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.	
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 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 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.
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.

Table E-1. SPL Compiler Error Messages (Continued)

MESSAGE	MEANING	ACTION
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.
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 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.

Table E-1. SPL Compiler Error Messages (Continued)

MESSAGE	MEANING	ACTION
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.
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 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.

Table E-1. SPL Compiler Error Messages (Continued)

MESSAGE	MEANING	ACTION
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.
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.

Table E-1. SPL Compiler Error Messages (Continued)

MESSAGE	MEANING	ACTION
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.
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.

Table E-1. SPL Compiler Error Messages (Continued)

MESSAGE	MEANING	ACTION
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.
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.

Table E-1. SPL Compiler Error Messages (Continued)

MESSAGE	MEANING	ACTION
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.
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.

Table E-1. SPL Compiler Error Messages (Continued)

MESSAGE	MEANING	ACTION
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.
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.
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.	Correct to address within accepted bounds possibly by removing declarations.

Table E-1. SPL Compiler Error Messages (Continued)

MESSAGE	MEANING	ACTION
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.
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).

Table E-1. SPL Compiler Error Messages (Continued)

MESSAGE	MEANING	ACTION
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.
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 @.

APPENDIX F

SPL SYNTAX

```
 1. <program>                   ::= BEGIN <declarations> <main body>END.|
                                    BEGIN <main body> END.
 2. <subprogram>                ::= BEGIN <declarations> END.
 3. <main body>                 ::= <statement>|
                                    <statement>;<main body>
 4. <declarations>              ::= <data group> <sub proc group>|
                                    <data group>|
                                    <sub proc group>
 5. <data group>                ::= <data declaration>;|
                                    <data declaration>;<data group>
 6. <data declaration>          ::= <define dec>|
                                    <equate dec>|
                                    <global simple var dec>|
                                    <global array dec>|
                                    <global pointer dec>
                                    <label declaration>|
                                    <switch declaration>|
                                    <entry declaration>
 7. <sub proc group>            ::= <proc int group><global sub group>|
                                    <proc int group>|
                                    <global sub group>
 8. <proc int group>            ::= <proc int dec>;|
                                    <proc int dec>;<proc int group>
 9. <proc int dec>              ::= <procedure declaration>|
                                    <intrinsic declaration>
10. <global sub group>          ::= <subroutine dec>;|
                                    <subroutine dec>;<global sub group>
11. <define dec>                ::= DEFINE <define list>
12. <define list>               ::= <define element>|
                                    <define element>,<define list>
13. <define element>            ::= <identifier>=<def text>#
14. <def text>                  ::= <def char>|
                                    <def char> <def text>
15. <def char>                  ::= {any ASCII character except #}
16. <equate dec>                ::= EQUATE <equate list>
17. <equate list>               ::= <equate element>|
                                    <equate element>,<equate list>
18. <equate element>            ::= <equate identifier>=<equate expression>
19. <equate expression>         ::= <sign> <equate term>|
                                    <equate term>|
                                    <equate expression> <addop> <equate term>
20. <equate term>               ::= <equate primary>|
                                    <equate term> <muldiv> <equate primary>
21. <equate primary>            ::= <unsigned integer>|
                                    <equate identifier>|
                                    (<equate expression>)
22. <equate identifier>         ::= <identifier>
23. <sign>                      ::= + | -
24. <addop>                     ::= + | -
25. <muldiv>                    ::= * | /
26. <global simple var dec>     ::= <type> <simple var list>|
                                    GLOBAL <type> <simple var list>
27. <type>                      ::= INTEGER|LOGICAL|BYTE|DOUBLE|REAL|LONG
28. <simple var list>           ::= <simple var element>|
                                    <simple var element>,<simple var list>
```

```
29. <simple var element>        ::=<simple variable>|
                                   <simple variable>:=<initial value>|
