HP 3000

SYSTEMS PROGRAMMING LANGUAGE

HP Computer Museum
www.hpmuseum.net

For research and education purposes only.

HP 3000

SYSTEMS PROGRAMMING LANGUAGE

03000-90002A November 1972

©Copyright, 1972, by HEWLETT-PACKARD COMPANY, 11000 Wolfe Road, Cupertino, California. All rights reserved.
Printed in the U.S.A.

List of Effective Pages

Pages	Effective Date
Title	Nov. 1972
Copyright	Nov. 1972
iii	Nov. 1972
v to xi	Nov. 1972
1-1 to 1-4	Nov. 1972
2-1 to 2-9	Nov. 1972
3-1 to 3-38	Nov. 1972
4-1 to 4-19	Nov. 1972
5-1 to 5-35	Nov. 1972
A-1 to A-3	Nov. 1972
B-1	Nov. 1972
C-1 to C-7	Nov. 1972
D-1 to D-2	Nov. 1972
E-1 to E-12	Nov. 1972
F-1 to F-3	Nov. 1972
G-1 to G-2	Nov. 1972

Printing History

Part No.	Date	Update Package	Date
03000-90002A	Nov. 1972		

PREFACE

This is the reference manual for the HP 3000 Systems Programming Language (SPL/3000). A prerequisite to this manual is previous experience with SPL/3000 gained through the *HP 3000 Systems Programming Language Textbook (03000-90003)* and/or a Hewlett-Packard training course in SPL/3000.

This book is divided into five sections:

Section I — Program Components

Section II — Constants and Identifiers

Section III — Declarations

Section IV — Expressions

Section V — Statements

Each topic in SPL/3000 is discussed in four parts:

Purpose

Syntax

Semantics

Examples

At the end of this manual are appendices listing reserved words, the ASCII character set used in SPL/3000, compiler commands, operating system commands, machine instructions, an index of all syntax rules, and instructions for building an intrinsic file.

The SPL/3000 compiler and the machine code which it generates operate within the HP 3000 Multiprogramming Executive (MPE/3000). The relevant documents are

HP 3000 Multiprogramming Executive Operating System (03000-90005)

HP 3000 Multiprogramming Executive Console Operator's Guide (03000-90006)

Information on HP 3000 hardware is contained in:

HP 3000 Reference Manual (03000-90019)

CONTENTS

PREFACE	iii
INTRODUCTION	vii
CONVENTIONS	vii
SYNTAX NOTATIONS	viii
Syntax References	xi
SECTION I PROGRAM COMPONENTS	
PROGRAM STRUCTURE	1-1
COMMENTS	1-3
DELIMITERS	1-4
SECTION II CONSTANTS AND IDENTIFIERS	
CONSTANT TYPES	2-1
INTEGER CONSTANTS	2-2
REAL CONSTANTS	2-4
LOGICAL CONSTANTS	2-6
STRING CONSTANTS	2-7
IDENTIFIERS	2-8
SECTION III DECLARATIONS	
DECLARATION TYPES	3-1
DECLARATION CONCEPTS	3-3
Data Types . .	3-3
Addressing Data	3-4
Address Allocation	3-4
Initialization	3-8
GLOBAL-EXTERNAL Attribute	3-8
Address Reference	3-8
DEFINE DECLARATION AND INVOCATION	3-11
EQUATE DECLARATION AND INVOCATION	3-12
SIMPLE VARIABLE DECLARATION	3-14
ARRAY DECLARATION	3-16
POINTER DECLARATION	3-22
LABEL DECLARATION	3-25
SWITCH DECLARATION	3-26
ENTRY DECLARATION	3-27

	Page
PROCEDURE DECLARATION	3-28
INTRINSIC DECLARATION	3-36
SUBROUTINE DECLARATION	3-37
SECTION IV EXPRESSIONS	
EXPRESSION TYPES	4-1
VARIABLES	4-3
FUNCTION DESIGNATOR	4-6
BIT OPERATIONS	4-8
ARITHMETIC EXPRESSIONS	4-11
LOGICAL EXPRESSIONS	4-14
SECTION V STATEMENTS	
STATEMENT TYPES	5-1
ASSEMBLE STATEMENT	5-2
ASSIGNMENT STATEMENT	5-6
CASE STATEMENT	5-8
DELETE STATEMENT	5-10
DO STATEMENT	5-12
FOR STATEMENT	5-13
GO STATEMENT	5-15
IF STATEMENT	5-17
MOVE STATEMENT	5-21
PROCEDURE CALL STATEMENT	5-25
PUSH AND SET STATEMENTS	5-27
RETURN STATEMENT	5-29
SCAN STATEMENT	5-30
SUBROUTINE CALL STATEMENT	5-33
WHILE STATEMENT	5-35
APPENDIX A ASCII CHARACTER SET	A-1
APPENDIX B RESERVED WORDS	B-1
APPENDIX C COMPILER COMMANDS	C-1
APPENDIX D MPE/3000 SUBSYSTEM COMMANDS	D-1
APPENDIX E HP 3000 MACHINE INSTRUCTIONS	E-1
APPENDIX F BNF SYNTAX INDEX	F-1
APPENDIX G BUILDING AN INTRINSIC FILE	G-1

FIGURE

Figure 3-1. Example Data Area 3-7

TABLE

Table 3-1. Parameters Passed to Formal Parameters 3-32

INTRODUCTION

SPL/3000 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/3000 has many features normally found only in high-level languages such as PL/1 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, however, machine-level constructs have been included to ensure that complete control of the machine is available to the programmer when he needs it. These 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.

CONVENTIONS

In the HP 3000, the bits of a word(s) are numbered from left to right starting with 0. Thus the sign bit, 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

SYNTAX NOTATIONS

The syntax of SPL/3000 is described in an expanded version of Backus Normal Form (BNF) notation. BNF is a symbolic description language that allows a very concise and precise definition of how the constructs of a language can be put together (i.e., how sentences in a language can look). BNF does not tell what a construct does (its meaning or semantics), only how it is analyzed into parts.

BNF consists of the following components:

<. . .> Left and right broken brackets are used to set off a metalinguistic variable. Each metalinguistic variable stands for one part of the language's syntax and each is defined by a metalinguistic formula. For English, <sentence>, <noun>, <verb>, and <adverb> are all metalinguistic variables.

<program> <statement>

::= The symbol double-colon-equals (::=) "is defined as" in BNF. It is always preceded by a single metalinguistic variable and followed by the formula which defines the structure of that variable from left to right.

<sentence> ::= *formula*

Formulas are composed of three things: metalinguistic variables, alternative definition indicators ("or" signs), and literal symbols (characters which stand for themselves).

The symbol | means "or." It separates multiple definitions (formulas) of a metalinguistic variable.

<sentence> ::= <simple sentence> | <compound sentence>

Each metalinguistic variable in the formula must be defined in another BNF definition (each definition is called a "production").

SYMBOLS Characters standing by themselves (not enclosed in < >) are literal characters that must appear in the program exactly as they stand in the definition. If any production is traced through completely (i.e., the productions of all variables used are also examined) the process will always come to a stop when productions are reached that consist of all literals. Literal symbols require no further definition. For example, in the formula below, BEGIN is a literal symbol:

<global head> ::= BEGIN <data group> <procedure group>

{. . .} Brackets are used to enclose an English explanation that is inserted in a production in place of a metalinguistic variable when the BNF required is either impossible or too cumbersome. For example:

<empty> ::= {a character string of zero length}

†	Comments or notes within BNF products appear indented below the variable to which they apply preceded by a † character. These comments are not metalinguistic variables; they are merely explanatory material.
list	The word "list" is used in metalinguistic variables to imply the common construct of a list of items of the same class separated by commas.

<sign> ::= + | -
† the symbol – indicates two's complement

<identifier list> ::= <identifier> | <identifier list>, <identifier>

This production says that an identifier list can consist of a number of identifiers separated by commas. Productions of this type are assumed whenever a metalinguistic variable is combined with the word "list" in a formula.

In the description of symbolic constructs it is important to indicate where different data types are allowed. Assume the following abbreviations:

i	Integer
r	Real
l	Logical
b	Byte
d	Double
e	Long (extended precision)

The symbol T with a subscript is used in the syntax to indicate allowable classes of data types:

T	Integer, logical, byte, real, long, double.
T_{ilb}	Integer, logical, byte.
T_{il}	Integer, logical.
T_{irlde}	Integer, real, logical, double, long.
T_{ire}	Integer, real, long.

The occurrence of one of these T symbols in a metalinguistic variable specifies that this symbol must be replaced consistently (throughout the production) by the appropriate set of English words. In some case this convention reduces the number of productions required dramatically. For example the syntactic rule

<T_{ilr} array identifier> ::= <identifier>

corresponds to

<integer array identifier> ::= <identifier>
<logical array identifier> ::= <identifier>
<real array identifier> ::= <identifier>

or, in another case, the notation

<T_{ilr} variable> ::= <T_{ilr} simpvar identifier> | <T_{il} pointer identifier>

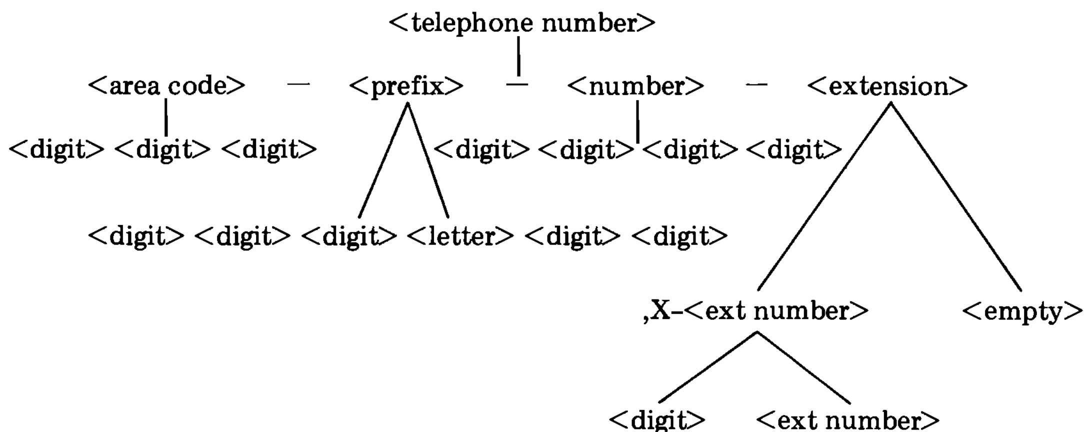
expands to

<integer variable> ::= <integer simpvar identifier> | <integer pointer identifier>
<logical variable> ::= <logical simpvar identifier> | <logical pointer identifier>
<real variable> ::= <real simpvar identifier>

The way in which BNF can be used to describe syntax is easily demonstrated by an example. Consider the structure of a telephone number:

area code — prefix — number — extension

All possible combinations of legal telephone numbers can be diagramed by this tree structure:



The definitions for <digit> and <letter> have been omitted from this diagram for clarity. Branches in the diagram represent alternative forms. The tree above says that a phone number consists of an area code, prefix, number, and extension. The prefix can be a three-digit number *or* a two-letter-digit exchange. The extension can be as long as desired or left off (<empty>). Note that extension number can be indefinitly long because "ext number" appears in the definition of "ext numbers." (Semantics may be used to restrict this indefinite length.)

The BNF description of this structure is:

<telephone number>	::=	<area code> — <prefix> — <number> <extension>
<area code>	::=	<digit> <digit> <digit>
<prefix>	::=	<digit> <digit> <digit> | <letter> <letter> <digit>
<number>	::=	<digit> <digit> <digit> <digit>
<extension>	::=	,X-<ext number> | <empty>
<ext number>	::=	<digit> | <ext number> <digit>
<digit>	::=	0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<letter>	::=	A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|R|S|T|U|V|W|X|Y
<empty>	::=	{a character string of zero length; i.e., nothing}

Syntax describes what elements must appear in what order for something to be a "legal" (qualified) telephone number. However, rules regarding *how* to combine these elements are usually omitted from the syntax for clarity and must be described in English sentences elsewhere. These omitted rules are called semantics and are as important in describing the grammar of a language as is the syntax.

For example, neither the area code nor the prefix in a telephone number can begin with the digit zero, but nowhere is this mentioned in the syntax. This restriction could be incorporated in the syntax, but such as inclusion might make the syntax clumsy and difficult to use. Instead, this restriction is easily described (along with other similar restrictions) in an English sentence (i.e., semantically).

Syntax References

The following notation is used after each block of syntax productions to provide references to metalinguistic variables which are left undefined:

<logical primary> → III, LOGICAL EXPRESSIONS

metalinguistic variable → *section, heading within section*

SECTION I
Program Components

PROGRAM STRUCTURE

The SPL/3000 compiler accepts either complete programs or subprograms. In a subprogram compilation global declarations allocate no space; code is generated for procedures only, not for a main body.

Syntax

```
<program>              ::=  <global head> <main body>.
<global head>          ::=  BEGIN <data group> <procedure group>
<main body>            ::=  <compound tail>
<compound tail>        ::=  <statement> END | <statement> ;<compound tail>
<compound statement>   ::=  BEGIN <compound tail>
<subprogram>           ::=  BEGIN <subdata group> <proc group> END.
<data group>           ::=  <data group> <data declaration> ; | <empty>
<procedure group>      ::=  <procedure group> <subroutine declaration> ;<proc group>
<proc group>           ::=  <proc group> <procedure declaration> ; |
                            <proc group> <intrinsic declaration> ; | <empty>
<subdata group>        ::=  <subdata group> <subdata declaration> ; |
                            <subdata declaration> ; | <empty>
<empty>                ::=  {a character string of zero length}
```

Syntax References

```
<data declaration>        →  III,  DECLARATION TYPES
<subroutine declaration>  →  III,  SUBROUTINE DECLARATIONS
<procedure declaration>   →  III,  PROCEDURE DECLARATIONS
<intrinsic declaration>   →  III,  INTRINSIC DECLARATIONS
<subdata declaration>     →  III,  DECLARATION TYPES
<statement>               →   V,   STATEMENT TYPES
```

Semantics

The <global head> contains all the global declarations for a main program and the <main body> contains all the statements of a main program. When procedures only are compiled (<subprogram>), global data declarations allocate no space because the procedures will be linked up with a main program (which is compiled separately) by the operating system. See Appendix C, "Compiler Commands," for details on subprogram compilation.

BEGIN and END are used as a delimiting pair and are matched, much like parentheses. Any compilation is bracketed by a BEGIN and an END. Within the body of a main program or a procedure, a BEGIN-END pair can be used to combine several statements into one. A program is terminated by an END. or when all BEGINs are matched.

EXAMPLES:

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

<<compound statement>>
     BEGIN
          A := B;
          B := D;
          E := F;
     END;
     BEGIN
          A := B;
          IF A > 10 THEN
               BEGIN
                    C := A+20;
                    D := E-F;
                    G := A*F
               END;
     END;

<<subprogram>>
     BEGIN
          INTEGER N,M,O; <<allocate no space>>
          EQUATE A := 101, B := 202; <<subdata group>>
          PROCEDURE C;
               BEGIN
                    .
                    .
                    .
                    .
                    .
               END;
```

```
PROCEDURE D;
     BEGIN
        .
        .
        .
     END;
PROCEDURE E;
     BEGIN
        .
        .
        .
<<main body, if any, is ignored>>
END
```

COMMENTS

A comment is used to document a program but has no effect upon the functioning of the program itself (i.e., a comment generates no code). In SPL/3000 comments of two types can be inserted wherever needed.

Syntax

<comment> ::= COMMENT {any sequence of ASCII characters except semicolon}; |
<< {any sequence of ASCII characters excluding>> }>>

Syntax Reference

ASCII characters → Appendix, C

Semantics

The first form of comment (COMMENT . . .;) is equivalent to a null statement and can be used anywhere a statement (or declaration) would be expected. The second form of comment (<<. . . .>>) can be used anywhere in a program, except within an identifier.

The characters within a comment are ignored by the compiler (they are *not* upshifted if lowercase).

EXAMPLES:

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

DELIMITERS

Blanks are always recognized as delimiters in SPL/3000, except within character strings. Special characters can also act as delimiters:

Semicolon (;)
Parentheses (())
Operators (+, –, *, /, ^)
Brackets ([ ])

SECTION II
Constants and Identifiers

CONSTANT TYPES

Constants are literal values that stand for themselves.

There are two basic types of constants in SPL/3000: numerical constants and string constants.

Syntax

```
<constant>    ::= <number> | <string>
<number>      ::= <integer> |
                       † 16 bits
                  <real number> |
                       † 32 bit floating point
                  <double integer> |
                       † 32 bit integer
                  <long real number> |
                       † 48 bit floating point
                  <logical value>
                       † 16 bits
```

Semantics

In SPL/3000 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 except for long (extended precision) real numbers; operations on long data use library procedures.

INTEGER CONSTANTS

There are three representations for integer constants: decimal integers, based integers, and composite integers. Any integer can be specified as type double by following it with a D.

Syntax

```
<integer>             ::=  <unsigned integer> |
                                 † unsigned assumed positive
                           <sign> <unsigned integer>

<decimal integer>     ::=  <digit> | <decimal integer> <digit>

<unsigned integer>    ::=  <decimal integer> | <based integer> | <composite integer> |
                           <equate invocation>

<digit>               ::=  0|1|2|3|4|5|6|7|8|9

<sign>                ::=  +|-
                                 †-is two's complement

<based integer>       ::=  %<base part> <base digit> | <based integer> <base digit>

<base part>           ::=  (<base>) | <empty>
                                 †<empty> means octal

<base>                ::=  {any number of the set (2,3,4,5,6,7,8,9,10,11,12,13,14,15,16)}

<empty>               ::=  {a character string of zero length}

<base digit>          ::=  {any of the digits 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)}

<composite integer>   ::=  [<integer field list>]

<integer field>       ::=  <number of bits> /<decimal integer> |
                           <number of bits> /<based integer> |
                           <number of bits> / <composite integer>

<number of bits>      ::=  <decimal integer>

<double integer>      ::=  <integer> D
                                 † up to 32 bits
```

Syntax References

```
<equate invocation>  →  III,  EQUATE DECLARATION AND INVOCATION
```

Semantics

Decimal integers are the simplest form of integer; they consist of a sequence of decimal digits. Octal (base eight) integers are next in complexity; they consist of a sequence of octal digits (0 to 7) preceded by a percent sign (%). Based integers consist of a base and a sequence of digits legal in that base. All bases from 2 through 16 are allowed (the digits A, B, C, D, E, and F stand for 10, 11, 12, 13, 14, and 15).

Composite integers are formed through left-to-right concatenation of binary bit fields. In 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.

The two's complement of any integer can be formed by preceding it with a minus sign (-).

Integers without a D (double) must be capable of fitting into 16 bits ($-32{,}768_{10}$ to $+32{,}767_{10}$). Double integers can range from -2147483648_{10} to $+2147483647_{10}$.

Care should be exercised when using blank as a delimiter in the specification of based integers. The based integer

```
%(16)ABCD
```

is not equivalent in size or value to the based integer

```
%(16)ABC D.
```

The blank inserted in the second case makes it type double.

EXAMPLES:

```
<<decimal integer>>
      123456
      34
      57
      999

<<based integer>>
      %1777 <<octal>>
      %(2)101110111
      %(11)1092A
      %(16)A012

<<composite integers>>
      [3/2, 12/%5252] <<equals %52524>>
      [2/211, 15/[3/%(2)101, 12/0], 10/123] D
            <<equals %720000173>>

<<integer>>
      -12345
      -%(2)1110010
      -[3/2, 12/%5252]
```

REAL CONSTANTS

Real numbers can be either standard (32 bits) or long (48 bits). In both cases they consist of a signed magnitude (e.g., -1056) and a signed decimal power (e.g., E20, L-15). Standard real numbers are specified by a decimal point or an E before the exponent; long real numbers must always contain an L to distinguish them from standard real numbers.

Syntax

<real number> ::= <unsigned real number> | <sign> <unsigned real number>

<unsigned real number> ::= <fraction> | <decimal integer> E <power> |
<fraction> E <power> | <composite integer> E |
<based integer> E

<fraction> ::= <decimal integer> . | . <digit> | <fraction> <digit>

<power> ::= <decimal integer> | <sign> <decimal integer>

<long real number> ::= <unsigned long real number> |
<sign> <unsigned long real number>

<unsigned long real number> ::= <decimal integer> L <power> | <fraction> L <power> |
<composite integer> L | <based integer> L

<sign> ::= + | -

Syntax References

<decimal integer> → II, INTEGER CONSTANTS

<composite integer> → II, INTEGER CONSTANTS

<based integer> → II, INTEGER CONSTANTS

<digit> → II, INTEGER CONSTANTS

Semantics

As shown in the syntax, the magnitude part of a real number can be a decimal fraction, a decimal integer, a composite integer, or a based integer and the power must be a decimal integer (with or without sign). Real and long real numbers are accurate to 6.9 decimal digits of magnitude and 11.7 digits respectively (0 can be represented exactly). The absolute value can range from 8.6366×10^{-78} to 1.1579×10^{77}.

When a composite or based integer is used, <power> does not follow the L or E. The bit pattern created for the integer is used directly as 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.

EXAMPLES:

<<real number>>

+1.324
-.1024
-1.105E-21
%(4)321000E
-%(2)111101111011E
[3/5, 5/273, 20/%(16)102AB39] E

<<long real number>>

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

LOGICAL CONSTANTS

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

Syntax

<logical value> ::= TRUE | FALSE | <integer>

Syntax References

<integer> → II, INTEGER CONSTANTS

Semantics

A logical value is considered true if its value is odd, false if its value is even (i.e., only the last bit 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_{10} to $+65{,}535_{10}$. When negative integers are used as logical values they are interpreted as large positive numbers (e.g., -1 equals %177777).

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 can be packed two per word.

Syntax

<string> ::= "<character string>"

<character string> ::= <character> | <character string> <character>

<character> ::= {a member of the set of ASCII character representations}

Syntax References

ASCII characters → Appendix, A

Semantics

A character string can contain from 1 to 64 ASCII characters. A quote (") is represented within a character string by a pair of quotes (" ") to avoid ambiquity with the string terminator.

EXAMPLE:

<<string>>

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

IDENTIFIERS

Identifiers are symbols used to name data and code constructs in an SPL/3000 program. They consist of uppercase letters and numbers, and are assigned uses by declarations. There is no implicit typing assumed for identifiers.

Syntax

```
<identifier>               ::= <letter> | <identifier> <letter> |
                               <identifier> <digit> | <identifier>'
<T simpvar identifier>     ::= <T identifier>
<T identifier>             ::= <identifier>
<T pointer identifier>     ::= <identifier>
<T array identifier>       ::= <identifier>
<procedure identifier>     ::= <T proc identifier> | <proc identifier>
<T proc identifier>        ::= <identifier>
<proc identifier>          ::= <identifier>
<entry identifier>         ::= <identifier>
<label identifier>         ::= <identifier>
<switch identifier>        ::= <identifier>
<equate identifier>        ::= <identifier>
<define identifier>        ::= <identifier>
<subroutine identifier>    ::= <T subr identifier> | <subr identifier>
<T subr identifier>        ::= <identifier>
<subr identifier>          ::= <identifier>
<intrinsic identifier>     ::= <identifier>
<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
<digit>                    ::= 0|1|2|3|4|5|6|7|8|9
```

Semantics

An identifier always starts with a letter and may contain from 1 to 15 contiguous 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). The attributes of an identifier are determined by a declaration, *not* by the form of the identifier (the syntax shows that all types of syntax identifiers have the same form).

Reserved words are combinations of characters that cannot be used as identifiers, since they are used with implied meanings in the language. (See Appendix B for a list of SPL/3000 reserved words.)

In the syntax, T is used as an abbreviation to indicate the following class of data types:

integer, logical, byte, real, long, double.

EXAMPLE:

```
<<identifier>>
    MATRIX
    A''''B
    AN'IDENTIFIER
    MAT1
    X
```

SECTION III
Declarations

DECLARATION TYPES

A declaration defines the attributes of an identifier before it is used in a program or procedure. All identifiers in SPL/3000 (with the exception of labels) must be explicitly declared once, and only once, within a single program or procedure. There are two possible levels of declarations in SPL/3000: global, for a main program and local, for a procedure. Global declarations can be accessed throughout a program (even within procedures) and are grouped together at the beginning of the program. Local declarations can be accessed only within the procedure where declared and are grouped together at the beginning of the procedure body.

Syntax

```
<data group>         ::= <data group> <data declaration> ; | <empty>
<data declaration>   ::= <define declaration> |
                               † any order
                         <equate declaration> |
                         <global simpvar declaration> |
                         <global array declaration> |
                         <global pointer declaration> |
                         <label declaration> |
                         <switch declaration> |
                         <entry declaration>
<procedure group>    ::= <procedure group> <subroutine declaration> ; |
                         <proc group>
                               † subroutines last
<proc group>         ::= <proc group> <procedure declaration> ; |
                         <proc group> <intrinsic declaration> ; |
                         <empty>
                               † any order
<subdata group>      ::= <subdata group> <subdata declaration> ; |
                         <subdata declaration> ; | <empty>
                               † subprogram compilations
```

<subdata declaration> ::= <define declaration> |
<equate declaration> |
<global simpvar declaration> |
<global array declaration> |
<global pointer declaration> |
<simpvar declaration> |
<pointer declaration> |
<array declaration>

† subdata declarations do not allocate global space

Syntax References

These declaration types are described in this section.

<define declaration>

<equate declaration>

<simpvar declaration>

<array declaration>

<pointer declaration>

<label declaration>

<switch declaration>

<entry declaration>

<procedure declaration>

<intrinsic declaration>

<subroutine declaration>

Semantics

As stated in Section I, "Program Structure," the SPL/3000 compiler accepts either main program or procedure-only compilations. When procedures are compiled alone they must be linked up with some main program before they can be executed.

All data declarations (e.g., the data group) must occur before the procedure group. Within the data group, data declarations can occur in any order. But, in procedure-only compilations, any data declarations which imply storage locations must use address reference (see "Declaration Concepts"). New global locations are never directly allocated in procedure-only compilations.

After data declarations come procedures and intrinsic declarations, intermixed in any order. The global subroutine declarations occur last, but are only allowed in main program compilations.

Declarations can also occur within procedures. The declarations which are allowed locally are described under "Procedure Declaration."

DECLARATION CONCEPTS

Certain concepts are common to many declarations:

Data Types
Addressing Data
Address Allocation
Initialization
GLOBAL-EXTERNAL Attribute
Address Reference

Data Types

A data type specification in a declaration defines the set of instructions which can operate on the item declared and the amount of storage its value will occupy. Six reserved words are used for type assignment:

INTEGER	Single precision positive and negative integer values, including zero. 16-bit, two's complement representation which can range from -32,768 to +32,767. Hardware provides addition, subtraction, multiplication, division, modulo, negation, and comparison. Software provides exponentiation.
REAL	Single precision positive and negative floating-point values, including zero. 32-bit sign plus magnitude representation with an absolute value that can range from 8.6366 x 10^{-78} to 1.1579 x 10^{+77} (zero can be represented exactly as a special case). Real numbers are accurate to 6.9 decimal digits of magnitude. Hardware provides addition, subtraction, multiplication, division, negation, and comparison. Software provides exponentiation.
DOUBLE	Double precision positive and negative integral values, including zero. 32-bit, two's complement representation which can range from -2,147,483,648 to +2,147,483,647. Hardware provides only addition, subtraction, comparison, and negation.
LONG	Extended precision positive and negative floating-point values, including zero. 48-bit sign plus magnitude representation with the same range as real, but with 11.7 decimal digits of accuracy. Hardware provides no operations; software provides addition, subtraction, division, multiplication, negation, comparison and exponentiation.
LOGICAL	Single precision positive integer values, including zero. 16-bit, positive binary representation which can range from 0 to +65,535. Hardware provides addition, subtraction, multiplication, division, modulo, or, and, exclusive or, comparison, and complement.

BYTE — Half precision positive integer values, including zero. Eight-bit integral representation which can range from 0 to 255. When type byte is used in arithmetic operations, 16-bit integer arithmetic is performed on the top of the stack. If the result is stored in a *byte*, it is always truncated to an 8-bit integer (which must therefore be interpreted as positive, even if the 16-bit result was negative) and is stored in the left half unless indexed. Hardware provides moves, scans, and compares for strings of bytes.

Addressing Data

Identifiers can be declared globally (e.g., in a main program) or locally (e.g., in a procedure). Global identifiers are recognized throughout the main program and all procedures (except that subroutines are recognized only in the main program). Local identifiers are recognized only within the procedure where they are declared. When a local identifier equals a global identifier, only the local identifier is recognized within the procedure.

Unless the programmer specifies otherwise, global data items are assigned DB-relative locations and local data items are assigned Q-relative locations. The programmer can override this assignment by asking that a global identifier be allocated Q relative or a local identifier DB-relative.

Data assigned DB-relative addresses or the index register can be addressed throughout the scope of the entire program. That is, the locations can be addressed in all cases, although the identifiers may not be recognized in some contexts. Data assigned Q-relative addresses can be addressed only within the scope of the Q-register setting corresponding to where they are declared. Data assigned S-relative addresses can be addressed consistently (i.e., the compiler corrects for changes in S) within a statement.

The address field of the memory reference instructions (bits 6 through 15) determines the range (in words) of each addressing mode:

	Bits 6	7	8	9	10	11	12	13	14	15	Range
P + relative	0	0	Displacement →								8 bits (0:255)
P - relative	0	1	Displacement →								8 bits (0:255)
DB + relative	1	0	Displacement →								8 bits (0:255)
Q + relative	1	1	0	Displacement →							7 bits (0:127)
Q - relative	1	1	1	0	Displacement →						6 bits (0:63)
S - relative	1	1	1	1	Displacement →						6 bits (0:63)

Address Allocation

When data identifiers are declared they are allocated addresses in the data stack. Normally, global identifiers are allocated the next available DB-relative locations and local identifiers are allocated the next available Q-relative locations. The number of locations allocated depends on the item being declared and its data type. It is also possible to assign data identifiers to specific relative locations (see "Address Reference"), thus overriding the normal default assignment.

SIMPLE VARIABLES

A simple variable is allocated the next available locations relative to DB or Q. The number of locations allocated is

- One for integer, logical, or byte (actual byte data occupies bits 0 through 7 of word allocated).
- Two for double or real.
- Three for long.

For example, if the next available DB location is DB+7 and three global variables are declared (integer I, real R, and byte B), the following locations are allocated:

$DB+7_8$	integer I
$DB+10_8$	real R
$DB+11_8$	
$DB+12_8$	byte B
$DB+13_8$	next available location

POINTERS

Pointers are always allocated only one location, regardless of the data type. Thus if three local pointers (P1, P2, P3) are declared and the next available Q-relative location in that procedure is Q+3, then:

Q+3	pointer P1
Q+4	pointer P2
Q+5	pointer P3
Q+6	next available location

INDIRECT ARRAYS

Normally, arrays are addressed indirectly through a data label (address). The data label is allocated the next available DB or Q location and the actual array storage is allocated either in the secondary DB (beyond the global variables) or on the top of the stack (beyond the local variables).

For dynamic local arrays, the data label is allocated the next available Q-relative location. When the procedure is entered, the upper and lower bounds are computed and the storage for the array is allocated on the top of the stack.

The quantity of space allocated for an array of N elements is dependent on its data type:

- N words for an integer of logical array.
- 2 times N words for a double or real array.
- 3 times N words for a long array.
- $(N + 1)/2$ words for a byte array.

DIRECT ARRAYS

SPL/3000 allows the programmer to specify that an array will be accessed directly (without going indirectly through a pointer). The total storage for a direct array is allocated starting with the next available DB or Q cell. There are certain restrictions on direct arrays based on the limited range of directly addressed locations.

EXAMPLES:

Assume that the following declarations have been made:

Main Program:

An indirect integer array, A, ranging from 0 to 72.

An integer simple variable, I.

A double pointer, P.

A direct logical array, AA, ranging from 0 to 9.

Procedure PROC:

A local integer simple variable, X.

A local integer simple variable, Y.

An indirect local logical array, B, ranging from 0 to 7.

When procedure PROC is called and has set up its local storage, the data area appears as follows:

Address	Low Memory	Description
DB+0		data label for array A
+1		cell for integer I
+2		cell for pointer P
+3		10 cells for direct array AA
⋮	⋮	
+12		Primary DB
+13		Secondary DB
⋮	⋮	Storage for indirect array A
+85		
	⋮	
Q+0		Q register setting for current procedure (PROC)
+1		cell allocated for integer X
+2		cell allocated for integer Y
+3	Q - DB + 4	data label for array B
+4	⋮	8 cells to array B
Q+11		
S		top of stack
		undefined area
Z		
	High Memory	

Figure 3-1. Example Data Area

Initialization

Compile-time initialization of data items is allowed in SPL/3000 wherever possible.

Simple variables are initialized when declared by following the identifier with a := and a constant of appropriate type. When execution of the program begins the variable has the initial value specified.

Pointers can be initialized with the addresses of other data items when they are declared.

The only arrays that can be initialized are global arrays, local OWN arrays, and local PB-relative arrays (all with defined bounds). The initialization consists of a list of constants separated by commas; repeat factors are allowed before lists of constants in parentheses to indicate that the list is to be repeated. Only the last array specified in a single array declaration list can be initialized.

Generally the constants used in initialization must match the data type of the variable being declared exactly. However, strings can be used to initialize any data type; bytes are used consecutively from the left. A remaining right byte, if any, is filled with a blank. In addition, variables declared type byte can be initialized with integer constants that fit in 8 bits (the constant is truncated on the left if too large and a warning is issued).

Local variables with initialization, except those declared OWN, are initialized every time the procedure is entered. Since OWN variables are allocated in the DB area they are initialized only once, before the start of the program.

GLOBAL-EXTERNAL Attribute

When a procedure is compiled separately, it is necessary to take special steps to establish a DB-relative variable that can be recognized in both the procedure and the main program to which it will eventually be linked by the operating system. The GLOBAL-EXTERNAL mechanism is provided for this purpose.

When a local variable is declared EXTERNAL, the variable must be linked with a variable of the same identifier and type declared GLOBAL in a main program. The GLOBAL-EXTERNAL variable is allocated to specific DB-relative location when the operating system links the procedure with the main program. This location can then be accessed by both the procedure and the program using the same identifier.

Address Reference

Address reference allows variables to be equivalenced to locations relative to other variables or the address registers. Generally, no storage is allocated when address reference is specified and these variables can never be initialized. There are three forms of address reference: variable reference, base register reference, and indexed identifier reference.

VARIABLE REFERENCE

The data item is assigned the location and addressing convention of a referenced identifier, adjusted by a plus or minus number of words. The resulting address must be within the direct address ranges (see "Addressing Data," this section).

<identifier> = <identifier> |
<identifier> <sign> <unsigned integer>

Assume the variable A has been assigned the location DB+5. If an integer simple variable **B** is declared with variable reference to a (B=A), then B is assigned location DB+5. If a pointer P is declared with variable reference to B+2 (P=B+2), then P is assigned DB+7. Variable reference is allowed with simple variables, pointers, and arrays (undefined bounds only).

BASE REGISTER REFERENCE

The data item is assigned an address relative to one of the data register or equal to the index register.

<identifier>	=	DB	+	<usi255> |	† unsigned integer 255
		Q	+	<usi127> |	† unsigned integer 127
		Q	–	<usi63> |	† unsigned integer 63
		S	—	<usi63> |	† unsigned integer 63
		X			† Index Register

Only simple variables of type integer, logical, or byte can be equivalenced to the index register. All variables thus equivalenced refer to one value, the current value of the index register. Since the index register is used in array indexing and other constructs, this value can change without explicit reference to an identifier equivalenced to the index register. The compiler does not save the value of any variables referenced to the index register; the programmer must maintain the integrity of these variables.

Simple variables and pointers of all types can be referenced to base register relative locations, but initialization is not allowed. If arrays are to be referenced to registers, they must have undefined bounds. If the undefined bounds specify an *, the referenced location is treated as the zero element of a direct array. If the undefined bounds specifier is an @, the referenced location is treated as a data label for an indirect reference to the zero element of an indirect array.

INDEXED IDENTIFIER REFERENCE

An array with undefined bounds (*) can be equivalenced to the location of a previously declared array or pointer.

<identifier> = <array identifier> <index> |
<pointer identifier> <index>
† <index> is optional; no <index> specifies the zero element

The referenced item specifies the zero element of the new array. If the reference is to the zero element and the referenced item is of compatible data label type (word or byte), then the data label location of the reference item is used as the data label of the new identifier. If the reference is to a direct array, the new array is direct also and no data label need be allocated.

However, if the reference is to other than the zero element, or the two items have different data label types (word versus byte), a location must be allocated for a new data label. These are the only cases in address referencing when storage is ever allocated.

For further details and examples, see "Array Declaration."

DEFINE DECLARATION AND INVOCATION

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.

Syntax

<define declaration> ::= DEFINE <definition list>

<definition> ::= <define identifier> = <text> #

<define identifier> ::= <identifier>

<text> ::= {any sequence of ASCII characters not including # except in a string}
† symbols should make sense when inserted where the define is invoked

<define invocation> ::= <define identifier>
† anywhere except within an identifier, string, or comment

Semantics

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

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

EXAMPLES:

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;>>

EQUATE DECLARATION AND INVOCATION

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 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.

Syntax