                                   <simple variable>=<id reg reference>
30. <simple variable>           ::=<identifier>
31. <local simple var dec>      ::=<type> <simple var list>|
                                   OWN <type> <nonref var dec list>|
                                   EXTERNAL <type> <simple variable list>
32. <nonref var dec list>       ::=<nonref var dec>|
                                   <nonref var dec>,<nonref var dec list>
33. <nonref var dec>            ::=<simple variable>|
                                   <simple variable>:=<initial value>
34. <initial value>             ::=<constant>
35. <id reg reference>          ::=<variable reference>|
                                   <register reference>
36. <variable reference>        ::=<data identifier>|
                                   <data identifier> <sign> <offset>
37. <data identifier>           ::=<simple variable>|
                                   <array name>|
                                   <pointer name>
38. <offset>                    ::=<unsigned integer>
39. <register reference>        ::=X|
                                   DB+<offset>|
                                   Q+<offset>|
                                   Q−<offset>|
                                   S−<offset>
40. <constant>                  ::=<number>|<string>
41. <number>                    ::=<integer>|
                                   <real number>|
                                   <double integer>|
                                   <long real number>|
                                   <logical value>
42. <integer>                   ::=<unsigned integer>|
                                   <sign> <unsigned integer>
43. <decimal integer>           ::=<digit>|
                                   <decimal integer> <digit>
44. <unsigned integer>          ::=<decimal integer>|
                                   <based integer>|
                                   <composite integer>|
                                   <equate invocation>
45. <equate invocation>         ::=<equate identifier>
46. <digit>                     ::=0|1|2|3|4|5|6|7|8|9
47. <based integer>             ::=%<base digit>|
                                   %<base part> <base digit>|
                                   <based integer> <base digit>
48. <base part>                 ::=(<base>)
49. <base>                      ::={any unsigned integer from 2 to 16 inclusive}
50. <base digit>                ::={any digit from 0 to <base>−1 inclusive, taken
                                   from the set (0,1,2,3,4,5,6,7,8,9,A,B,C,D,
                                   E,F)}
51. <composite integer>         ::=[<integer field list>]
52. <integer field list>        ::=<integer field>|
                                   <integer field>,<integer field list>
53. <integer field>             ::=<number of bits>/<unsigned integer>
54. <number of bits>            ::=<unsigned integer>
55. <double integer>            ::=<integer> D
56. <real number>               ::=<unsigned real number>|
                                   <sign> <unsigned real number>
```

```
57. <unsigned real number>      ::=<fraction>|
                                   <decimal integer>E<power>|
                                   <fraction>E<power>|
                                   <composite integer>E|
                                   <based integer> E
58. <fraction>                  ::=<decimal integer>.|
                                   <digit>|
                                   <fraction> <digit>
59. <power>                     ::=<decimal integer>|
                                   <sign> <decimal integer>
60. <long real number>          ::=<unsigned long real no>|
                                   <sign> <unsigned long real no>
61. <unsigned long real no>     ::=<decimal integer>L<power>|
                                   <fraction>L<power>|
                                   <composite integer>L|
                                   <based integer>L
62. <logical value>             ::=TRUE|
                                  FALSE|
                                   <integer>
63. <string>                    ::="<character string>"
64. <character string>          ::=<character>|
                                   <character string> <character>
65. <character>                 ::={a member of the ASCII character set}
66. <identifier>                ::=<letter>|
                                   <identifier> <letter>|
                                   <identifier> <digit>|
                                   <identifier>'
67. <letter>                    ::=A|B|C|D|
                                 E|F|G|H|I|J|K|
                                 L|M|
                                   N|O|P|Q|R|S|
                                 T|U|V|W|
                                 X|Y|Z
68. <expression>                ::=<arithmetic expression>|
                                   <logical expression>|
69. <variable>                  ::=<simple variable>|
                                   <pointer name>|<pointer name> <index>|
                                   <array name>|<array name> <index>|
                                   TOS
70. <integer variable>          ::=@<simple variable>|
                                   @<pointer name>|@<pointer name> <index>|
                                   @<array name>|@<array name> <index>|
                                   @<label>|
                                   @<procedure name>|
                                   @<entry point>|
                                   ABSOLUTE|ABSOLUTE<index>
71. <index>                     ::=(<expression>)|
                                   (<assignment statement>)
72. <function designator>       ::=<procedure id>|
                                   <procedure id> <actual param part>|
                                   <subroutine name>|
                                   <subroutine name> <actual sparam part>
73. <procedure id>              ::=<procedure name>|
                                   <entry point>
74. <bit operation>             ::=<bit extraction>|
                                   <bit concatenation>|
                                   <bit shift>
75. <bit extraction>            ::=<primary>.(<bit extract field>)
76. <bit extract field>         ::=<left extract bit>:<extract field length>
77. <left extract bit>          ::=<unsigned integer>
78. <extract field length>      ::=<unsigned integer>
```

```
79. <bit concatenation>          ::=<primary> CAT <primary>(<bit cat field>)
80. <bit cat field>              ::=<left deposit bit>:<bit extract field>
81. <left deposit bit>           ::=<unsigned integer>
82. <bit shift>                  ::=<primary>&<shift op>(<shift count>)
83. <shift op>                   ::=LSL|LSR|ASL|ASR|
                                    CSL|CSR|DASL|DASR|DLSL|
                                    DLSR|DCSL|DCSR|TASL|TASR|TNSL|
                                    QASL|QASR
84. <shift count>                ::=<integer expression>
85. <integer expression>         ::=<arithmetic expression>
86. <arithmetic expression>      ::=<aexp>|
                                    <addop> <aexp>|
                                    <IF expression>
87. <aexp>                       ::=<term>|
                                    <aexp> <addop> <term>
88. <term>                       ::=<factor>|
                                    <term> <mulop> <factor>
89. <factor>                     ::=<primary>|
                                    <factor>-<primary>
90. <primary>                    ::=<variable>|
                                    <constant>|
                                    <bit operation>|
                                    (<aexp>)|
                                    \<aexp>\|
                                    <function designator>|
                                    <assignment statement>
91. <mulop>                      ::=*|/|MOD
92. <logical expression>         ::=<lexp>|
                                    <IF expression>
93. <IF expression>              ::=IF <condition clause> THEN <expression> ELSE
                                    <expression>
94. <lexp>                       ::=<disjunction>|
                                    <lexp> LOR <disjunction>|
                                    <integer expression> <= <integer expression>
                                    <= <integer expression>
95. <disjunction>                ::=<conjunction>|
                                    <disjunction> XOR <conjunction>
96. <conjunction>                ::=<logical element>|
                                    <conjunction> LAND <logical element>
97. <logical element>            ::=<logical term>|
                                    <logical term> <relop> <logical term>|
                                    <arithmetic expression> <relop>
                                    <arithmetic expression>|
                                    <byte ref> <relop> <byte ref> <count>|
                                    <byte ref> <relop> <byte ref>
                                    <count>,<sdec>|
                                    <byte ref> <relop>*PB<count>|
                                    <byte ref> <relop>*PB<count>,<sdec>|
                                    <byte ref> <relop> <string>|
                                    <byte ref> <relop> <string>,<sdec>|
                                    <byte ref> <relop>(<listelmt>)|
                                    <byte ref> <relop>(<listelmt>),<sdec>|
                                    <byte variable> = <btestword>|
                                    <byte variable> <> <btestword>
98. <logical term>               ::=<logical factor>|
                                    <logical term> <addop> <logical factor>
99. <logical factor>             ::=<logical primary>|
                                    <logical factor> <logical mulop>
                                    <logical primary>
```

```
100.  <logical primary>            ::=<logical variable>|
                                      <logical value>|
                                      <string>|
                                      <logical bit operation>|
                                      (<lexp>)|
                                      <logical func desig>|
                                      (<logical assign stmt>)|
                                      NOT <logical primary>
101.  <byte variable>              ::=<variable>
102.  <logical variable>           ::=<variable>
103.  <logical bit operation>      ::=<bit operation>
104.  <logical func desig>         ::=<function designator>
105.  <logical assign stmt>        ::=<assignment statement>
106.  <relop>                      ::=<>|=|<|>|<=|>=
107.  <logical mulop>              ::=*|/|MOD|**|//|MODD
108.  <byte ref>                   ::=<byte pointer name>|
                                      <byte pointer name> <index>|
                                      <byte array name>|
                                      <byte array name> <index>|
                                      *
109.  <byte pointer name>          ::=<pointer name>
110.  <byte array name>            ::=<array name>