```
<equate declaration>  ::=  EQUATE <equate list>

<equate>              ::=  <equate identifier> = <equate expression>

<equate identifier>   ::=  <identifier>

<equate expression>   ::=  <sign> <equate term> |
                                 † lowest precedence
                           <equate term> |
                           <equate expression> <addop> <equate term>

<equate term>         ::=  <equate term> <muldiv> <equate primary> | <equate primary>

<equate primary>      ::=  <unsigned integer> |
                                 † highest precedence
                           <equate identifier> |
                           (<equate expression>)

<sign>                ::=  + | -

<addop>               ::=  + | -

<muldiv>              ::=  * | /

<equate invocation>   ::=  <equate identifier>
                                 † anywhere that an integer constant is allowed
```

Syntax References

<unsigned integer> → II, INTEGER CONSTANTS

Semantics

The value to be assigned to an equate identifier is determined by an equate expression. Equate expressions consist of operators (*, /, +, -), unsigned integers, previously-defined equates, 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. 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. Equate declarations are allowed globally and locally.

EXAMPLE:

```
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>>
```

SIMPLE VARIABLE DECLARATION

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

Syntax

```
<global simpvar declaration>  ::=  <global attribute> <type> <var dec list>

<global attribute>            ::=  GLOBAL | <empty>
                                          † linkage for external procedures

<local simpvar declaration>   ::=  OWN <type> <nonref var dec list> |
                                   EXTERNAL <type> <T simpvar identifier list> |
                                          † linkage to main programs compiled separately
                                   <type> <var dec list>

<type>                        ::=  INTEGER | LOGICAL | BYTE |
                                   DOUBLE | REAL | LONG

<var dec>                     ::=  <T simpvar identifier> <var reference> |
                                   <T simpvar identifier> = X |
                                          † equivalenced to the index register; integer, logical,
                                            byte only
                                   <T simpvar identifier> <simpvar init>

<nonref var dec>              ::=  <T simpvar identifier> <simpvar init>

<T simpvar identifier>        ::=  <T identifier>

<T identifier>                ::=  <identifier>

<var reference>               ::=  <empty> |
                                          † allocated next DB or Q cell
                                   = <T data identifier> |
                                          † address reference
                                   = <T data identifier> <sign> <usi> |
                                   <base register reference>

<T data identifier>           ::=  <T simpvar identifier> |
                                   <T array identifier> |
                                   <T pointer identifier>

<base register reference>     ::=  DB + <usi> |
                                          † 0 to 255
                                   Q + <usi> |
                                          † 0 to 127
                                   Q - <usi> |
                                          † 0 to 63
                                   S - <usi>
                                          † 0 to 63

<usi>                         ::=  <unsigned integer>

<simpvar init>                ::=  <empty> |
                                          † not initialized
                                   ::= <initial value>

<initial value>               ::=  <constant>
                                          † truncated on left if too large
```

Syntax References

<array> → III, ARRAY DECLARATION

<pointer> → III, POINTER DECLARATION

<constant> → II, CONSTANT TYPES

Semantics

All of the topics discussed under "Declaration Concepts" apply to some extent in the declarations of simple variables; refer to the appropriate portions of that discussion for general information on data type, address assignment, initialization, GLOBAL-EXTERNAL, variable reference, and register reference.

Simple variables can be declared globally or locally. An OWN option is available for local variables. The OWN option specifies that, although the identifier is to be recognized only locally, the storage for the variable is to be allocated in the primary DB area (the global area). This storage provides the procedure with a local variable which is not deleted when the procedure exits.

Simple variables which are address referenced to arrays are assigned either the data label location of the array (if indirect); or the zero element of the array adjusted by the word offset specified (if direct). In any case, the final address must be within the direct address range.

EXAMPLE:

```
BEGIN <<global declarations>>
      INTEGER I,J := 1245;
      DOUBLE IL := -1234579D;
      REAL A,B,C := 1.321 E-21,
            Z = DB+3;
      LOGICAL INDX = X, LI = I, JI = J;
      GLOBAL BYTE B1 := "$";
PROCEDURE PROC; <<local declarations>>
      BEGIN
            INTEGER I; <<overrides global I>>
            OWN REAL R;
            LOGICAL QP = Q-3;
            INTEGER TS = S-0;
            LONG LN := 124.0 L-25;
            LOGICAL L9 := TRUE;
            .
            .
            .
      END;
      .
      .
      .
END.
```

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/3000 provides one-dimensional arrays only — one subscript or index, not a pair or more). 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).

Syntax

```
<global array declaration>  ::=  <atype> ARRAY <g-array dec list> |
                                 GLOBAL <atype> ARRAY <G-dec list>
                                      † linkage to external procedures

<local array declaration>   ::=  <atype> ARRAY <l-array dec list> |
                                 EXTERNAL <atype> ARRAY <E-dec list> |
                                 OWN <atype> ARRAY <own array dec list>

<atype>                     ::=  <type> |
                                 <empty>
                                      † LOGICAL assumed

<type>                      ::=  INTEGER | LOGICAL | BYTE
                                 DOUBLE | REAL | LONG

<G-dec>                     ::=  <T array identifier> (<db>) = DB <array init> |
                                      † direct array, defined bounds, can be initialized
                                 <T array identifier> (*) = DB |
                                      † direct array, no bounds, user next cell as zero element
                                 <T array identifier> (@) = DB |
                                      † indirect array, no bounds, uses next cell as pointer to
                                      zero element
                                 <T array identifier> (<db>) <array init>
                                      † indirect array, defined bounds

<g-array dec>               ::=  <G-dec> |
                                 <T array identifier> (@) <indirect base register reference> |
                                      † indirect
                                 <T array identifier> (<udb>) <reference part>
                                      † direct or equivalenced

<T array identifier>        ::=  <identifier>

<db>                        ::=  <integer> : <integer>
                                      † defined bounds
```

<udb> ::= *

† undefined bounds

<E-dec> ::= <T array identifier> (<udb>)

† EXTERNAL linkage to main program compiled separately

<l-array dec> ::= <T array identifier> (<db>) = Q <array init> |

† direct array, defined bounds, can be initialized

<T array identifier> (*) = Q |

† direct array, no bounds, uses next cell as zero element

<T array identifier> (@) = Q |

† indirect array, no bounds, uses next cell as pointer to zero element

<T array identifier> (<db>) |

† indirect array

<T array identifier> (<db>) = PB := <listelmt> |

† P-relative array of constants

<T array identifier> (<vb>) |

† dynamic array

<T array identifier> (<udb>) <reference part> |

† equivalenced array or direct (base register reference)

<T array identifier> (@) <indirect base register reference>

† indirect array

<own array dec> ::= <T array identifier> (<db>) <array init>

<vb> ::= <T_{ilb} simpvar identifier> : <T_{ilb} simpvar identifier>

† variable bounds

<reference part> ::= <var reference> |

= <indexed ident reference>

<indexed ident reference> ::= <T array identifier> |

† zero element

<T pointer identifier> |

† zero element

<T array identifier> (<integer>) |

<T pointer identifier> (<integer>)

<array init> ::= <listelmt> |

<empty>

<listelmt> ::= <initial value> |

<decimal integer> (<initial value list>) |

† repeat factor allowed

<listelmt list>

† one level deep only

† no nesting

<initial value> ::= <constant>

† truncated on left if too large; assigned one constant per element except that strings are used completely from left to right

<var reference> ::= <empty> |

† allocates next DB or Q cell as pointer cell; ARRAY A (*) is equivalent to ARRAY A (@)

= <T data identifier> |

† address reference

= <T data identifier> <sign> <usi>

<base register reference>

```
<T data identifier>          ::=  <T simpvar identifier> |
                                  <T array identifier> |
                                  <T pointer identifier>

<sign>                       ::=  +|-

<indirect base register
     reference>              ::=  <empty> |
                                       † allocates next DB or Q cell as pointer to zero element
                                  = <base register reference>

<base register reference>    ::=  DB + <usi> |
                                       † 0 to 255
                                  Q + <usi> |
                                       † 0 to 127
                                  Q - <usi> |
                                       † 0 to 63
                                  S - <usi> |
                                       † 0 to 63

<usi>                        ::=  <unsigned integer>

<initial value>              ::=  <constant>
```

Syntax References

<simpvar> → III, SIMPLE VARIABLE DECLARATION

<pointer> → III, POINTER DECLARATION

<constants> → II, CONSTANT TYPES

<unsigned integer> → II, INTEGER CONSTANTS

Semantics

All of the topics discussed under "Declaration Concepts" apply to some extent in the declaration of arrays; refer to the appropriate portions of that discussion for general information on data types, address assignment, initialization, GLOBAL-EXTERNAL, variable reference, and indexed identifier reference.

Arrays can be declared globally or locally. Only global arrays can be declared with attribute GLOBAL or with directly addressed storage in the DB area. Only local arrays can be declared with attribute EXTERNAL, with variable bounds, with direct addressing in the Q area, PB relative, or OWN. Run-time bounds checking is not provided in SPL/3000.

Local OWN arrays are allocated space in the DB area and thus are not deleted when a procedure exists. A Q-relative data label pointing to the array is established each time the procedure is entered. An OWN array can be accessed only by the procedure where it is declared or by procedures to which the declaring procedure has passed it as a parameter.

When arrays are accessed, an index is specified which determines the element desired. The index register is used in accessing index arrays and pointers. See "Variables" in Section IV.

BOUNDS SPECIFICATION

Bounds specification can be defined, undefined, or variable (local only). Defined bounds consist of a pair of integer constants; the first integer is the lower bound (lowest element subscript) and the second is the upper bound (highest element subscript). The lower must be less than or equal to the upper and the number of elements in the array is determined by subtracting lower from upper and adding 1. Although all array references are relative to the zero element, zero need not be included in the defined bounds of the array.

EXAMPLE:

```
INTEGER ARRAY DONE (-100 : 100),
              START (-100 : -50);
REAL ARRAY    RVAL (0 : 20);
DOUBLE ARRAY  DVAL (20 : 200);
```

Undefined bounds are specified by an * if a direct or equivalenced array is desired, or by a @ if an indirect array is desired. Arrays with undefined bounds are not allocated any space; they must reuse storage allocated to other variables. An exception is when an array with undefined bounds is not equivalenced (<empty> reference), this case always allocates the next location as an indirect cell, whether the bounds specifier is @ or *.

EXAMPLE:

```
INTEGER ARRAY UNDEFINED (*) = DONE,
              INDIRECT (@) = START;
```

Variable bounds are allowed in local arrays. The bounds are specified by a pair of global simple variables or simple variable parameters of type integer, logical, or byte. The first must be less than the second each time they are evaluated. These bounds are calculated once each time the array is called.

EXAMPLE:

```
PROCEDURE PROC (A,B); INTEGER A,B;
    BEGIN ARRAY LOCVALUE (A:B);
        .
        .
        .
        .
    END;
```

DATA TYPES

Arrays of all data types are allowed in SPL/3000. If a data type is not specified, type LOGICAL is assumed.

STORAGE ALLOCATION

Only arrays with defined bounds are allocated space for elements. Global arrays are allocated space relative to DB and local arrays relative to Q. Local OWN arrays are allocated space relative to DB and arrays specified P relative (=PB) are allocated space in the code segment (these arrays cannot be modified during execution and must be fully initialized at compile-time). Local arrays with variable bounds are allocated space on the top of the stack each time the procedure is entered.

Data label locations are normally allocated only for indirect arrays with defined bounds. However, if an indirect array is declared with undefined bounds and no address reference (ARRAY L(@) or ARRAY L(*)), then the next available DB or Q location is allocated to it as a data label which can be filled with an address by the program during execution. Also, if an array is referenced to an indexed identifier a new data label location is required if the two items do not match in data label type or if the reference is to other than the zero element.

EXAMPLE:

```
REAL ARRAY R(0 : 10);
REAL ARRAY R2(@); REAL ARRAY R1(*);
INTEGER ARRAY I (*) = R(5);
BYTE ARRAY B(*) = R;
```

INITIALIZATION

Only arrays with defined bounds can be initialized and local PB arrays *must* be fully 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 decimal repetition factor (N) to specify that the constants in parentheses are to be used N times in initializing the array before going onto 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. There is no default value for uninitialized variables.

EXAMPLE:

```
INTEGER ARRAY N(1 : 10) := 10,9,8,7,6,5,4,3,2,1;
LOGICAL ARRAY M(0 : 5)  := "ABCDEFGHIJKL";
REAL ARRAY TEST(10:20):= 0.0, 10(1.0);
```

DIRECT/INDIRECT

Arrays can be direct or indirect. A direct array is always accessed without going through a data label; therefore, its zero element must be within the direct address range. Indirect arrays are accessed by refering indirectly through a data label; the data label must be in a directly addressed location, but the storage for the elements can be elsewhere (secondary DB or top of the stack).

Indirect arrays are the default type for arrays with defined bounds such as

```
INTEGER ARRAY A(0 : 20);
```

If this array is to be direct, the programmer must specify =DB (global only) or =Q (local only) after the bounds:

```
INTEGER ARRAY A(0 : 20) = DB;
PROCEDURE PROC:
     BEGIN
     INTEGER ARRAY A(0 : 20) = Q;
     END;
```

If the bounds specifier is * (ARRAY L(*) = DB or ARRAY L(*) = Q, the zero element of the array is the next available location, but this location is not allocated. If the bounds specifier is @, the array is indirect and the next location is used as the pointer to the zero element of the array, but his location is not allocated.

For direct arrays with defined bounds, the zero element (if there is one) must fall within the directly addressed range, assuming that the array storage is allocated starting with the next available location.

With undefined bounds the direction must be specified explicitly. An @ bounds specifier indicates that an indirect array is desired. If there is no reference part, the compiler allocates the next available location as the data label for the array.

```
INTEGER ARRAY NUMB (@); <<ALLOCATES NEXT CELL>>
ARRAY TRUTH (@) = DB+7;
```

An * bounds specifier indicates that a direct or equivalenced array is desired. If there is no reference part, the next available location is allocated as an indirect cell just as @ does. If register reference is used, the referenced address is the zero element of the direct array. If variable reference or indexed identifier reference is used, the array takes its direction from the referenced item which is also the zero element. If the referenced item is a direct array or variable the array is direct. If the referenced item is an indirect array or a pointer the array is indirect.

```
INTEGER ARRAY A(0 : 10),        <<INDIRECT>>
              B(0 : 10) = DB;<<DIRECT>>
LOGICAL ARRAY L(*) = A,         <<INDIRECT>>
              LM(*) = B,        <<INDIRECT>>
              LN(*);            <<INDIRECT: NEXT CELL IS
                                   ALLOCATED AS POWER>>
INTEGER D;                      <<ZERO ELEMENT OF LN>>
```

POINTER DECLARATION

A pointer declaration defines an identifier as a "pointer" — a single-word quantity used to contain a DB-relative address. The address points to 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.

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.

Syntax

<global pointer declaration>	::=	<global attribute> <atype> POINTER <pointer dec list>
<local pointer declaration>	::=	<atype> POINTER <pointer dec list> | OWN <atype> POINTER <T pointer identifier list> | EXTERNAL <atype> POINTER <T pointer identifier list> † linkage to main program compiled separately
<pointer dec>	::=	<T pointer identifier> <pointer init> | † allocated next DB or Q cell <T pointer identifier> <var reference>
<T pointer identifier>	::=	<identifier>
<pointer init>	::=	:=@ <address specification> | <empty> † not initialized
<address specification>	::=	<T simpvar identifier> | † initialization <indexed ident reference>
<indexed ident reference>	::=	<T array identifier> | † zero element <T pointer identifier> | † zero element <T array identifier> (<integer>) | <T pointer identifier> (<integer>)
<global attribute>	::=	GLOBAL | † linkage to external procedures <empty>
<atype>	::=	<type> | <empty> † LOGICAL assumed
<type>	::=	INTEGER | LOGICAL | BYTE | DOUBLE | REAL | LONG

```
<var reference>              ::= <empty> |
                                     † allocated next DB or Q cell
                                 =<T data identifier> |
                                     † address reference
                                 =<T data identifier> <sign> <usi> |
                                 <base register reference>

<T data identifier>          ::= <T simpvar identifier> |
                                 <T array identifier> |
                                 <T pointer identifier>

<sign>                       ::= +|-

<usi>                        ::= <unsigned integer>

<base register reference>    ::= DB + <usi> |
                                     † 0 to 255
                                 Q + <usi> |
                                     † 0 to 127
                                 Q - <usi> |
                                     † 0 to 63
                                 S - <usi>
                                     † 0 to 63
```

Syntax References

<array> → III, ARRAY DECLARATION

<simpvar> → III, SIMPLE VARIABLE DECLARATION

<integer> → II, INTEGER CONSTANTS

Semantics

Many of the topics discussed under "Declaration Conecpts" apply to some extent in the declaration of pointers. The reader should refer to the relevant portions of that discussion for general information on data types, address assignment, GLOBAL-EXTERNAL, variable reference, and register reference.

OWN pointers are allocated an address in the DB area but are recognized only in the procedure where declared. Pointers are initialized with addresses of other variables, not constants. The method is to follow the pointer by := and @ and a data reference (simple variable, pointer element, or array element). The address of the specified data item (adjusted to the address mode of the pointer) is stored in the cell allocated for the pointer.