111.  <btestword>                  ::=ALPHA|NUMERIC|SPECIAL
112.  <global array dec>           ::=ARRAY<g-array dec list>|
                                      <type>ARRAY<g-array dec list>|
                                      GLOBAL ARRAY <G-dec list>|
                                      GLOBAL <type> ARRAY <G-dec list>
113.  <type>                       ::=INTEGER|LOGICAL|BYTE|DOUBLE|REAL|
                                     LONG
114.  <local array dec>            ::=ARRAY <l-array dec list>|
                                      <type> ARRAY <l-array dec list>|
                                      EXTERNAL ARRAY <E-dec list>|
                                      EXTERNAL <type> <E-dec list>|
                                      OWN ARRAY <own array dec list>|
                                      OWN <type> ARRAY <own array dec list>
115.  <G-dec>                      ::=<array name>(<db>)|
                                      <array name>(<db>)=DB|
                                      <array name>(*)=DB|
                                      <array name>(@)=DB
116.  <G-dec initial>              ::=<array name>(<db>):=<array init>
                                      <array name>(<db>)=DB:=<array init>
117.  <G-dec list>                 ::=<G-dec initial>|
                                      <G-dec list1>|
                                      <G-dec list1>,<G-dec initial>
118.  <G-dec list1>                ::=<G-dec>|
                                      <G-dec>,<G-dec list1>
119.  <g-array dec list>           ::=<G-dec initial>|
                                      <g-array dec list1>|
                                      <g-array dec list1>,<G-dec initial>
120.  <g-array dec>                ::=<G-dec>|
                                      <array name>(@)<indirect base reg ref>|
                                      <array name>(<udb>)<reference part>
121.  <array name>                 ::=<identifier>

122.  <db>                         ::=<lower>:<upper>
123.  <lower>                      ::=<integer>
124.  <upper>                      ::=<integer>
125.  <udb>                        ::=*
126.  <E-dec list>                 ::=<E-dec>|
                                      <E-dec>,<E-dec list>
127.  <E-dec>                      ::=<array name>(<udb>)|<array name>(@)
```

```
128. <l-array dec list>           ::=<l-array initial>|
                                     <l-array dec list1>
129. <l-array dec list1>          ::=<l-array dec>|
                                     <l-array dec>,<l-array dec list1>
130. <l-array initial>            ::=<array name> <db>=PB:=<array init>
131. <l-array dec>                ::=<array name>(<db>)|
                                     <array name>(<db>)=Q|
                                     <array name>(*)=Q|
                                     <array name>(@)=Q|
                                     <array name>(<vb>)|
                                     <array name>(<udb>)<reference part>|
                                     <array name>(@)<indirect base reg ref>
132. <own array dec list>         ::=<own array initial>|
                                     <own array dec list1>|
                                     <own array dec list1>,<own array initial>
133. <own array dec list1>        ::=<own array dec>|
                                     <own array dec>,<own array dec list1>
134. <own array dec>              ::=<array name>(<db>)
135. <own array initial>          ::=<array name>(<db>):=<array init>
136. <vb>                         ::=<lower variable>:<upper variable>
137. <lower variable>             ::=<simple variable>
138. <upper variable>             ::=<simple variable>
139. <reference part>             ::=<var reference>|
                                     =<indexed ident ref>
140. <indexed ident ref>          ::=<array name>|
                                     <pointer name>|
                                     <array name>(<integer>)|
                                     <pointer name>(<integer>)
141. <array init>                 ::=<listelmt>
142. <listelmt>                   ::=<initial value>|
                                     <repetition>(<initial value list>)|
                                     <listelmt list>
143. <initial value list>         ::=<initial value>|
                                     <initial value>,<initial value list>
144. <listelmt list>              ::=<listelmt>|
                                     <listelmt>,<listelmt list>
145. <repetition>                 ::=<unsigned integer>
146. <var reference>              ::==<base reg ref>|
                                     =<data identifier>|
                                     =<data identifier> <sign> <offset|
147. <indirect base reg ref>      ::==<base reg ref>
148. <base reg ref>               ::=DB+<offset>|
                                     Q+<offset>|
                                     Q-<offset>|
                                     S-<offset>
149. <global pointer dec>         ::=POINTER <pointer dec list>|
                                     GLOBAL POINTER <pointer dec list>|
                                     GLOBAL <atype> POINTER <pointer dec list>|
                                     <type> POINTER <pointer dec list>
150. <local pointer dec>          ::=<type> POINTER <pointer dec list>|
                                     OWN <type> POINTER <pointer name list>|
                                     EXTERNAL <type> POINTER <pointer name list>
151. <pointer dec list>           ::=<pointer dec>|
                                     <pointer dec>,<pointer dec list>
152. <pointer name list>          ::=<pointer name>|
                                     <pointer name>,<pointer name list>
153. <pointer dec>                ::=<pointer name> <pointer init>|
                                     <pointer name> <var reference>
154. <pointer name>               ::=<identifier>
155. <pointer init>               ::=@<address specification>
```

```
156.  <address specification>       ::=<simple variable>|
                                       <indexed ident ref>
157.  <label declaration>           ::=LABEL <label list>
158.  <label list>                  ::=<label>|
                                       <label>,<label list>
159.  <label>                       ::=<identifier>
160.  <switch declaration>          ::=SWITCH <switch name>:=<label list>
161.  <switch name>                 ::=<identifier>
162.  <entry declaration>           ::=ENTRY <entry point list>
163.  <entry point list>            ::=<entry point>|
                                       <entry point>,<entry point list>
164.  <entry point>                 ::=<identifier>
165.  <procedure declaration>       ::=<type> PROCEDURE <proc head> <proc body>|
                                       PROCEDURE <proc head> <proc body>|
                                       <for-ext proc dec>
166.  <for-ext proc dec>            ::=<type> PROCEDURE <proc head>|
                                       PROCEDURE <proc head>
167.  <external proc dec>           ::=<for-ext proc dec>
168.  <proc head>                   ::=<procedure name> <formal part> <option part>|
                                       <procedure name> <formal part>|
                                       <procedure name>;<option part>
169.  <procedure name>              ::=<identifier>
170.  <formal part>                 ::=(<formal param list>);<value part> <spec part>
                                       (<formal param list>);<spec part>
171.  <formal param list>           ::=<formal param>|
                                       <formal param>,<formal param list>
172.  <formal param>                ::=<identifier>
173.  <value part>                  ::=VALUE <formal param list>;
174.  <spec part>                   ::=<specification>;|
                                       <spec part> <specification>;
175.  <specification>               ::=<type> <simple variable list>|
                                       <type>ARRAY <array name list>|
                                       ARRAY <array name list>|
                                       LABEL <label list>|
                                       POINTER <pointer name list>|
                                       <type> POINTER <pointer name list>
                                       PROCEDURE <procedure id list>|
                                       <type> PROCEDURE <procedure id list>
176.  <simple variable list>        ::=<simple variable>|
                                       <simple variable>,<simple variable list>
177.  <array name list>             ::=<array name>|
                                       <array name>,<array name list>
178.  <procedure name list>         ::=<procedure name>|
                                       <procedure name>,<procedure name list>
179.  <procedure id list>           ::=<procedure id>|
                                       <procedure id>,<procedure id list>
180.  <option part>                 ::=OPTION <option list>
181.  <option list>                 ::=<option>|
                                       <option>,<option list>
182.  <option>                      ::=UNCALLABLE|PRIVILEGED|EXTERNAL|CHECK <level>|
                                       VARIABLE|FORWARD|INTERRUPT|INTERNAL
183.  <level>                       ::=0|1|2|3
184.  <proc body>                   ::=<statement>
                                       BEGIN <proc data group> <procedure group>
                                       <compound tail>
185.  <proc data group>             ::=<proc data group> <proc data declaration>;|
                                       <proc data declaration>;
```

```
186. <proc data declaration>        ::=<define declaration>|
                                       <equate declaration>|
                                       <local simple variable dec>|
                                       <local array dec>|
                                       <local pointer dec>|
                                       <label declaration>|
                                       <switch declaration>|
                                       <entry declaration>
187. <procedure group>              ::=<procedure group> <subroutine dec>|
                                       <proc group>
188. <proc group>                   ::=<external proc dec>;|
                                       <proc group> <intrinsic declaration>;
189. <intrinsic declaration>        ::=INTRINSIC <procedure name list>|
                                       INTRINSIC (<file-ref>) <procedure name list>
190. <file-ref>                     ::=<file>|
                                       <file>.<group>|
                                       <file>.<group>.<acct>
191. <file>                         ::=<letter>|
                                       <file> <letter>|
                                       <file> <digit>
192. <group>                        ::=<letter>|
                                       <group> <letter>|
                                       <group> <digit>
193. <acct>                         ::=<letter>|
                                       <acct> <letter>|