```
ARRAY LOG(0:10);
POINTER P := @LOG(5);
```

See "Variables" in Section V for methods of referring to and through pointers. Pointers can be indexed like arrays and can contain word or byte data labels.

DATA TYPE

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.

STORAGE ALLOCATION

Pointers which are not address referenced are allocated the next available DB (global) or Q (local) 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.

EXAMPLE:

```
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;
```

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 within the program. In SPL/3000, labels can be declared implicitly when used to label a statement; they need not be declared explicitly unless the programmer wishes.

Syntax

<label declaration> ::= LABEL <label identifier list>

<label identifier> ::= <identifier>

Semantics

Labels are used to identify statements as follows:

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

The syntax for labeled statements is given under "Statement Types" in Section V. In SPL/3000 a label implicitly declares itself when it is used to identify a statement or as the object of a GOTO statement or in a switch declaration. It need not be explicitly declared in a label declaration except as desired for documentation purposes. See "Go Statement," Section V and "Switch Declaration," Section III for uses of labels.

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.

Syntax

<switch declaration>	::=	SWITCH <switch identifier> := <label identifier list>
<switch identifier>	::=	<identifier>
<label identifier>	::=	<identifier> † implicit label declaration

Semantics

Only one switch identifier 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 number indicates the position of the label in the list. When the switch is invoked (see "Go Statement," Section V), 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.

EXAMPLE:

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

ENTRY DECLARATION

The purpose of an entry declaration is to specify multiple entry points to a procedure or main program (beyond the implicit entry point—the first statement of the program or procedure body). Each entry identifier must occur somewhere in the body as a statement label, but cannot be the object of a GOTO.

Syntax

<entry declaration> ::= ENTRY <entry identifier list>

<entry identifier> ::= <identifier>

Semantics

An entry declaration for a procedure is equivalent to another name for the procedure that can be called with the same formal parameters, but begins execution of the procedure at a point other than the natural beginning of the procedure. Local variables are set up and initialized regardless of which entry point is used.

Programs can also have multiple entry points. By specifying the entry point to the operating system the program can be started at other than its natural beginning.

EXAMPLE:

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

REAL PROCEDURE F(X); VALUE X; REAL X;
BEGIN REAL Y := 1.354, Z := 1.0 E-5;
    ENTRY F1, F2;
        F := Y*X+Z; <<entry point for F>>
        RETURN;
    F1 : TOS := Y*X; <<entry point for F1>>
        GOTO L1;
    F2 : IF X<0.0 THEN <<entry point for F2>>
        TOS := Y+X ELSE TOS := 0.0;
    L1 : F := TOS;
END <<F, F1, F2>>;
```

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 body (i.e., optional external).

Syntax

<procedure declaration>	::=	<ctype> PROCEDURE <proc head> <proc body>
<ctype>	::=	<type> | † determines data type of function <empty> † non-function procedure
<type>	::=	INTEGER | LOGICAL | BYTE | DOUBLE | REAL | LONG
<proc head>	::=	<procedure identifier> <formal part> <option part> | <procedure identifier> ; <option part> † no parameters
<procedure identifier>	::=	<T proc identifier> | † typed or not <proc identifier>
<proc identifier>	::=	<identifier>
<formal part>	::=	(<formal param list>) ; <value part> <specification part>
<formal param>	::=	<identifier> † defined only within procedure
<value part>	::=	VALUE <identifier list> ; | † formal params <empty> † those formal params not mentioned in <part value> are passed by reference
<specification part>	::=	<specification> ; | † specify types of formal params <specification part> <specification> ;

```
<specification>          ::= <type> <identifier list> |
                                † simple variables
                             <atype> ARRAY <identifier> |
                                † array
                             LABEL <identifier list> |
                                † labels
                             <atype> POINTER <identifier list> |
                                † pointers
                             <atype> PROCEDURE <identifier list>
                                † procedures — no parameter checking

<atype>                  ::= <type> |
                                † determines data type of parameter
                             <empty>
                                † LOGICAL assumed

<option part>            ::= OPTION <option list> |
                             <empty>

<option>                 ::= UNCALLABLE |
                             PRIVILEGED |
                             EXTERNAL |
                             CHECK {unsigned integer from 0 to 3}|
                             VARIABLE |
                             FORWARD |
                             INTERRUPT |
                                † see MPE/3000 documentation on external interrupt
                                  procedures
                             INTERNAL

<proc body>              ::= <statement> |
                             BEGIN <proc data group> <procedure group>
                                <compound tail> |
                             <empty>
                                † no body when option EXTERNAL, FORWARD

<proc data group>        ::= <empty> |
                                † local declarations
                             <proc data group> <proc data declaration> ; |
                             <proc data declaration> ;

<proc data declaration>  ::= <define declaration> |
                                † any order
                             <equate declaration> |
                             <local simpvar declaration> |
                             <local array declaration> |
                             <local pointer declaration> |
                             <label declaration> |
                             <switch declaration> |
                             <entry declaration>

<procedure group>        ::= <procedure group> <subroutine declaration> |
                                <proc group>
                                      † subroutines last
```

<proc group> ::= <proc group> <procedure declaration> ; |
    † EXTERNAL procedures only
    <proc group> <intrinsic declaration> ; |
    <empty>
    † any order

<compound tail> ::= <statement> END |
    <statement> ; <compound tail>

Syntax References

<statement> → V, STATEMENT TYPES

<define> → III, DEFINE DECLARATION AND INVOCATION

<equate> → III, EQUATE DECLARATION AND INVOCATION

<local simpvar> → III, SIMPLE VARIABLE DECLARATION

<local array> → III, ARRAY DECLARATION

<local pointer> → III, POINTER DECLARATION

<label> → III, LABEL DECLARATION

<switch> → III, SWITCH DECLARATION

<entry> → III, ENTRY DECLARATION

<subroutine> → III, SUBROUTINE DECLARATION

<intrinsic> → III, INTRINSIC DECLARATION

Semantics

A procedure is a self-contained section of code which is called to perform a function. Procedures are very hardware-dependent in SPL/3000; 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 realtive 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 (i.e., a procedure can call itself). For the syntax and semantics of calling procedures see "Function Designator," Section IV and "Procedure Call Statement," Section V. Multiple entry points for procedures are covered under "Entry Declaration," this section.

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," Section IV.

If a data type is not specified, the procedure does not return a value and cannot be called as a function.

PARAMETERS

The formal parameters (if any) of a procedure must be fully specified as to type and whether each is by value or 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 or formal parameter.

Simple variables, arrays, labels, pointers, and procedures can be passed as parameters. Simple variables and pointers can be passed by value or 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," below).

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 by the user; see "Procedure Call Statement," Section V, for details).

If the formal parameter is a simple variable, it is passed the address (by reference) or actual value (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 address (by reference) or contents (by VALUE) of a pointer.

Table 3-1 shows what actual parameters can be passed to what formal parameters (a blank space is an error condition):

Table 3-1. 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	

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 is as follows:

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.

PRIVILEGED

This option causes the procedure to be run in privileged mode, assuming 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.

EXTERNAL

This option specifies that the procedure body (or code) exists external to the program being compiled. The procedure body is deleted from the rest of the declaration and is linked to the main program later by the operating system. If the programmer needs to refer to a procedure compiled separately, he 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," this section). When procedures are compiled separately (to be called later as option EXTERNAL), they can use the EXTERNAL-GLOBAL mechanism to establish data linkages (see "Declaration Concepts," this section).

CHECK

This option is provided for option external procedure declarations and full 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 parameters is used to determine the level of checking (except that intrinsics determine their level of checking, never the caller). The check values are:

0 — no checking

1 — check procedure type only

2 — checks procedure type and number of parameters.

3 — checks procedure type, number of parameters, and type of each parameter.

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 e.g., NOW(A,,C); the corresponding bit in the mask is set to zero. The correspondence is from right to left (the rightmost bit—bit 15—corresponds to the right parameter). In the procedure call, the occurrence of a right parenthesis before the parameter list is filled, implies that the rest of the parameters are missing. When the procedure is entered, it is the responsibility of the procedure to examine the bit mask. Parameters will always occur in the same Q– addresses, but missing parameters will have garbage in their locations.

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).

INTERRUPT

This option specifies that the procedure is an external interrupt procedure. The structure and uses of interrupt routines are covered in *HP 3000 Multiprogramming Executive Operating System (03000-90005).*

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 another procedure.

LOCAL DECLARATIONS

Procedures can declare local variables that are known only within the procedure and are allocated space in the Q+ area when the procedure is called. Thus, they occupy space only when 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 topic in this section.)

Many of the differences of local declarations are discussed under "Declaration Concepts," this section.

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 initialized, they are initialized at the beginning of the program, not every time the procedure is called. Since they are allocated in DB, they are not deleted when a procedure exits, but are still in existence (with their last value) when the procedure is called again.

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 SPL/3000 statements. If the body contains no 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," Section V).

EXAMPLE:

```
PROCEDURE BLANKBUF <<NAME>>
     (BUFFER,COUNT); <<FORMAL PARAMETERS>>
     VALUE COUNT; <<VALUE PART>>
     LOGICAL ARRAY BUFFER; <<SPECIFICATION>>
     INTEGER COUNT; <<PART>>
     <<EMPTY OPTION PART>>
<<BODY>>
     BEGIN
          LOGICAL BLANK WORD := "  "; <<DATA GROUP>>
          BUFFER := BLANKWORD ; <<STATEMENTS>>
          MOVE BUFFER(1) := BUFFER,(COUNT);
     END; <<END 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);<<THIS IS EQUIVALENT TO THE STATEMENT:>>
                                            <<NIX := NUM / ((4+5)*6);>>
     END.
<<OPTION FORWARD EXAMPLE>>
PROCEDURE PROC1;OPTION FORWARD <<dummy declaration>>
PROCEDURE PROC2;OPTION FORWARD; <<dummy declaration>>

PROCEDURE PROC1;IF X = (Y := Y+1) THEN PROC2; <<real declaration>>
PROCEDURE PROC2;IF X = (Z := Z+1) THEN PROC1; <<real declaration>>
```

INTRINSIC DECLARATION

An intrinsic declaration specifies that one of the system-provided procedures (an intrinsic) will be used by the program. Intrinsics are pre-compiled procedures that are supplied to SPL/3000 programmers for performing input/output, file access, and utility functions as part of the Multiprogramming Executive (MPE/3000). SPL/3000 provides a simple interface to intrinsics because SPL/3000 has no built-in construct for input/output (unlike FORTRAN, BASIC, COBOL, and other high-level languages). Input and output of data in SPL/3000 programs must be performed by using the MPE/3000 file system intrinsics. The user can also declare intrinsics from his own intrinsic file.

Syntax

```
<intrinsic declaration>  ::=  INTRINSIC <intrinsic identifier list> |
                               INTRINSIC (<file>) <intrinsic identifier list>
<intrinsic identifier>   ::=  <identifier>
                                  †equivalent to an option EXTERNAL declaration of
                                   each intrinsic
<file>                   ::=  { any valid random-access file of the operating system }
```

Semantics

The identifiers in an intrinsic list must be included in an installation-defined intrinsic file. The SPL/3000 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," this section) and specifies the procedure's parameters, etc. Operating system intrinsics are described in *HP 3000 Multiprogramming Executive Operating System (03000-90005).* 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 G describes how to build intrinsic files.

EXAMPLES:

```
INTRINSIC FOPEN,FREAD,FWRITE,PRINT,READ;
INTRINSIC (MYFILES) ASCII, CONVERT, OUTPUT, DATA'MAP3;
```

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 means of 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.

Syntax

<subroutine declaration>	::=	<stype> SUBROUTINE <sub head> <sub body>
<sub head>	::=	<subroutined identifier> <formal part> | <subroutine identifier> † no parameters
<sub body>	::=	<statement>
<subroutine identifier>	::=	<T subr identifier> | † typed <subr identifier> † not typed
<subr identifier>	::=	<identifier>
<formal part>	::=	(<formal param list>) ; <value part> <specification part>
<formal param>	::=	<identifier> † defined only within subroutine
<value part>	::=	VALUE <identifier list> ; | † formal params <empty> † Those formal params not mentioned in the <value part> are passed by reference
<specification part>	::=	<specification> ; | † specify types of formal params <specification part> <specification>
<specification>	::=	<type> <identifier list> | † simple variables <atype> ARRAY <identifier list> | † arrays <atype> POINTER <identifier list> | † pointers <ctype> PROCEDURE <identifier list> † procedures — no parameter checking

```
<stype>              ::=  <type> |
                              † determines data type of function
                          <empty>
                              † non-function subroutine

<ctype>              ::=  <type> |
                              † specifies data type of function procedure parameter
                          <empty>
                              † non-function procedure parameter

<atype>              ::=  <type> |
                              † specifies data type of array/pointer parameter
                          <empty>
                              † LOGICAL assumed

<type>               ::=  INTEGER | LOGICAL | BYTE | DOUBLE | REAL | LONG
```

Syntax References

<statement> → V, STATEMENT TYPES

Semantics

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 any SPL/3000 statement, including a compound statement, but cannot contain declarations. Global subroutines can reference global variables and local subroutines can reference local and global variables. Subroutines can be called recursively. Subroutines are called using the SCAL instruction and return using SXIT. For details on calling subroutines, see "Function Designator," Section IV, and "Subroutine Call Statement," Section V.

EXAMPLE:

```
INTEGER SUBROUTINE S(A,B,C);
    VALUE A,B,C;
    INTEGER A,B,C;
    S := (A^2) + (B*C);
SUBROUTINE ZERO (ARRAY,HISUB);
    VALUE HISUB;
    INTEGER HISUB;
    INTEGER ARRAY ARRY;
BEGIN
    I := 0; <<global variable>>
    WHILE I < = HISUB>
        BEGIN
            ARRY (I) := 0;
            I := I + 1;
        END;
END;
```

SECTION IV
Expressions

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 (byte, integer, double, real, 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 a hardware and/or software conditions.

Syntax

<expr>	::=	<T expr> † results in value of type T
<T_{irbde} expr>	::=	<aexp> | † arithmetic expression <T_{irbde} IF expr>
<logical expr>	::=	<lexp> | † logical expression <logical IF expr>
<T IF expr>	::=	IF <cond clause> THEN <expr> ELSE <expr> † both <expr> must be of same word size; byte treated as one word

Syntax References

<lexp>	→	IV,	LOGICAL EXPRESSIONS
<aexp>	→	IV,	ARITHMETIC EXPRESSIONS
<cond clause>	→	V,	IF STATEMENT

Semantics

Expressions are used to determine values to be used in statements. Where <expr> is specified in the syntax any type of expression is allowed: arithmetic, logical, or IF. When <aexp> is specified, only an expression resulting in a numerical data type is allowed. When <lexp> is specified only an expression resulting in a logical value is allowed. The T mechanism (see "Introduction") is often used to specify an expression resulting in only certain data types.

The IF expression consists of two alternative expressions and a condition clause. The two expressions must be of the same word size (byte is treated as one word). The condition clause is a combination of logical expressions and hardware branch words which results in a true or false value. If the condition clause is true, the expression after THEN is selected; if the condition clause is false, the expression after ELSE is selected. For the definition of condition clause, see "IF Statement," Section V.

Within SPL/3000 expressions, only variables of the same data type can appear on either side of an operator. That is, integer can be multiplied by integer, but not by real. The only exception to this is the exponentiate operator (^) in arithmetic expressions; real and long data types can be exponentiated to integer powers. In all other cases the combination of data items of differing 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(<$T_r$expr>)

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

FIXT(<$T_r$expr>)

There are not type transfer functions 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(<$T_r$expr>)) or INTEGER(FIXT($T_r$expr>))).

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	—

NOTE: LOGICAL (Double) leaves the 32-bit value on the stack so that a 32-bit divide (//) can be performed. If only 16 bits of the double are desired, use LOGICAL (INTEGER (double)).

VARIABLES

Variables are one of the items that occur in expressions. Each variable, whether it be a simple variable, an array element, a pointer reference, or the top of 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.

Syntax

<T variable>	::=	<T simpvar identifier> | <T pointer identifier> <index> | <T array identifier> <index> TOS † top of stack
<integer variable>	::=	@ <T simpvar identifier> | † 16 bit DB-relative address @ <T pointer identifier> <index> | @ <T array identifier> <index> | † DB or PB relative @ <label identifier> | † PB-relative address @ <procedure identifier> | † PB-relative address of entry point @ <entry identifier> | † PB-relative address of entry point ABSOLUTE (<index>) † privileged access to contents of absolute location <index>
<index>	::=	<empty> | † zero element (<T_{ilb} expr>) | † element number (<T_{ilb} assignment statement>)

Syntax References

<simpvar>	→	III,	SIMPLE VARIABLE DECLARATION
<pointer>	→	III,	POINTER DECLARATION
<array>	→	III,	ARRAY DECLARATION
<assignment statement>	→	V,	ASSIGNMENT STATEMENT
<expr>	→	IV,	EXPRESSION TYPES

Semantics

The three most common types of variables (they occur in all data types) are the simple variable, the array reference, or 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. (This is more efficient than explicitly specifying 0 as the index; in the first case the index register is not used, in the second it is.)

TOS

This 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 := <expr>; see "Assignment Statement," Section V), the normal store operation is omitted and the result of the expression 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 two words, or in another, the top three words. Note that TOS does not refer to the same memory location from one statement to the next, since S is constantly changing. Also, the default type for TOS is integer. A 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. There is only one exception to this:

```
INT. (BIT : FIELD) := TOS;<<XCH;DPF;STOR>>
TOS := 7; <<LOAD 7>>
A := TOS + 6; <<A = 13>>
REAL1 := -TOS <<illegal if REAL1 is real>>
```

ADDRESSES (@)

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 (<array 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)>>
```

ABSOLUTE

This 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>) := <expr>; see "Assignment Statement," Section V), a PSTA (privileged store) instruction is generated which stores the top of the stack (<expr>) 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, Q2, A3 = X;
    L1 := ABSOLUTE (A1 * A2);
    ABSOLUTE (L2) := A1 + 5;
    ABSOLUTE (A3) := A1 + 5; <<A3 is index register>>
    L1 := ABSOLUTE (ABSOLUTE (3));
    L1 := ABSOLUTE (A3);
```

FUNCTION DESIGNATOR

Function designators are another of the possible components of an expression. A function designator specifies a function identifier (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.