                                       <acct> <digit>
194. <subroutine dec>               ::=SUBROUTINE <sub head> <sub body>|
                                       <type>SUBROUTINE <sub head> <sub body>
195. <sub head>                     ::=<subroutine name> <sformal part>|
                                       <subroutine name>
196. <sub body>                     ::=<statement>
197. <subroutine name>              ::=<identifier>
198. <sformal part>                 ::=(<formal param list>);<value part>
                                       <s-spec part>|
                                       (<formal param list>);<s-spec part>
199. <s-spec part>                  ::=<s-specification>;|
                                       <s-spec part> <s-specification>;
200. <s-specification>              ::=<type> <simple variable list>|
                                       ARRAY <array name list>|
                                       <type> ARRAY <array name list>|
                                       POINTER <pointer name list>|
                                       <type> POINTER <pointer name list>|
                                       PROCEDURE <procedure name list>|
                                       <type> PROCEDURE <procedure name list>
201. <statement>                    ::=<label>:<statement>|
                                       <compound statement>|
                                       <assignment statement>|
                                       <GO TO statement>|
                                       <IF statement>|
                                       <CASE statement>|
                                       <FOR statement>|
                                       <DO statement>|
                                       <WHILE statement>|
                                       <MOVE statement>|
                                       <SCAN statement>|
                                       <PROCEDURE call stmt>|
                                       <RETURN statement>|
                                       <SUBROUTINE call stmt>|
                                       <DELETE statement>|
                                       <PUSH statement>|
                                       <SET statement>|
                                       <ASSEMBLE statement>
```

```
202. <compound statement>       ::= BEGIN <compound tail>
203. <compound tail>            ::= <statement> END|
                                     <statement>;<compound tail>
204. <ASSEMBLE statement>       ::= ASSEMBLE(<instruction slist>)
205. <instruction slist>        ::= <instruction>|
                                     <instruction slist>;<instruction>
206. <instruction>              ::= <label>:<opcode format>|
                                     <opcode format>
207. <opcode format>            ::= <format-1>|<format-2>|<format-3>|
                                     <format-4>|<format-5>|<format-6>|
                                     <format-7>|<format-8>|<format-9>|
                                     <format-10>
208. <format-1>                 ::= <memory ref opcode> <address part>|
                                     <memory ref opcode> <address part> <IX fields>|
                                     <sub memref opcode> <label>
209. <memory ref opcode>        ::= <sub memref op>|
                                     STOR|INCM|DECM|LDB|LDD|STB|STD
210. <sub memref op>            ::= LOAD|LDX|LRA|CMPM|ADDM|
                                   SUBM|MPYM|BR|BL|BE|
                                     BLE|BG|BNE|BGE|
                                   TBA|MTBA|TBX|MTBX
211. <address part>             ::= <var identifier>|<addr mode> <offset>
212. <var identifier>           ::= <simple variable>|<pointer name>|<array name>
213. <addr mode>                ::= DB+ | Q+ | Q- |
                                      P+ | P- | S-
214. <IX fields>                ::= <I-field>|<X-field>|<I-field> <X-field>
215. <format-2>                 ::= <stack opcode>|
                                     <stack opcode>,<stack opcode>
216. <stack opcode>             ::= NOP|DELB|DDEL|ZROX|
                                   INCX|DECX|ZERO|DZRO|DCMP|
                                     DADD|DSUB|MPYL|DIVL|DNEG|
                                   DXCH|CMP|ADD|SUB|
                                     MPY|DIV|NEG|TEST|STBX|
                                   DTST|DFLT|BTST|XCH|
                                     INCA|DECA|LDXA|DUP|DDUP|
                                   FLT|FCMP|FADD|FSUB|
                                     FMPY|FDIV|FNEG|CAB|LCMP|
                                   LADD|LSUB|LMPY|LDIV|
                                     NOT|OR|XOR|AND|FIXR|
                                   FIXT|INCB|DECB|XBX|ADBX|
                                     ADXB
217. <format-3>                 ::= <branch subop1> <arg1>|
                                      <branch subop1> <arg1> <I-field>|
                                      <non-branch subop1> <unsigned integer>|
                                      <non-branch subop1> <unsigned integer> <X-field>
218. <I-field>                  ::= ,I
219. <X-field>                  ::= ,X
220. <arg1>                     ::= <label>|
                                      P<sign> <offset>|
                                      *<sign> <offset>
221. <branch subop1>            ::= IABZ|IXBZ|DXBZ|DABZ|
                                   BCY|BNCY|CPRB|BOV|BNOV|
                                      BRO|BRE
222. <non-branch subop1>        ::= ASL|ASR|LSL|LSR|
                                   CSL|CSR|SCAN|TASL|TASR|