Syntax

```
<T function designator> ::= <T proc identifier> <actual param part> |
                            <T subr identifier> <actual param part>
                                  † call to function procedure or subroutine

<actual param part>     ::= <empty> |
                                  † no parameters
                            (<actual param list>) |
                                  † no stacked params
                            (<stacked param list>) |
                                  † all stacked params
                            (<stacked param list>, <acutal param list>)
                                  † stacked params must come first

<actual param>          ::= <reference param> |
                                  † passes an address
                            <value param> |
                                  † passes a value
                            <empty>
                                  † missing parameter; option VARIABLE only

<stacked param>         ::= *
                                  † address or value and zero for function result are already
                                    stacked by user

<reference param>       ::= <T simpvar identifier> |
                            <T array identifier> <index> |
                            <T pointer identifier> <index> |
                            <procedure identifier> |
                            <label identifier>

<value param>           ::= <aexp> |
                            <lexp> |
                            <assignment statement>
```

Syntax Reference

<T proc identifier>	→	III, PROCEDURE DECLARATION
<T subr identifier>	→	III, SUBROUTINE DECLARATION
<simpvar>	→	III, SIMPLE VARIABLE DECLARATION
<array>	→	III, ARRAY DECLARATION
<pointer>	→	III, POINTER DECLARATION
<label>	→	III, LABEL DECLARATION

<assignment statement> → V, ASSIGNMENT STATEMENT

<index> → IV, VARIABLES

<aexp> → IV, ARITHMETIC EXPRESSIONS

<lexp> → IV, LOGICAL EXPRESSIONS

Semantics

The function, procedure or subroutine must have been previously declared (see "Procedure Declaration," and "Subroutine Declaration," Section III). The actual parameters must match one-to-one the formal parameters as specified in the declaration; correspondence is checked left to right.

A stacked parameter is specified by an asterisk (*) in place of an actual parameter; this specifies that the necessary address or value has already been loaded onto the stack by the user. Labels cannot be stacked. If any parameter is stacked, all parameters to its left must be stacked too. In addition, functions require that a 1-, 2-, or 3-word zero (depending on function type) be pushed onto the stack before the function parameters for the return value. Normally, the compiler provides this automatically. However, if stacked parameters are used, the programmer must arrange for this zero. For example,

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

For more details on calling procedures and subroutines, see "Procedure Call Statement," and "Subroutine Call Statement," Section V.

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

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 consist 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.

Syntax

<T bit operation>	::=	<T_{ilb} bit extraction> | <T_{ilb} bit concatenation> | <T bit shift>
<T_{ilb} bit extraction>	::=	<T_{ilb} primary>. (<bit extract field>)
<bit extract field>	::=	<left extract bit> : <extract field length>
<left extract bit>	::=	<unsigned integer> † bit to start with from 0 through 15
<extract field length>	::=	<unsigned integer> † number of bits to extract from 1 through 15
<T_{ilb} bit concatenation>	::=	<T_{ilb} primary> CAT <T_{ilb} primary> (<bit cat field>)
<bit cat field>	::=	<left deposit bit> : <bit extract field>
<left deposit bit>	::=	<unsigned integer> † bit to start deposit with 0 through 15
<T bit shift>	::=	<T primary> & <shift op> (<shift count>)
<shift op>	::=	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

<shift op> (cont.)		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
<shift count>	::=	<integer expr> † if this is not a constant the index register is used to hold the variable shift count

Syntax References

<unsigned integer> → II, INTEGER CONSTANTS

<primary> → IV, ARITHMETIC EXPRESSIONS, LOGICAL EXPRESSIONS

<integer expr> → IV, EXPRESSION TYPES

Semantics

Bit extraction and concatenation are defined for one word quantities only. Bit shifts are provided for one, two, and three word quantities. In all bit expressions the original primaries operated upon are unchanged by the operation. See "Assignment Statement," Section V, for bit deposit.

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 primary specified by <left extract bit> and continues for <extract field length> bits to the right, wrapping around to bit 0, if necessary. For example,

```
%125.(8:2)  <<result is %1>>
```

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 deposit bit> indicates in which bit position of the first primary (to the left of CAT) to deposit the field extracted from the second. The <left extract bit> indicates at which position in the second primary to begin extracting the bit field. The <extract field length> indicates how many contiguous bits to

extract and subsequently deposit. Bit concatenation uses both the EXF (extract field) and DPF (deposit field) instructions. For example,

```
%(16)69A2 CAT %(16)ABCD(8:4:4) <<result is %(16)69B2>>
```

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 zeroes) and propogate the sign bit on a right shift (e.g., fill with the sign bit); and *circular* shifts have no fill bit (e.g., bits shifted off one end are shifted in at the other end). SPL/3000 performs no type or word-size tests in bit shifts; if the programmer specifies a triple shift on a single word quantity, a triple shift is generated. The programmer is responsible for maintaining compatibility. Note that if the shift count is not a constant less than 64, the index register is used. For example,

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

ARITHMETIC EXPRESSIONS

An arithmetic expression is a sequence of operations upon numerical 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 backword slashes (absolute value), function designators, or assignment statements in parentheses.

Syntax

<T_{irbde} aexp> ::= <aexp> | <addop> <aexp>
   † lowest precedence

<aexp> ::= <T_{irbde} term> |
   <aexp> <addop> <T_{irbde} term>

<T_{irbde} term> ::= <T_{irbde} factor> |
   <T_{irbe} term> <mulop> <T_{irbe} factor>
   † no double multiply

<T_{irbde} factor> ::= <T_{irbde} primary> |
   <T_{ire} factor> ^ <T_{ire} primary>
   † exponentiation
   † allowable combinations are: i^i, r^r, r^i, e^e, e^i

<T_{irbde} primary> ::= <T_{irbde} variable> |
   † highest precedence
   <constant> |
   † if number, must match type of other operand; if string, 1 character can be used as T_{ilb}; 2 characters as T_{il}, and 3 or 4 characters as T_d only. Expressions containing only integer constants are considered type integer, not logical

   <T_{irbde} bit operation> |
   (<aexp>) |
   \<aexp>\ |
   † absolute value
   <T_{irbde} function designator> |
   (<T_{irbde} assignment statement>)

<addop> ::= + | -

<mulop> ::= * | / | MOD

Syntax References

<variable>	→ III,	VARIABLES
<constant>	→ II,	CONSTANT TYPES
<bit operation>	→ IV,	BIT OPERATIONS
<function designator>	→ IV,	FUNCTION DESIGNATORS
<assignment statement>	→ V,	ASSIGNMENT STATEMENT

Semantics

An arithmetic expression defines a sequence of operations, which results in a single value of a certain data type that is the expression's result. What is done with this value depends upon where the expression occurs.

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 rank from highest to lowest, is as follows:

Rank 1: <primary>

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

Rank 2: <factor>

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

Rank 3: <term>

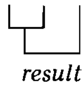
- Multiply (*) and divide (/) for integer, real, byte and long data; no double multiply or divide.
- Modulo (MOD) or remainder for integer and byte data.

Rank 4: <addop>

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

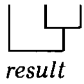
The order in which operations are performed is determined by this rank. For example,

A-B+C



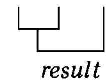
Operators of the same rank are performed from left to right.

A+B*C



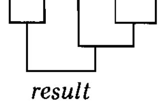
Operators of different rank are performed according to their position in the hierarchy of operators (highest rank first).

(A+B)*C



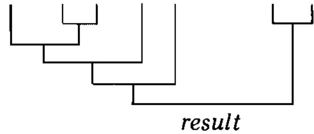
Operators enclosed in parentheses take precedence over operators outside of parentheses, even those of higher rank.

A-B+C*D^E



Left-to-right order is maintained until an operator occurs that is of lower rank than the next operator or the next item is in parentheses.

A^(B-C)*D/E MOD P^G



TYPE MIXING

Mixing of data types across operands is not allowed in SPL/3000, except that real and long values can be exponentiated to integer powers. Type transfer functions are available to handle conflicts (see "Expression Types," this section).

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

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 specialized 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). 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 and LOR should not be confused with AND and OR (see "IF STATEMENT," Section V).

Syntax

```
<lexp>              ::= <disjunction> |
                              † lowest precedence
                        <lexp> LOR <disjunction> |
                              † logical or
                        <integer aexp> <= <integer aexp> <= <integer aexp>
                              † compare Range and Branch, uses index register

<disjunction>       ::= <conjunction> |
                        <disjunction> XOR <conjunction>
                              † exclusive or

<conjunction>       ::= <logical elem> |
                        <conjunction> LAND <logical elem>
                              † logical and

<logical elem>      ::= <logical term> |
                        <logical term> <relop> <logical term> |
                              † logical compare
                        <T_irbde aexp> <relop> <T_irbde aexp> |
                              † arithmetic compare
                        <byte ref> <relop> <byte ref> <count> <sdeca> |
                        <byte ref> <relop> *PB <count> <sdeca> |
                              † stacked PB address
                        <byte ref> <relop> <string> <sdeca> |
                        <byte ref> <relop> (<listelmt>) <sdeca> |
                        <byte variable> = <btestword> |
                              † test type of byte
                        <byte variable> < > <btestword>

<logical term>      ::= <logical factor> |
                        <logical term> <logical addop> <logical factor>

<logical factor>    ::= <logical primary> |
                        <logical factor> <logical mulop> <logical primary>

<logical primary>   ::= <logical variable> |
                              † highest precedence
                        <logical value> |
                        <string> |
                              † 1 or 2 characters only
                        <logical bit operation> |
                        (<lexp>) |
```

<logical primary> (cont.)	::=	<logical function designator> | (<logical assignment statement>) | NOT <logical primary> † one's complement
<relop>	::=	> | † greater than < | † less than = | † equals < > | † not equals < = | † greater than or equal < = † less than or equal
<logical addop>	::=	+ | -
<logical mulop>	::=	* | † multiply — 16-bit result / | † divide — 16-bit dividend MOD | † remainder — 16-bit dividend ** | † multiply — 32-bit result // | † divide — 32-bit dividend MODD † remainder — 32-bit dividend
<byte ref>	::=	<byte pointer identifier> <index> | <byte array identifier> <index> | * † stacked byte address
<count>	::=	, (<integer expr>)
<btestword>	::=	ALPHA | † "A" through "Z" and "a" through "z" NUMERIC | † "0" through "9" SPECIAL † all other characters
<sdeca>	::=	<empty> | † delete all values , <sdec>
<sdec>	::=	0|1|2 † number of words to delete
<listelmt>	::=	<initial value> | <decimal integer> (<initial value list>) | † repeat factor <listelmt list> † no nesting of repeats
<initial value>	::=	<constant> † truncated to 8 bits

Syntax References

<index>	→	IV,	VARIABLES
<aexp>	→	IV,	ARITHMETIC EXPRESSIONS
<string>	→	II,	STRING CONSTANTS
<logical variable>	→	IV,	VARIABLES
<logical value>	→	II,	LOGICAL CONSTANTS
<bit operation>	→	IV,	BIT OPERATIONS
<function designator>	→	IV,	FUNCTION DESIGNATORS
<assignment statements>	→	V,	ASSIGNMENT STATEMENT
<pointer identifier>	→	III,	POINTER DECLARATION
<array identifier>	→	III,	ARRAY DECLARATION
<constant>	→	II,	CONSTANT TYPES

Semantics

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 "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/3000 are 16-bit positive integers, (see "Logical Constants," Section II). A logical value is considered true if its integer value is odd, false if its value is even (that is, only the last bit is checked when using the result of a logical expression to make a decision. Use of the reserved words TRUE and FALSE is equivalent to the numeric values -1 and 0 (%177777 and %000000).

In general, the result of a logical expression is left as a full word operand on the top of the stack. An exception is when a relational operator is encountered, in which case a value -1 (true) or 0 (false) is left on the top of the stack. A further exception is when the result of a relational operator is used to make a decision (see <cond clause> under "IF Statement," Section V); in this case, nothing is left in the stack and the status register is examined for the result.

SEQUENCE OF OPERATIONS

Logical 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 rank, from highest to lowest, is as follows:

Rank 1: <primary>

- Logical bit operation.
- Logical expression in parentheses.
- Logical function designator.
- Logical assignment statement in parentheses (value assigned to variable and left on the stack).
- NOT (unary one's complement)

Rank 2: <factor>

- * (Logical multiply, one word result).
- / (Logical divide, one word dividend).
- MOD (Logical modulo or remainder, one word dividend).
- ** (Logical multiply, two-word result left on stack for user).
- // (Logical divide, requires two-word dividend; if dividend equals a variable, two words are loaded from the location assigned to be variable; if the dividend is a partially evaluated expression, the compiler assumes a two-word dividend on the top of the stack).
- MODD (Logical modulo or remainder, requres two-word dividend; same conventions as //).

Rank 3: <term>

Logical addition (+) and subtraction (-); there is no logical unary minus.

Rank 4: <elem>

Algebraic and logical comparisons (=, <, >, < >, <=, >=); logical compares use the LCMP instruction to perform a 16-bit comparison which treats bit 0 as a data bit, not a sign bit; algebraic compares use one of three instructions (CMP, DCMP, FCMP) which perform comparisons taking into account the sign bit (negative numbers are less than positive numbers); the result is a TRUE (-1) if the relation specified holds or a FALSE (0) if it does not.

Compare bytes (see below).

Test bytes (byte variable is (=) or is not (< >) type alphabetic (ALPHA), numeric (NUMERIC), or other (SPECIAL)).

Rank 5: <conjunction>

Logical and of 16 bits (LAND).

Rank 6: <disjunction>

Logical exclusive or of 16 bits (XOR).

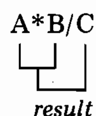
Rank 7: <lowest>

Logical inclusive or of 16 bits (LOR).

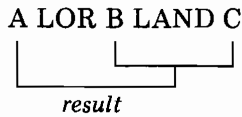
Compare integer range (A <= B <= C, where A, B, C, are integer expressions; this uses the CPRB (compare range and branch instruction) instruction and the index register; result is TRUE (-1) if the middle integer is within the range, FALSE (0) if not.

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

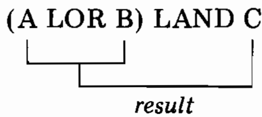
A*B/C



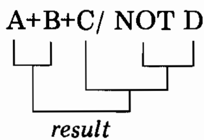
Operators of the same rank are performed from left to right.



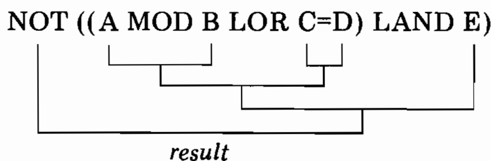
Operators of different rank are performed according to their position in the hierarchy of operators (highest rank first).



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 in parentheses.



TYPE MIXING

Mixing of data types across operands is not allowed in SPL/3000, but type transfer functions are available to handle conflicts. In logical expressions, logical operands are always used, except in the special cases where the operands are arithmetic, but the result is logical (compares, byte tests, and range tests). See "Expression Types," this section for all type transfer functions.

COMPARING BYTE STRINGS

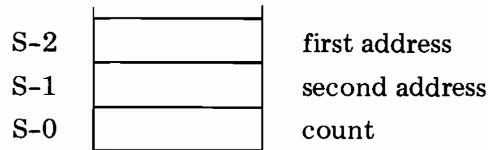
Logical expressions provide a mechanism for comparing strings of bytes to determine whether a relation between them is true or false. The instruction generated is CMPB (compare bytes). The byte strings are compared, one by one, at their numeric values until the compared bytes are unequal or until a specified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

The string to the left of the <relop> can be specified by a byte pointer or array reference (DB relative only) or a stacked DB byte address (*). The asterisk specifies that the user has already loaded the stack with the byte address.

The string to the right can be specified by a byte pointer or array reference (PB or DB relative), a stacked DB address (*), a stacked PB address (*PB), a literal string (no <count>), or a list of contents in parentheses (see "Array Declaration," Section III) with no <count>.

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

The <sdeca> operand specifies how many of the temporary values used by CMPB to delete from the stack. An empty <sdeca> specifies sdec of 3 or delete all values. The values are loaded on the stack in this order:



For example,

```
TARGET = SOURCE, (20)
* = *, (72)
* < "SINCERE", 0
BAR(5) > "BOB SMITH", 0
```

SECTION V
Statements

STATEMENT TYPES

A statement is an order to perform some action. Statements are the components for the bodies of programs and procedures. It is possible to label any statement and to combine any sequence of statements into a single statement called a compound statement.

Syntax

```
<statement>            ::= <label identifier> := <statement> |
                           <compound statement> |
                           <assignment statement> |
                           <GO statement> |
                           <IF statement> |
                           <CASE statement> |
                           <FOR statement> |
                           <DO statement> |
                           <WHILE statement> |
                           <MOVE statement> |
                           <SCAN statement> |
                           <PROCEDURE call statement> |
                           <RETURN statement> |
                           <SUBROUTINE call statement> |
                           <DELETE statement> |
                           <PUSH and SET statement> |
                           <ASSEMBLE statement> |
                           <empty>
<compound statement>   ::= BEGIN <compound tail>
<compound tail>        ::= <statement> END |
                           <statement> ;<compound tail>
```

Syntax References

<label identifier> → III, LABEL DECLARATION

All of the statement types referred to are covered in this section (in alphabetic order).

ASSEMBLE STATEMENT

The purpose of the ASSEMBLE statement is to generate any code desired by specifying literal hardware instructions. Instructions within an ASSEMBLE statement can be labeled (and branched to from without). Identifiers outside the ASSEMBLE statement can be used within, but indirect references are never provided unless they are specified explicitly.