                                   TNSL|
                                     DASL|DASR|DLSL|DLSR|DCSL|
                                   DCSR|TBC|TRBC|TSBC|
                                     TCBC|QASL|QASR
223. <format-4>                 ::= <sub op2> <unsigned integer>|
                                      EXF<unsigned integer>:<unsigned integer>|
                                      DPF<unsigned integer>:<unsigned integer>
```

```
224.  <sub op2>                 ::= LDI|LDXI|CMPI|ADDI|
                                 SUBI|MPYI|DIVI|PSHR|
                                 LDNI|
                                   LDXN|CMPN|SETR
225.  <format-5>                ::= RSW|LLSH|PLDA|PSTA|
                                LSEA|SSEA|LDEA|SDEA|IXIT|
                                   LOCK|PCN|UNLK
226.  <format-6>                ::= <special op> <unsigned integer>
227.  <special op>              ::= PAUS|SED|XCHD|SMSK|
                                 RMSK|XEQ|SIO|RIO|
                                   WIO|TIO| CIO|CMD|SIN|
                                 HALT|LST|PSDB|DISP|PSEB|
                                 SCLK|RCLK|SST
228.  <format-7>                ::= <sub op3> <unsigned integer>|
                                   PCAL<procedure id>|
                                   SCAL 0|
                                   LLBL <procedure id>
229. <sub op3>                  ::= PCAL|SCAL|EXIT|SXIT|
                                   ADXI|SBXI|LLBL|LDPP|
                                   LDPN|ADDS|SUBS|ORI|XORI|ANDI
230.  <format-8>                ::= <sub move op>|
                                   <sub move op> PB|
                                   <sub move op>,<sdec-8a>|
                                   <sub move op> PB,<sdec-8a>7
                                   <sub pmove op>|
                                   <sub pmove op>,<sdec-8d>|
                                   MVBW <ccf>|
                                   MVBW <ccf>,<sdec>|
                                   <scan op>,<sdec>
231.  <sdec>                    ::= 0|1|2
232.  <sdec-8a>                 ::= 0|1|2|3
233.  <sdec-8d>                 ::= 0|1|2|3|
                                 4|5
234.  <ccf>                     ::= A|N|AN|AS|ANS
235.  <sub move op>             ::= MOVE|MVB|CMPB
236.  <sub pmove op>            ::= MABS|MTDS|MDS|MFDS
237.  <scan op>                 ::= SCW|SCU|MVBL|MVLB
238.  <format-9>                ::= CON <const list>
239.  <const list>              ::= <const>|
                                   <const>,<const list>
240.  <const>                   ::= <constant>|
                                <label>
241.  <format-10>               ::= <ext arith op>|
                                   <decimal op>
242.  <ext arith op>            ::= DMUL|DDIV|EADD|ESUB|
                                 EMPY|EDIV|ENEG|
                                   ECMP|DMPY
243.  <decimal op>              ::= <decimal conv>|
                                   <decimal arith>
244.  <decimal conv>            ::= <decimal conv inst>|
                                   <decimal conv inst> <sdec-10b>|
                                   <decimal-ASCII>
245.  <decimal conv inst>       ::= CVAD|CVBD|CVDB
246.  <sdec-10b>                ::= 0|1
247.  <decimal-ASCII>           ::= CVDA|
                                   CVDA<sdec-10b>|
                                   CVDA <abs>|
                                   CVDA <abs>,<sdec-10b>
248.  <abs>                     ::= ABS|NABS
249.  <decimal arith>           ::= <decimal arith op>|
                                   <decimal arith op> <sdec>
```

```
250.  <decimal arith op>         ::= ADDD|SUBD|MPYD|CMPD|
                                  SLD|NSLD|SRD
251.  <assignment statement>    ::= <left part>:=<right part>|
                                   <left part>:=<assignment statement>
252. <left part>                ::= <variable>|
                                   <variable>.(<deposit field>)
253.  <deposit field>           ::= <left deposit bit>:<deposit field length>
254.  <left deposit bit>        ::= <unsigned integer>
255.  <deposit field length>    ::= <unsigned integer>
256.  <right part>              ::= <expression>
257.  <CASE statement>          ::= CASE <expression> OF <case body>|
                                   CASE* <expression> OF <case body>
258.  <case body>               ::= <compound statement>
259.  <DELETE statement>        ::= DEL|DELB|DDEL
260.  <DO statement>            ::= DO <statement> UNTIL <condition clause>
261.  <FOR statement>           ::= <FOR clause> <statement>
262.  <FOR clause>              ::= FOR <simple variable>:=<expression>
                                   <STEP clause> UNTIL <expression> DO|
                                   FOR <simple variable>:=<expression>
                                   UNTIL <expression> DO|
                                   FOR* <simple variable>:=<expression>
                                   <STEP clause> UNTIL <expression> DO|
                                   FOR* <simple variable>:=<expression>
                                   UNTIL <expression> DO
263.  <STEP clause>             ::= STEP <expression>
264.  <GO TO statement>         ::= GO <label ref>|
                                   GO TO <label ref>|
                                   GOTO <label ref>
265.  <label ref>               ::= <label>|
                                   <switch name> <index>|
                                   *<switch name> <index>
266.  <IF statement>            ::= IF <condition clause> <THEN part>|