Syntax

```
<ASSEMBLE statement>  ::=  ASSEMBLE (<instruction slist>)

<instruction slist>   ::=  <instruction> |
                           <instruction slist> ; <instruction>
                                † note that semicolon is delimiter

<instruction>         ::=  <label identifier> : <opcode format> |
                           <opcode format>

<opcode format>       ::=  <format-1> | <format-2> | <format-3> |
                           <format-4> | <format-5> | <format-6> |
                           <format-7> | <format-8> | <format-9>

<format-1>            ::=  <memory ref opcode> <address part> <I-field> <X-field> |
                           <sub memref op> <label identifier>

<memory ref opcode>   ::=  <sub memref op> |
                           STOR | INCM | DECM | LDB |
                           LDD | STB | STD

<sub memref op>       ::=  LOAD | LDX | LRA | CMPM | ADDM | SUBM | MPYM |
                           BR | BL | BE | BLE | BG | BNE | BGE |
                           TBA | MTBA | TBX | MTBX

<address part>        ::=  <var identifier> | <addr mode> <usi 255>

<var identifier>      ::=  <T simpvar identifier> |
                           <T pointer identifier> |
                           <T array identifier>

<addr mode>           ::=  DB+ | Q+ | Q- | P+ | P- | S-

<usi 255>             ::=  <unsigned integer>
                                † less than an equal to 255

<I-field>             ::=  ,I |
                                † indirect
                           <empty>
                                † direct

<X-field>             ::=  ,X |
                                † indexing
                           <empty>
                                † no indexing

<format-2>            ::=  <stack opcode> |
                                † fills with NOP
                           <stack opcode> , <stack opcode>
```

<stack opcode> ::= NOP | DELB | DOEL | 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 |
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

<format-3> ::= <branch subop1> <arg1> <I-field> |
<non-branch subop1> <usi63> <X-field>

<usi31> ::= <unsigned integer>
† less than or equal to 31

<usi63> ::= <unsigned integer>
† less than or equal to 63

<arg1> ::= <label identifier> |
P <sign> <usi31> |
* <sign> <usi31>

<sign> ::= + | -

<branch subop1> ::= IABZ | IXBZ | DXBZ | DABZ | BCY | BNCY |
CPRB | BOV | BNOV | BRO | BRE |

<non-branch subop1> ::= ASL | ASR | LSL | LSR | CSL | CSR | SCAW |
TASL | TASR | TNSL | DASL | DASR | DLSL |
DLSR | DCSL | DCSR | TBC | TRBC | TSBC |
TCBC

<format-4> ::= <sub subop2> <usi255> |
EXF <usi> : <usi>
DPF <usi> : <usi>

<sub subop2> ::= LDI | LDXI | CMPI | ADDI | SUBI | MPYI |
DIVI | PSHR | LDNI | LDXN | CMPN | SETR

<format-5> ::= RSW | LLSH | PLDA | PSTA

<format-6> ::= <special op> <K-field>

<K-field> ::= <unsigned integer>
† less than or equal to 15

<special op> ::= PAUS | SED | XCHD | SMSK | RMSK | XEQ |
SIO | RIO | WIO | TIO | CIO | CMD | SIRF |
SIN | HALT

<format-7> ::= <subop3> <usi255> |
PCAL <procedure identifier> |
SCAL 0 |
LLBL <procedure identifier>

<subop3> ::= PCAL | SCAL | EXIT | SXIT | ADXI | SBXI |
LLBL | LDPP | LDPN | ADDS | SUBS | TSBM |
ORI | XORI | ANDI

```
<format-8>          ::= <sub move op> <sadmode> |
                              † delete all values
                        <sub move op> <sadmode> <sdec> |
                        MVBW <ccf> <sdeca> |
                        <scan op> |
                              † delete all values
                        <scan op> <sdec>

<sadmode>           ::= <empty> |
                              † DB
                        PB
                              † PB rel. address

<sdeca>             ::= <empty> |
                              † delete all values
                        <sdec>

<sdec>              ::= 0 | 1 | 2

<ccf>               ::= A | N | AN | AS | ANS

<sub move op>       ::= MOVE | MVB | CMPB

<scan op>           ::= SCW | SCU | MVBL | MVLB

<format-9>          ::= CON <const list> |

<const>             ::= <constant> |
                        <label identifier>
                              † creates PB address
```

Syntax References

<label>	→	III,	LABEL DECLARATION
<simpvar>	→	III,	SIMPLE VARIABLE DECLARATION
<pointer>	→	III,	POINTER DECLARATION
<array>	→	III,	ARRAY DECLARATION
<unsigned integer>	→	II,	INTEGER CONSTANTS
<procedure>	→	III,	PROCEDURE DECLARATION
<subroutine>	→	III,	SUBROUTINE DECLARATION
<entry>	→	III,	ENTRY DECLARATION
<constant>	→	II,	CONSTANT TYPES

Semantics

The ASSEMBLE statement allows the programmer to generate machine instructions of his choice. Appendix E contains general information on the HP 3000 machine instructions, but the programmer should refer to the hardware documentation for exact details. The

opcodes as listed in the hardward manuals are used in ASSEMBLE except that BCC (branch on condition code) has been replaced by six mnemonics which specify the exact branch condition:

BL	Branch if less than (condition code equals 1)
BE	Branch if equal (condition code equals 2)
BLE	Branch if less than or equal (condition code equals 1 or 2)
BG	Branch if greater than (condition code equals 0)
BNE	Branch if not equal (condition code equals 0 or 1)
BGE	Branch if greater than or equal (condition code equals 0 or 2)

The compiler does not modify P-relative displacements within an ASSEMBLE. Consequently, branches out of range cause an error. The programmer must explicitly specify indirect whenever that is desired.

The format-9 opcodes are psuedo-opcodes; they do not generate machine instructions but are used to generate constants within the code at the next PB-relative location. The CON opcode can generate indirect addresses for labels as well as numerical and string constants.

EXAMPLES:

```
PROCEDURE OCTOUT(BUF,T,N); VALUE T,N;
     BYTE ARRAY BUF;
     LOGICAL T;
     INTEGER N;
BEGIN    <<CONVERT N OCTAL DIGITS OF T TO ASCII CHARACTERS IN BUF>>
     LABEL LOOP;
     ASSEMBLE (
          LDX N;
          DECX, NOP;
          LOAD T;
LOOP:     DUP;
          ANDI 7;
          ADDI %60;
          STB BUF,I,X;
          LSR 3;
          DECX;
          BGE LOOP);
END;
```

ASSIGNMENT STATEMENT

The purpose of an assignment statement is to store the result of an expression into a variable of the same size as the result. Multiple assignments allow the same result to be stored in several variables and bit deposits allow a one-word result to be stored into a variable starting at a specific bit position. Assignment statements can be of any data type (i.e., the variable assigned a value can be of any type).

Syntax

<T assignment statement> ::= <T left part> := <right part> |
<T left part> := <assignment statement>
† multiple assignment

<T left part> ::= <T variable> |
<T_{il} variable> . (<deposit field>)
† when used with multiple assignments only the leftmost assignment can be a deposit

<deposit field> ::= <left deposit bit> : <deposit field length>

<left deposit bit> ::= <unsigned integer>
† bit to start deposit with; 0 through 15

<deposit field length> ::= <unsigned integer>
† 1 through 15

<right part> ::= <T expr>
† must match left part in number of words

Syntax References

<variable> → IV, VARIABLES

<unsigned integer> → II, INTEGER CONSTANTS

<expr> → IV, EXPRESSION TYPES

Semantics

The result of an expression is stored in the variables (simple variable, array, or pointer) specified on the left side of the assignment operator (:=). The result must be of the same word size (but not necessarily the same type) as the assignment variable. Type byte is treated as if it were one word.

When a deposit is specified, the result must be a one-word quantity. The number of contiguous bits required (<deposit field length>) is taken from the rightmost bits of the result and deposited (DPF instruction) in the variable, starting with the bit position specified (<left deposit bit>).

If a deposit is combined with multiple assignments, only the leftmost assignment can be a deposit

Instructions used in assigning values include STOR (store word), STB (store byte), and STD (store double).

EXAMPLES:

```
INTEGER I,J; LOGICAL K,L;
BYTE B1, B2; REAL R1, R2;
DOUBLE D;
      I := K*L;
      I(5:6) := J := L,
      I(0:8) := B1;
      R1 := R1 := R1+(R2*REAL(I));
      D := R1;
```

CASE STATEMENT

The purpose of a CASE statement is to select one of a set of statements for execution by using a variable index in a compound statement. The first statement has index 0 and the others are numbered consecutively (1, 2, 3....). After execution of the specified statement, control transfers to the statement follow the CASE statement.

Syntax

<CASE statement>	::=	CASE <T_{ilb} expr> OF <case body> |
		CASE * <T_{ilb} expr> OF <case body>
		† no bounds checking
<case body>	::=	<compound statement>
<compound statement>	::=	BEGIN <compound tail>
<compound tail>	::=	<statement> END |
		<statement> ; <compound tail>

Syntax References

<exp>	→	IV,	EXPRESSION TYPES
<statement>	→	V,	STATEMENT TYPES

Semantics

A CASE statement contains an integer expression specifying the statement desired and a compound statement. The statements in the compound statement are numbered consectively starting with 0. For example,

```
CASE J OF
     BEGIN
          A := 100;  <<#0>>
          B := 200;  <<#1>>
          BEGIN
               C := 300;
               IF A<B THEN D := 100;
          END;
          QR := 500; <<#3>>
     END;
```

Normally, if the integer expression evaluates to less than zero or greater than the acceptable indices, control transfers to the statement following the CASE statement. However, if the * option is specified, no bounds checking is performed and invalid indices will cause unpredictable results.

EXAMPLES:

```
CASE (N*2)+(QR-3) OF
     BEGIN
          GOTO FIRST;
          GOTO SECOND;
          IF A <300 GOTO THIRD;
          <<null statement #4>>;
          IF A > 300 GO TO THIRD;
          A := 512;
     END;
CASE * J OF
     BEGIN    A := 0;
              A := 1;
              A := 2;
              A := 3;
              A := 4;
     END;
```

DELETE STATEMENT

The purpose of the DELETE statement is to generate one of these three hardware instructions:

- DEL (delete contents of S-0, decrement S register by one).
- DELB (delete contents of S-1 by storing contents of S-0 into it; decrement S register by one).
- DDEL (delete contents of S-0 and S-1; decrement S register by two).

Syntax

```
<DELETE statement>   ::=  DEL |
                                † delete TOS
                          DELB |
                                † delete B
                          DDEL
                                † double delete
```

Semantics

The Delete statements have the following effect:

	Before DEL		After DEL
S-2	7	S-1	7
S-4	6	S-0	6
S-0	5		

	Before DELB		After DELB
S-2	7	S-1	7
S-1	6	S-0	5
S-0	5		

	Before DDEL		After DDEL
S-2	7	S-0	7
S-1	6		
S-0	5		

These statements should be used with caution as they override the compiler's management of the stack.

EXAMPLES:

DEL; <<DEL instruction>>

DELB; <<DELB instruction>>

DDEL; <<DDEL instruction>>

DO STATEMENT

The purpose of the DO statement is to repeatedly execute a statement until a specified condition clause becomes true. The condition clause is evaluated and tested after each execution of the statement. When the condition becomes true, execution transfers to the statement following the DO statement.

Syntax

<DO statement> ::= DO <statement> UNTIL <cond clause>

Syntax References

<statement>	→	V,	STATEMENT TYPES
<cond clause>	→	V,	IF STATEMENT

Semantics

After the statement specified is executed the condition clause is evaluated and tested. If false, the statement is executed again; if true, control transfers to the next statement following the DO statement. Each time the loop statement is executed the condition clause is again checked.

The condition clause can consist of logical expressions and hardware branch words as described under "IF Statement," this section.

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;
```

FOR STATEMENT

The purpose of the FOR statement is 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 in SPL/3000 is very machine-dependent, because it makes use of loop control instructions which require special stack markers.

Syntax

<FOR statement> ::= <FOR clause> <statement>

<FOR clause> ::= FOR <integer simpvar identifier> :=
<T_{ilb} expr> <STEP clause>
UNTIL <T_{ilb} expr> DO |
FOR * <integer simpvar identifier> :=
<T_{ilb} expr> <STEP clause>
UNTIL <T_{ilb} expr> DO

<STEP clause> ::= <empty> |
† step = 1
<T_{ilb} expr>

Syntax References

<statement>	→	V,	STATEMENT TYPES
<expr>	→	IV,	EXPRESSION TYPES
<simpvar identifier>	→	III,	SIMPLE VARIABLE DECLARATION

Semantics

The initial value, step value, and final value are all integer values which are calculated once upon entry into the FOR. The initial value is stored in the integer variable (FOR A :=) and tested before the loop statement is executed. After each execution of the loop statement, the variable is changed by the step value and compared with the final value. If the loop variable is less than or equal to the final value, the loop statement is executed again. If the loop variable is greater than the final value, control transfers to the next statement following the FOR statement.

There are two variations allowed in FOR: the STEP clause can be omitted, in which case the step is implicitly 1; and an * after FOR can be used to specify that the loop statement is to be executed once before incrementing and testing the variable. This guarantees that the loop statement is executed at least once even if the initial test should fail.

If the loop variable is equivalenced to the index register, the TBX and MTBX instructions are used for loop control. If the loop variable is a simple variable, the TBA and MTBA instructions are used. Since all of these instructions use a series of values placed in the stack, unpredictable results may occur if the user modifies the stack during the loop statement. If the index register is used as the loop variable, any operations within the loop statement which change the index register (such as array referencing) can destroy the loop control.

EXAMPLES:

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

GO STATEMENT

The purpose of a GO statement is to transfer control unconditionally to a labeled statement. The labeled statement can be specified by a label identifier or an indexed switch identifier (selects one of a set of labels).

Syntax

<GO statement> ::= GO <label ref> |
GOTO <label ref> |
GO TO <label ref>

<label ref> ::= <label identifier> |
<switch identifier> (<sindex>) |
* <switch identifier> (<sindex>)

<sindex> ::= <T_{ilb} expr> | <T_{ilb} assignment statement>
† must result in one-word value

Syntax References

<label identifier>	→	III,	LABEL DECLARATION
		IV,	STATEMENT TYPES
<switch identifier>	→	III,	SWITCH DECLARATION
<aexp>	→	IV,	ARITHMETIC EXPRESSIONS
<lexp>	→	IV,	LOGICAL EXPRESSIONS
<assignment statement>	→	V,	ASSIGNMENT STATEMENT

Semantics

The object of a global GO statement must be global and the object of a local GO statement must be local. There are no branches into or out of procedures, but procedures can go to labels passed as parameters. Entry points cannot be the object of a GO statement. If a main program or procedure passes a label to a procedure as an actual parameter, the procedure can transfer to the label. For example,

```
BEGIN
     LABEL L1;
     PROCEDURE PRO2(A,B,C); <<DECLARATION OF PROC2>>
          VALUE A; INTEGER A,B; LABEL C;
          BEGIN
               .
               .
               .
               GO TO C; <<REFERENCE TO LABEL PARAMETER>>
          END; <<END OF PROC2 DECLARATION>>
     .
     .
     .
          <<MAIN PROGRAM>>
     PROC2(2,N,L1); <<CALL TO PROC2>>
     .
     .
     .
     L1: <<SPECIAL RETURN POINT>>
     .
     .
     .
END.
```

Switches are invoked using an indexed GO statement; the index is an integer value that specifies the label desired (labels in a switch declaration are numbered consecutively starting with 0; see "Switch Declaration," Section III). Normally, if the index value is less than zero or greater than the number of labels minus one, control transfers to the statement following the GO statement. However, if the asterisk (*) is specified, no bounds checking is performed and invalid indices will cause unpredictable results:

```
SWITCH SW := L1, L2, L3;
GO TO SW(N);       <<bounds checking>>
GO TO * SW(N);     <<no bounds checking>>
```

IF STATEMENT

The IF statement chooses which of two statements to execute based on whether a condition clause is true or false. The condition clause can consist of logical expressions and/or hardware-defined branch words (overflow, carry, condition code, etc.). Indefinite nesting of IF statements is allowed.

Syntax

```
<IF statement>  ::=  IF <cond clause> <THEN part> <ELSE part>

<cond clause>   ::=  <cond elem> |
                          † lowest precedence
                     <cond clause> OR <cond elem>

<cond elem>     ::=  <cond term> |
                     <cond elem> AND <cond term>

<cond term>     ::=  <cond primary> |
                     (<cond factor>)

<cond factor>   ::=  <cond primary> OR <cond factor> |
                     <cond primary>
                          † parens override precedence of AND but cannot be nested

<cond primary>  ::=  <branch word> |
                          † hardware test
                     <lexp>
                          † logical expression

<THEN part>     ::=  THEN <statement>
                          † true alternative; can be empty but cannot be null statement
                          (i.e. no semi-colon).
<ELSE part>     ::=  <empty> |
                          † no false alternative
                     ELSE <statement>
                          † false alternative

<branch word>   ::=  CARRY | NOCARRY |
                     OVERFLOW | NOVERFLOW |
                     IABZ | DABZ |
                     IXBZ | DXBZ |
                     <relop>

<relop>         ::=  = |
                          † condition code equals 2
                     < > |
                          † condition code equals 0 or 1
                     < |
                          † condition code equals 1
                     > |
                          † condition code equals 0
                     < = |
                          † condition code equals 1 or 2
                     > =
                          † condition code equals 0 or 2
```

Syntax References

<lexp> → IV, LOGICAL EXPRESSIONS

<statement> → V, STATEMENT TYPES

Semantics

The IF statement allows the programmer to select one of two statements for execution based on a condition clause (see below). There are two formats of the IF statement: Format 1 without an ELSE part, and Format 2 with an ELSE part.

FORMAT 1

In this case, control is transferred to the statement following the THEN if the condition is true; if the condition is false, control falls through to the next statement following the IF statement. For example,

```
IF A < B THEN NX := A + B;
IF NO (FINAL LOR SUSPICIOUS) THEN
      BEGIN
            TEST' DONE := FALSE;
            GO TO AGAIN
      END;
```

FORMAT 2

In this case, there are two alternative statements within the IF statement. If the condition is true, control transfers to the statement following THEN; if the condition is false, control transfers to the statement following ELSE. When the statement selected is complete, control transfers to the next statement following the IF statement. For example,

```
IF A < B THEN XA := XA + A
            ELSE XA := XA + B;
IF TESTVAR THEN Y := Y + 1
      ELSE IF EXTRATEST THEN Y := Y - 1;
<<INVALID>>
IF TEST THEN A := A + B; ELSE A : A - B;
      <<NO SEMICOLON>>
```

NESTING

IF statements can be indefinitely nested (i.e., the alternative statements of an IF can themselves be IF statements). The innermost THEN is paired with the closest following ELSE and pair proceeds outward. For example,

```
IF <cond clause>
  ⎧THEN
  ⎪     IF <cond clause>
  ⎪         ⎧THEN
  ⎪         ⎪     IF <cond clause>
  ⎨         ⎨         ⎧THEN <statement>
  ⎪         ⎪         ⎩ELSE <statement>
  ⎪         ⎩ELSE <statement> ;
  ⎩<<NO OUTERMOST ELSE>>
```

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

Logical expressions and/or branch words can be combined using two special branch operators: OR and AND. If two items are combined with OR, the result is true if either or both is true; if two items are combined with AND, the result is true only if both are true. AND has precedence over OR, but this can be overriden by putting OR'ed sequences in parentheses. For example,