                                   IF <condition clause> <THEN part> <ELSE part>
267.  <condition clause>        ::= <condition element>|
                                   <condition clause> OR <condition element>
268.  <condition element>       ::= <condition term>|
                                   <condtion element> AND <condition term>
269.  <condition term>          ::= <condition primary>|
                                   (<condition factor>)
270.  <condition factor>        ::= <condition primary> OR <condition factor>|
                                   <condition primary> OR <condition primary>
271.  <condition primary>       ::= <branch word>|
                                   <logical expression>
272.  <THEN part>               ::= THEN <statement>
273.  <ELSE part>               ::= ELSE <statement>
274.  <branch word>             ::= CARRY|NOCARRY|OVERFLOW|
                                  NOVERFLOW|IABZ|DABZ|
                                   IXBZ|DXBZ|<relop>
275.  c5fiMOVE statement>       ::= <MOVE stmt>|
                                   <MOVE stmt>,<sdec>|
                                   <MOVE-WHILE stmt>|
                                   <MOVE-WHILE stmt>,<sdec>
276.  <MOVE stmt>               ::= MOVE <dest ref>:=<pointarr> <count>|
                                   MOVE <pointarr>:=*<count>|
                                   MOVE <pointarr>:=*PB<count>|
                                   MOVE <pointarr>:=<string>|
                                   MOVE <pointarr>:=(<listelmt>)
277.  <dest ref>                ::= <pointarr>|
                                *
278.  <MOVE-WHILE stmt>         ::= MOVE <byte ref>:=<byte ref> WHILE <ccf>
279.  <byte ref>                ::= <byte pointarr>|
                                   *
```

```
280. <pointarr>                 ::=<pointer name>|
                                   <pointer name> <index>|
                                   <array name>|
                                   <array name> <index>
281. <byte pointarr>            ::=<pointarr>
282. <count>                    ::=,(<integer expression>)
283. <PROCEDURE call stmt>      ::=<procedure id>|
                                   <procedure id> <actual param part>
284. <actual param part>        ::=(<actual param list>)|
                                   (<stacked param list>)|
                                   (<stacked param list>,<actual param list>)
285. <actual param list>        ::=<actual param>|
                                   <actual param>,<actual param list>|
                                   ,<actual param list>
286. <actual param>             ::=<reference param>|
                                   <value param>
287. <stacked param list>       ::=*|
                                   *,<stacked param list>
288. <reference param>          ::=<simple variable>|
                                   <array name>|
                                   <array name> <index>|
                                   <pointer name>|
                                   <pointer name> <index>|
                                <procedure id>|
                                <label>
289. <value param>              ::=<arithmetic expression>|
                                   <logical expression>|
                                   <assignment statement>
290. <PUSH statement>           ::=PUSH(<register spec list>)
291. <SET statement>            ::=SET(<register spec list>)
292. <register spec list>       ::=register|
                                   register,<register spec list>
293. <register>                 ::=S|Q|X|STATUS|
                                  Z|DL|DB|SBANK
294. <RETURN statement>         ::=RETURN|
                                   RETURN <pcount>
295. <pcount>|                  ::=<unsigned integer>
296. <SCAN statement>           ::=<SCAN-WHILE stmt>|
                                   <SCAN-WHILE stmt>,<sdec>|
                                   <SCAN-UNTIL stmt>|
                                   <SCAN-UNTIL stmt>,<sdec>
297. <SCAN-WHILE stmt>          ::=SCAN <byte ref> WHILE <testword>
298. <testword>                 ::=<simple variable>|
                                   <integer>|
                                   "<char> <char>"|
                                   *
299. <SCAN-UNTIL stmt>          ::=SCAN <byte ref> UNTIL <testword>
300. <char>                     ::={any member of the ASCII character set}
301. <SUBROUTINE call stmt>     ::=<subroutine name>|
                                   <subroutine name> <actual sparam part>
302. <actual sparam part>       ::=(<actual sparam list>)|
                                   (<stacked param list>)|
                                   (<stacked param list>,<actual sparam list>)
303. <actual sparam list>       ::=<actual sparam>|
                                   <actual sparam>,<actual sparam list>
304. <actual sparam>            ::=<reference sparam>|
                                   <value param>
```

```
305. <reference sparam>           ::=<simple variable>|
                                     <array name>|
                                     <array name> <index>|
                                     <pointer name>|
                                     <pointer name> <index>|
                                     <procedure name>
306. <WHILE statement>            ::=WHILE <condition clause> DO <statement>
```

INDEX

Part No. 30000-90024
Printed in U.S.A. 9/76
Update No. 1 Incorporated 12/76

HEWLETT hp PACKARD

Sales and service from 172 offices in 65 countries.
5303 Stevens Creek Blvd., Santa Clara, California 95050