```
A < B OR DXBZ
└─┬─┘    └─┬┘
  └────────┘

A < B OR DXBZ AND CARRY
└─┬─┘    └─┬┘     └─┬─┘
  │        └────────┘
  └──────────┘

(A < B OR DXBZ) AND CARRY
 └─┬─┘    └─┬┘      └─┬─┘
   └────────┘         │
        └─────────────┘
```

CONDITION CLAUSE

The <cond clause> which is used to select the desired statement in IF is also used in the DO-UNTIL and WHILE -DO statements as well as the IF expression (see "Expression Type," Section IV). It is composed of two types of items: logical expressions and hardware branch words. Logical expressions (see "Logical Expressions," Section IV) result in a value of true or false. Branch words are hardware dependent branch conditions which are also either true or false:

Branch Word	**True Condition**
CARRY	Carry bit on
NOCARRY	Carry bit off
OVERFLOW	Overflow bit on
NOVERFLOW	Overflow bit off

Branch	True Condition
IABZ	Increment TOS(S-0). True if then zero.
DABZ	Decrement TOS(S-0). True if then zero.
IXBZ	Increment index register. True if then zero.
DXBZ	Decrement index register. True if then zero.
<	Condition Code equals 1
=	Condition Code equals 2
< =	Condition Code equals 1 or 2
>	Condition Code equals 0
< >	Condition Code equals 0 or 1
> =	Condition Code equals 0 or 2

OR and AND use branch instructions such as BCC, BOV, BNOV, BCY, BNCY, BRO, BRE, IABZ, IXBZ, DABZ, and DXBZ; they never generate arithmetic ands and ors. All parts of the condition clause may not be executed every time, since OR and AND branch out of the condition as soon as the truth value of the condition is determined (e.g., if a series of items is ANDed together, only one need be false for the total to be false).

EXAMPLES:

```
IF A > B THEN C := A;
IF A > B THEN C := A ELSE C := B;
IF B LOR C LAND NOT D THEN
    GO TO START ELSE
    GO TO FINISH;
IF OVERFLOW AND (N = 0) THEN
    BEGIN
        NO := 255;
        GO TO RESTART;
    END
ELSE
    BEGIN
        N := N - 1;
        GO TO START;
    END.
```

MOVE STATEMENT

The purpose of the MOVE statement is to move words or bytes from one location to another. The locations are specified by means of DB- or PB-relative addresses. There are three types of move operations, corresponding to three move-group instructions:

- Move words (MOVE instructions).
- Move bytes (MVB instruction).
- Move bytes while alphabetic and/or numeric, with or without upshifting of lowercase letters (MVBW).

In any move operation, the original contents of the MOVE source are unchanged.

Syntax

Computer Museum

<MOVE statement>	::=	<MOVE stmt> <sdeca> | <MOVE-WHILE stmt> <sdeca>
<MOVE statement>	::=	MOVE <T_{irlde} dest ref> := <T_{irlde} pointarr> <count> | † move words MOVE <T_{irlde} pointarr> := * <sadmode> <count> | † stacked source address MOVE <T_{irlde} pointarr> := <string> | MOVE <T_{irlde} pointarr> := (<listelmt>) | MOVE <byte ref> := <byte pointarr> <count> | † move bytes MOVE <byte ref> := * <sadmode> <count> | †stacked source address MOVE <byte ref> := <string> | MOVE <byte ref> := (<listelmt>)
<T_{irlde} dest ref>	::=	* | † stacked destination address <T_{irlde} pointarr>
<MOVE-WHILE stmt>	::=	MOVE <byte ref> := <byte ref> WHILE <cc>
<byte ref>	::=	<byte pointarr> | * † stacked byte address
<T pointarr>	::=	<T pointer identifier> <index> | <T array identifier> <index>
<count>	::=	, (<integer expr>)
<sdeca>	::=	<empty> | † delete all values , <sdec>
<sdec>	::=	0 | 1 | 2 † stack decrement

```
<sadmode>          ::= <empty> |
                          † DB stacked address
                      PB
                          † PB stacked address
<ccf>              ::= A |
                          † ALPHA only — condition code field
                      N |
                          † NUMERIC only
                      AS |
                          † ALSPH only, upshift
                      AN |
                          † ALPHA or NUMERIC
                      ANS
                          † ALPHA or NUMERIC; upshift
<listelmt>         ::= <initial value> |
                      <decimal integer> (<initial value list>) |
                          † only one level of repeat factors allowed
                      <listelmt list>
<initial value>    ::= <constant>
                          † truncated on left to 8 bits for byte move
```

Syntax References

<string>	→	II,	STRING CONSTANTS
<pointer>	→	III,	POINTER DECLARATION
<array>	→	III,	ARRAY DECLARATION
<expr>	→	IV,	EXPRESSION TYPES
<decimal integer>	→	II,	INTEGER CONSTANTS
<constant>	→	II,	CONSTANT TYPES
<index>	→	IV,	VARIABLES

Semantics

The move statements in SPL/3000 are very machine-dependent because they are based upon specific machine instructions.

SOURCE AND DESTINATION

The first reference after the MOVE is the destination address; the address, constant, or * after the := is the source address. Move words use integer, real, long, double, or long arrays or pointer references as source and destination. Move bytes use only byte array or pointer references. When the source is a string or a list of constants, the constants are generated into the code stream and moved from there. The list of constants (<listemt>) is the same as described under "Array Declaration," Section III.

Where * or *PB appears in place of an address, the DB of PB address must have been previously loaded onto the stack by the user. The source in move words or bytes (but not move bytes while) can also be a PB-relative address specified either by a local P-relative array reference or a stacked PB address (*PB). If both addresses are stacked, a byte MOVE is always assumed.

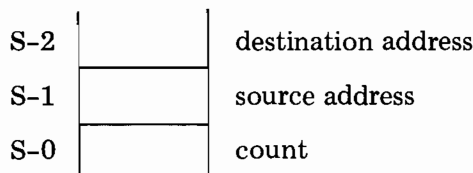
COUNT

The count value 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 list character moved.

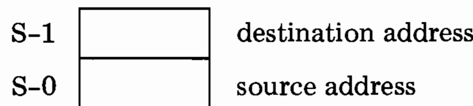
SDEC

The sdec value, or stack decrement operand, is an integer constant of value 0, 1, or 2 (or empty) which specifies how many of the temporary values required by move should be deleted from the stack after the move. An empty sdec restores the stack to its setting before the move statement.

The stacked values used by move words and bytes are as follows:



The stacked values for move bytes while are as follows:



In move bytes while, the <ccf> operand is the condition code field; it specifies the conditions for continuing the move to the next character:

A	current character is alphabetic.
N	current character is numeric.
AS	current character is alphabetic; upshift if lowercase.
AN	current character is alphabetic or numeric.
ANS	current character is alphabetic or numeric; upshift if lowercase.

EXAMPLES:

```
BYTE ARRAY BAR(0:20);
ARRAY DATA (0:100),DATAC(0:10);
MOVE DATA(4) := DATA(4),(4)2;       <<moves 4 words, leaves destination addresses on stack>>
MOVE * := DATAC(3),(6);             <<moves 6 words from DATAC into DATA starting
                                      where previous move left off>>
MOVE DATA(0) := (5(0,1,2,3,4),10,20,99);
MOVE BAR := "ABCDEFGHIJKLMNOPQRST",1;
DEST := TOS;
```

PROCEDURE CALL STATEMENT

The purpose of a PROCEDURE CALL statement is to invoke a previously-defined procedure and pass to it a list of actual parameters (addresses or values). When the procedure completes, execution normally returns to the next statement following the call (unless the procedure itself overrides the return).

Syntax

<PROCEDURE call statement>	::=	<procedure identifier> <actual param part>
<actual param part>	::=	<empty> | † no parameters (<actual param list>) | † no stacked params (<stacked param list>) | † all stacked params (<stacked param list> , <actual param list>) † stacked params must come first
<actual param>	::=	<reference param> | † passes an address <value param> | † passes a value <empty> † missing parameter; option VARIABLE only
<stacked param>	::=	* † address or value is already stacked by user
<reference param>	::=	<T simpvar identifier> | <T array identifier> <index> | <T pointer identifier> <index> | <procedure identifier> | <entry identifier> | † formal parameter must be procedure <label identifier>
<value param>	::=	<aexp> | <lexp> | <assignment statement>

Syntax References

<simpvar>	→	III,	SIMPLE VARIABLE DECLARATION
<pointer>	→	III,	POINTER DECLARATION
<array>	→	III,	ARRAY DECLARATION
<procedure>	→	III,	PROCEDURE DECLARATION
<label>	→	III,	LABEL DECLARATION
<aexp>	→	IV,	ARITHMETIC EXPRESSIONS
<lexp>	→	IV,	LOGICAL EXPRESSIONS
<assignment statement>	→	V,	ASSIGNMENT STATEMENT

Semantics

The procedure call statement generates a PCAL instruction to the procedure body associated with the specified <procedure identifier>.

The parameter list contains the list of actual parameters to be passed to the procedure. These must match the formal parameters, as declared in the procedure, in a one-to-one correspondence. The legitmate actual parameters for each formal parameter are listed under "Procedure Declaration," Section III.

Stacked parameters (specified by *) are parameters for which the user has already loaded the address or value onto the stack. If any parameter is stacked, all parameters to the left must be stacked too. Labels cannot be stacked. In addition, if the procedure is a function, the user must push a one, two, or three word zero onto the stack prior to the parameters for the return value. When stacked parameters are not used, the compiler generates this automatically. After the procedure returns, the function value space is deleted. For example,

```
PROC2 (*, *, R2);                <<2 stacked params>>
ASSEMBLE (ZERO; LOAD R1);<<push zero for integer function>>
PROC3 (*, R1, R2);               <<one stacked param; call to function procedure>>
```

If the procedure is declared with option VARIABLE, parameters can be omitted from the list by leaving a comma to hold their place. See "Procedure Declaration," Section III. For example,

```
PROCEDURE P(A,B,C,D,E,F); . . . . ;
     OPTION VARIABLE . . . . . . ;
P(R , , , R2);                   <<B,C,E,F missing>>
P(R);                            <<B,C,D,E,F missing>>
```

If the actual procedure identifier is itself a formal parameter (i.e., a procedure called from within a procedure), no parameter checking is performed, and the actual parameters are treated as if the corresponding formal parameters were all called by reference.

The procedure returns to the point of call when it reaches the final END of the procedure. Additional returns can be included in a procedure with the RETURN statement (see "RETURN Statement," this section). In addition, the procedure can return to a point other than the normal return by using GOTO with a label passed to it is a parameter (see "GO Statement," this section).

EXAMPLE:

COMPUTE (23.0, L2, PROC5);

PUSH AND SET STATEMENTS

The purpose of the PUSH statement is to push the contents of any or all of the registers onto the stack (PSHR instruction). The purpose of the SET statement is to set the contents of any or all of the registers using values taken from the top of stack (SETR instruction).

Syntax

<PUSH and SET statement>	::=	PUSH (<register spec list>) | SET (<register spec list>)
<register spec>	::=	S | Q | X | STATUS | Z | DL | DB † privileged mode required to set DB, DL, STATUS, or Z

Semantics

These two statements generate PSHR and SETR instructions for the registers specified. In a SETR the programmer must have previously loaded the values. In a PSHR the values are left on the top of the stack. The instructions operate as follows:

PSHR

If more than one register is specified, they are stacked in the order shown below (e.g., if all were stacked, DB would be in S-0, DL-DB in S-1, etc.).

Register Specified	Value Stacked
S	S-DB (relative S before PSHR)
Q	Q-DB (relative Q)
X	Index register
STATUS	Status register
Z	Z-DB (relative DB)
DL	DL-DB (relative DL)
DB	DB (absolute address)

SETR

The appropriate values must be loaded onto the stack before executing SETR. If more than one register is specified, they are set in the order shown below (DB first, S last). After the SETR instruction, the values have been deleted from the stack. SETR requires privileged mode except to set the index register, Q, S, and bits 4 through 7 and 2 of the status register (user traps, overflow, carry, and condition code).

Register Specified	Value taken from Stack
DB	DB (absolute address)
DL	DB-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 for DB before setting the registers.

EXAMPLES:

```
PUSH(DB,STATUS,Q);
      TDB := TOS;
      TSTATUS := TOS;
      TQ := TOS;
.
.
.
      TOS := TDB;
      TOS := TSTATUS;
      TOS := TQ;
SET(DB,STATUS,Q);
```

RETURN STATEMENT

The purpose of a RETURN statement is to generate additional exit points within a procedure or subroutine body. The final END of a procedure or subroutine declaration also generates an exit, but only the RETURN statement allows the programmer to leave some or all of the parameters on the stack after exiting back to the point of call.

Syntax

```
<RETURN statement>  ::=  RETURN <pcount>

<pcount>            ::=  <unsigned integer> |
                               † number of words to delete on exit
                         <empty>
                               † delete all parameters
```

Syntax References

<unsigned integer> → II, INTEGER CONSTANTS

Semantics

A RETURN within a procedure generates an EXIT instruction; a RETURN within a subroutine generates an SXIT. Multiple RETURNs within a single subroutine or procedure are allowed.

If the count in a RETURN is omitted, all parameters are deleted from the stack. If the count equals *N*, then only the top *N* parameter words are deleted after exiting. If *N* equals zero, all parameters are left on the stack.

The calling program must know how many parameters will be left on return from the procedure or subroutine, because it must take care of them (examine them, save them, etc.). Integer, logical, and byte variables by value use one word; double and real use two words; and long variables and labels use three. All other parameters use one word.

EXAMPLES:

```
PROCEDURE P(A,B); VALUE A; DOUBLE A; INTEGER B;
   BEGIN
      .
      .
      .
      RETURN 1;     <<EXIT 1; LEAVE A>>
      .
      .
      RETURN;       <<EXIT 3; DELETE BOTH>>
      .
      .
   END;
```

SCAN STATEMENT

The purpose of a SCAN statement is 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. This is the one SPL/3000 statement that cannot be used properly without explicitly accessing the stack. 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).

Syntax

<SCAN statement>	::=	<SCAN-WHILE stmt> <sdeca> | <SCAN-UNTIL stmt> <sdeca>
<SCAN-WHILE stmt>	::=	SCAN <byte ref> WHILE <testword>
<byte ref>	::=	<byte pointer identifier> <index> | † no PB arrays <byte array identifier> <index> | * † stacked byte address
<testword>	::=	<T_{il} simpvar identifier> | <integer> | "<char> <char>" | † terminal character — test character * † stacked testword
<SCAN-UNTIL stmt>	::=	SCAN <byte ref> UNTIL <testword> † no PB arrays
<sdeca>	::=	<empty> | † delete all values , <sdec>
<sdec>	::=	0 | 1 | 2
<char>	::=	{any member of the ASCII character set; " is represented by " "}

Syntax References

<pointer>	→	III,	POINTER DECLARATION
<array>	→	III,	ARRAY DECLARATION
<simpvar>	→	III,	SIMPLE VARIABLE DECLARATION
<integer>	→	II,	INTEGER CONSTANTS
<index>	→	IV,	VARIABLES

Semantics

The scan statements in SPL/3000 are very machine-dependent because they are based on specific machine instructions.

BYTE REFERENCE

The byte reference (which specifies where to start scanning) can be a byte array reference, a byte pointer reference, or an asterisk (*) if the DB-relative address already is stacked by the user. Local P-relative arrays cannot be scanned. The address of the byte reference is loaded onto the stack.

TESTWORD

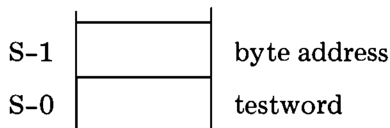
The testword is an integer logical simple variable, an integer constant, or a two character string where the first character (bits 0 to 7) specifies the terminal character and the second character (bits 8 through 15) specifies the test character. In both scans each byte is tested against the test and terminal character.

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 is set after a scan to indicate whether the scan terminated because of the test character (carry = 0) or the terminal character (carry = 1). This can be tested with an IF statement:

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

SDEC

Sdec specifies how many words to delete from the stack after the scan. The stack decrement factor is very important in scan because when the scan terminates, the address of the terminating byte is left in the stack. The stack for a SCU or SCW instruction appears as follows:



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

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

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

EXAMPLES:

```
SCAN INPUT UNTIL ".$",1;
IF CARRY THEN GOTO FINAL;
SCAN * UNTIL ".X",1;
ADR := TOS;
```

SUBROUTINE CALL STATEMENT

The purpose of a SUBROUTINE CALL statement is to invoke a previously-defined subroutine and pass to it a list of parameters (addresses or values). When the subroutine completes, execution normally returns to the next statement following the SUBROUTINE CALL (unless the subroutine itself overrides the return).

Syntax

```
<SUBROUTINE call statement>  ::=  <subroutine identifier> <actual param part>

<actual param part>          ::=  <empty> |
                                        † no parameters
                                  (<actual param list>) |
                                        † no stacked parameters
                                  (<stacked param list>) |
                                        † all stacked parameters
                                  (<stacked param list> , <actual param list>)
                                        † stacked parameters must come first

<actual param>               ::=  <reference param> |
                                        † passes an address
                                  <value param> |
                                        † passes a value
                                  <empty>
                                        † missing parameter; option VARIABLE only

<stacked param>              ::=  *
                                        † address or value is already stacked by user

<reference param>            ::=  <T simpvar identifier> |
                                  <T array identifier> <index> |
                                  <T pointer identifier> <index> |
                                  <T procedure identifier>

<value param>                ::=  <aexp> |
                                  <lexp> |
                                  <assignment statement>
```

Syntax References

<subroutine>	→	III,	SUBROUTINE DECLARATION
<simpvar>	→	III,	SIMPVAR DECLARATION
<pointer>	→	III,	POINTER DECLARATION
<array>	→	III,	ARRAY DECLARATION
<index>	→	IV,	VARIABLES
<procedure>	→	III,	PROCEDURE DECLARATION
<label>	→	III,	LABEL DECLARATION
<aexp>	→	IV,	ARITHMETIC EXPRESSIONS
<lexp>	→	IV,	LOGICAL EXPRESSIONS
<assignment statement>	→	V,	ASSIGNMENT STATEMENT

Semantics

The SUBROUTINE CALL statement generates an SCAL instruction to the subroutine specified. In a main program only global subroutines can be called; within a procedure only subroutines local to that procedure can be called.

The conventions for parameters are exactly the same as described under "Procedure Declaration," Section II, and "Procedure Call Statement," this section, except that subroutines cannot have a variable number of parameters and labels cannot be passed to subroutines.

EXAMPLES:

```
SUBROUTINE S(A,B,C);
    INTEGER A,B,C;
    BEGIN
        .
        .
        .
    END;
S(K,R5,TESTVAL);
```

WHILE STATEMENT

The purpose of the WHILE statement is to repeatedly execute a statement as long as a specified condition clause is true. The condition clause is evaluated and tested before executing the statement. When the condition becomes false, execution transfers to the statement following the WHILE statement.

Syntax

<WHILE statement> ::= WHILE <cond clause> DO <statement>

Syntax References

<statement> → V, STATEMENT TYPES

<cond clause> → V, IF STATEMENT

Semantics

The condition clause is always tested before executing the loop statement. When the condition is false, control transfers to the statement following the WHILE statement.

The condition clause can consist of logical expressions and hardware branch words as described under "IF Statement," in this section.

However, the following exceptions hold for the WHILE statement.

<branch lexp>	Action
IABZ	Increment TOS. Execute <statement> if TOS is non-zero.
DABZ	Decrement TOS. Execute <statement> if TOS is non-zero.
IXBZ	Increment X reg. Execute <statement> if X register is non-zero.
DXBZ	Decrement X reg. Execute <statement> if X register is non-zero.

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;
```

APPENDIX A
ASCII Character Set

Graphic	Decimal Value	Octal Value	Comments
	0	0	Null
	1	1	Start of heading
	2	2	Start of text
	3	3	End of text
	4	4	End of transmission
	5	5	Enquiry
	6	6	Acknowledge
	7	7	Bell
	8	10	Backspace
	9	11	Horizontal tabulation
	10	12	Line feed
	11	13	Vertical tabulation
	12	14	Form feed
	13	15	Carriage return
	14	16	Shift out
	15	17	Shift in
	16	20	Data link escape
	17	21	Device control 1
	18	22	Device control 2
	19	23	Device control 3
	20	24	Device control 4
	21	25	Negative acknowledge
	22	26	Synchronous idle
	23	27	End of transmission block
	24	30	Cancel
	25	31	End of medium
	26	32	Substitute
	27	33	Escape
	28	34	File separator
	29	35	Group separator
	30	36	Record separator
	31	37	Unit separator
	32	40	Space
!	33	41	Exclamation point
"	34	42	Quotation mark
#	35	43	Number sign
$	36	44	Dollar sign

Graphic	Decimal Value	Octal Value	Comments
%	37	45	Percent sign
&	38	46	Ampersand
'	39	47	Apostrophe
(	40	50	Opening parenthesis
)	41	51	Closing parenthesis
*	42	52	Asterisk
+	43	53	Plus
,	44	54	Comma
–	45	55	Hyphen (Minus)
.	46	56	Period (Decimal)
/	47	57	Slant
0	48	60	Zero
1	49	61	One
2	50	62	Two
3	51	63	Three
4	52	64	Four
5	53	65	Five
6	54	66	Six
7	55	67	Seven
8	56	70	Eight
9	57	71	Nine
:	58	72	Colon
;	59	73	Semi-colon
<	60	74	Less than
=	61	75	Equals
>	62	76	Greater than
?	63	77	Question mark
@	64	100	Commercial at
A	65	101	Uppercase A
B	66	102	Uppercase B
C	67	103	Uppercase C
D	68	104	Uppercase D
E	69	105	Uppercase E
F	70	106	Uppercase F
G	71	107	Uppercase G
H	72	110	Uppercase H
I	73	111	Uppercase I
J	74	112	Uppercase J
K	75	113	Uppercase K
L	76	114	Uppercase L
M	77	115	Uppercase M
N	78	116	Uppercase N
O	79	117	Uppercase O
P	80	120	Uppercase P
Q	81	121	Uppercase Q
R	82	122	Uppercase R
S	83	123	Uppercase S
T	84	124	Uppercase T
U	85	125	Uppercase U
V	86	126	Uppercase V
W	87	127	Uppercase W

Graphic	Decimal Value	Octal Value	Comments
X	88	130	Uppercase X
Y	89	131	Uppercase Y
Z	90	132	Uppercase Z
[	91	133	Opening bracket
\	92	134	Reverse slant
]	93	135	Closing bracket
^	94	136	Circumflex
___	95	137	Underscore
`	96	140	Grave accent
a	97	141	Lowercase a
b	98	142	Lowercase b
c	99	143	Lowercase c
d	100	144	Lowercase d
e	101	145	Lowercase e
f	102	146	Lowercase f
g	103	147	Lowercase g
h	104	150	Lowercase h
i	105	151	Lowercase i
j	106	151	Lowercase j
k	107	152	Lowercase k
l	108	154	Lowercase l
m	109	155	Lowercase m
n	110	156	Lowercase n
o	111	157	Lowercase o
p	112	160	Lowercase p
q	113	161	Lowercase q
r	114	162	Lowercase r
s	115	163	Lowercase s
t	116	164	Lowercase t
u	117	165	Lowercase u
v	118	166	Lowercase v
w	119	167	Lowercase w
x	120	170	Lowercase x
y	121	171	Lowercase y
z	122	172	Lowercase z
{	123	173	Opening (left) brace
|	124	174	Vertical line
}	125	175	Closing (right) brace
~	126	177	Tilde
	127	177	Delete

APPENDIX B
Reserved Words

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

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

APPENDIX C
Compiler Commands

The compiler subsystem commands are entered from the *textfile* and *masterfile* (see "MPE/3000 Commands") and take effect only after the compiler has been accessed. In the description of these commands that follows, square brackets are used to enclose optional items, braces are used to enclose required items (one of which must be chosen), and ellipses (. . .) indicate repeated items.

The basic syntax of subsystem commands is

$command [*parameter-list*]

$$*command* [*parameter-list*]

The $ must be in column one, immediately followed by the command name, which must be completely spelled out. The parameter list, separated from the name by at least one blank, is optional; some commands have parameters, other do not. Parameters are separated from each other by commas. Blanks may be freely inserted between items in the list. Commands with $$ are not sent to *newfile.*

A command is continued to the next record if the last nonblank character is an ampersand (&). In this case, the following record must begin with a $. The effect is to combine the characters preceding with & with those following the $ of the next record with a blank inserted between them. Therefore, command names and other elements cannot be broken by an &. Columns 73 to 80 of each record (source or command) can be used for sequence numbers (see the EDIT command).

PAGE COMMAND

$PAGE [*"character-string" list*]

The PAGE command ejects the current page of *listfile* to the top of the next page, prints the optional *character-string,* and skips two more lines. Listing continues at that point. The strings must be enclosed in quotes (which are deleted) and separated by commas. PAGE is effective only if the LIST option (see CONTROL command) is on. The PAGE command itself is not sent to the *listfile.* The *character-string* replaces that specified by a previous TITLE command, unless it is null, in which case the preceding TITLE (if any) is printed after paging. This command is sent to *newfile.*

EXAMPLE:

$PAGE "MIDDLE OF PROGRAM COMPILE"

$PAGE

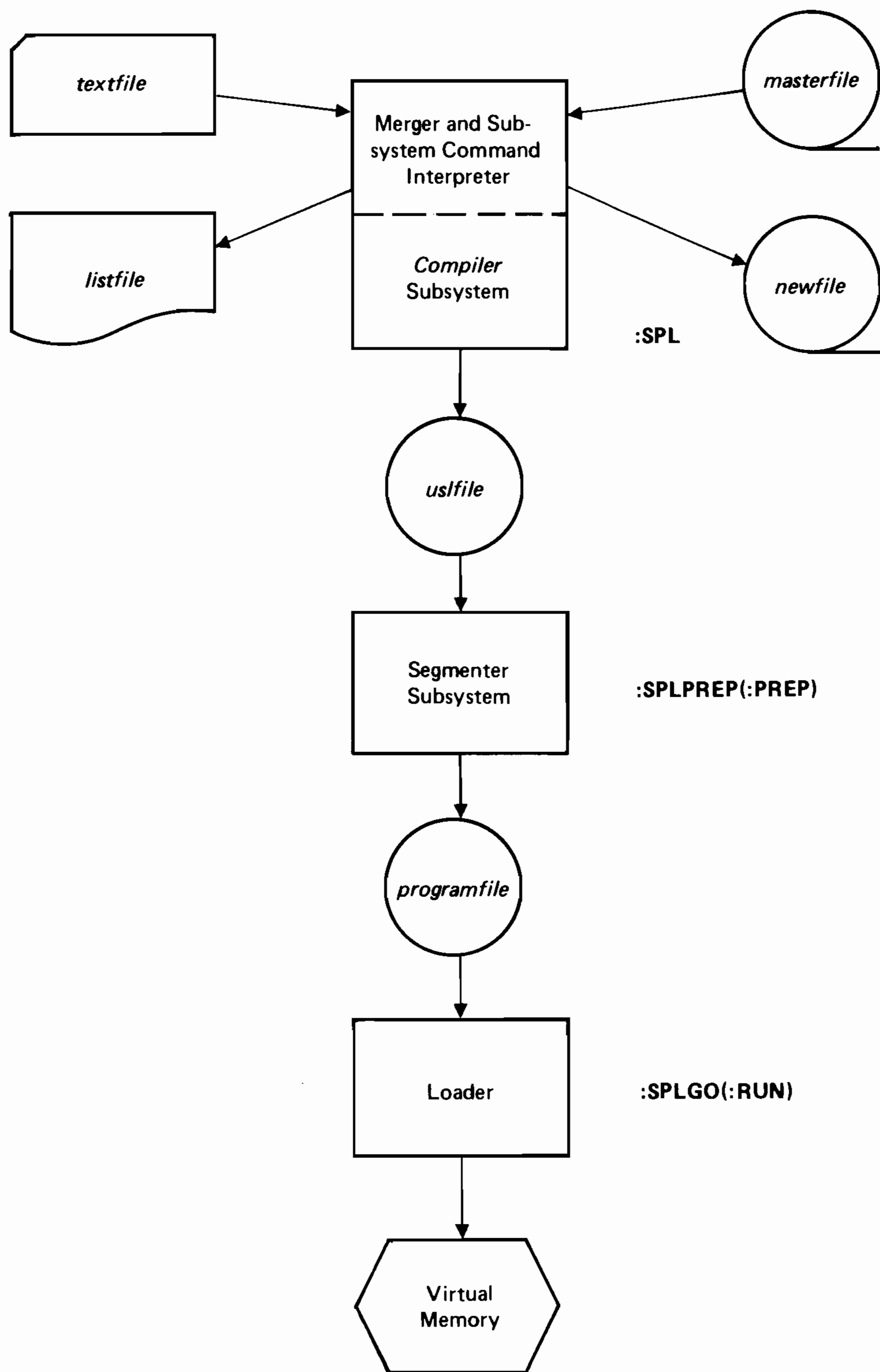


Figure C-1. Subsystem File Relationships

TITLE COMMAND

$TITLE ["*character-string*" *list*]

The *character-string* is stripped of quotes and saved. It is used whenever a page heading is printed (unless overridden by a PAGE command or a new TITLE command). If *character-string* is not specified, the title is set to the null string (i.e., no title from then on). A quote mark is specified in a string by a pair of quotes. This command is sent to *listfile* and *newfile*.

EXAMPLE:

```
$TITLE "ACCOUNTING PROGRAM 7/7/73"
$TITLE "FINAL COMPILATION",&
$" " "ACC-PROG" " "
```

CONTROL COMMAND

$CONTROL [*parameter-list*]

The possible parameters of the CONTROL command are:

LIST	Sends each source record (after editing, see EDIT command) to *listfile*. (Activates MAP and CODE, if they were suppressed by NOLIST.)
NOLIST	Sends only offending source records and their error messages to *listfile;* also suppresses CODE, ADR, INNERLIST, and MAP.
WARN	Sends warning messages to *listfile* with offending source records.
NOWARN	Does not send warning messages to *listfile*.
MAP	Prints a symbol table of all identifiers on *listfile* at the end of the compilation listing or at the end of procedures and subroutines in which they are local.
NOMAP	Turns off MAP.
ERRORS=*nnn*	Sets the maximum number of severe errors to *nnn*. If this number is exceeded during compilation, the compiler terminates ($0 < nnn <= 999$).
CODE	Sends blocks of final instructions (in octal) to the *listfile* after appropriate sections of the compilation listing.
NOCODE	Does not send records of machine code to *listfile*.

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, and may contain apostrophes (').
USLINIT	Initializes the USL file by deleting all existing content.
LINES=*nnnn*	Sets the number of lines per page on the *listfile* to *nnnn*. (0 < *nnnn* < = 9999).

The following five parameters are unique to SPL/3000:

ADR	After each declaration, send a record to the *listfile* (if LIST is in effect) showing the addressing mode and displacement of the variables declared. This is turned off by NOLIST.
INNERLIST	After each line of statement, sends an innerlist of unoptimized code (using mnemonics) emitted by the compiler to the *listfile* (if LIST is in effect). This is turned off by NOLIST.
MAIN=*program-name*	Assigns the specified name to the main program. Format for program names is the same as for segment names. Replaces the heading (columns 13-27 inclusive) starting with page 2.
UNCALLABLE	Makes the outer block entry point uncallable (i.e., can only be called by code running in privileged mode). This command must be at the beginning of the source file.
PRIVILEGED	Makes the code segment containing the outer block privileged. This command must be at the beginning of the program.

SUBPROGRAM [(*proc* [, *proc*] . . .)]

Where *proc* is a procedure name or a procedure name followed by an asterisk (*). This command places the compiler in subprogram mode and must occur at the beginning of a compilation.

If no parameters are specified, all of the procedures in the merged source program are compiled, but not the outer block or main program, if any.

If procedure parameters appear, only those procedures specified are compiled; all others are skipped. In addition, procedure names followed by an asterisk (*) are compiled with LIST, CODE, and MAP options on. Those without * are compiled, but not listed. The asterisk mechanism is overridden by explicit CONTROL commands specifying LIST, ADR, etc.

The default mode is program mode, not subprogram mode.

Even in subprogram mode, global declarations and OPTION FORWARD and 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 compile. The compiler also automatically CEASEs any existing procedures in the file with the same procedure name as one being currently compiled, except for INTERNAL procedures.

EXAMPLE:

$CONTROL SUBPROGRAM
Compiles all procedures.
$CONTROL SUBPROGRAM (PROC1, PROC2*)
Compiles PROC1 without listing and PROC2 with listing.

The default parameters of CONTROL for SPL/3000 are:

LIST
WARN
MAP off
ERRORS=100 (decimal)
NOCODE
SEGMENT=SEG'
MAIN=OB'
program mode, rather than subprogram mode
ADR off
INNERLIST off
LINES=60 (decimal)
USL file not initialized, unless new uslfile.
Callable, non-privileged outer block.

SET COMMAND

$SET [X*n* = ON / OFF [, X*n* = ON / OFF] . . .]

Where *n* is a digit from 0 through 9.

Sets toggle X*n* ON or OFF. If no parameters are specified, all ten toggles are set off. If more than one parameter is given, there must be separating commas. All toggles begin the compilation in the OFF state. The state of a toggle can be checked by an IF command and used to skip portions of the source file.

IF COMMAND

$IF [X*n* = ON / OFF]

Where *n* is a digit from 0 through 9.

If the specified condition is false (i.e., the toggle is not in the state specified), all succeeding source records are ignored by the compiler (except that they are listed and sent to *newfile*)

until another IF command. The only commands that are recognized during this skip are EDIT and IF. If the specified condition is true, succeeding source records are compiled normally. If no condition is specified, the IF command merely terminates the last previous IF command.

EDIT COMMAND

$EDIT *parameter* [, *parameter*] . . .

The EDIT command and file parameters of the :SPL command allow a wide range of editing capabilities: corrections can be merged with an old master program to produce a new program for compilation; sequence fields can be checked for ascending order; sections of the source program can be skipped; and sequence fields can be renumbered.

If both the *masterfile* and the *textfile* are specified, then they are merged and both are checked for ascending sequence fields. (Each sequence field in columns 73 to 80 must be greater than the previous one or all blanks.) In merginging, one record is read from each file; their sequence fields are compared. The one with the lower sequence field (according to the ASCII collating sequence) is compiled and passed to *newfile.* If they are the same, the record from *testfile* is taken. Portions of the *masterfile* can be skipped with the VOID parameter of EDIT.

If *newfile* is specified, merged output is sent to that file with sequence fields unchanged. EDIT must be used explicitly to renumber the *newfile.*

If *textfile* is specified and not *masterfile*, input is not sequence checked. However, if *textfile* = $NULL, then *masterfile* is the input file and is sequence checked.

All source records including commands are sent to *newfile*, except EDIT commands and those command images which have dollar signs in columns one and two ($$). EDIT commands can have sequence numbers, but EDIT continuation records must have blank sequence fields.

The parameters of EDIT are:

VOID = *sequence-number*

All records from the *masterfile* are skipped until one is read with a sequence field greater than *sequence-number. Sequence-number* can be specified in two ways:

= number
= "string"

Both forms are used right justified, but number is left-filled with ASCII zero (0) digits (i.e., %60), while string is stripped of its quotes and left-filled with blank characters. The first form is compatible with sequence fields as generated by SEQNUM (see below); the second with the sequence fields as generated by the *HP 3000 Text Editor.*

SEQNUM = *number*

The merged source records are renumbered starting with *number* assigned to the next source record. The *number* must be a legal number. The increment for each record (see INC) is reset to the default value (1000).

NOSEQ

Succeeding merged source records are not renumbered. This is the default condition.

INC = *number*

The sequence numbers of records sent to *newfile* are incremented by *number.* This parameter is ignored if *newfile* is $NULL or if there has not been a SEQNUM specified.

TRACE COMMAND

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

The TRACE command specifies which identifiers within the outer block (no program-unit name means the main program or outer block) or procedures of a program will be traced at run-time. The tracing is implemented through calls to the SYMBOL TRACE. This subsystem allows references to variables, arrays, pointers, labels, and procedures to be monitored with appropriate printout and breakpoints. For further details, see the *HP 3000 SYMBOL TRACE (03000-90015).*

APPENDIX D
MPE/3000 Subsystem Commands

User access to the compiler is provided by three MPE/3000 commands. The :SPL command compiles only; the :SPLPREP command compiles and prepares; and the :SPLGO command compiles, prepares, and executes.

In the command descriptions below, the parameters all specify files; the actual file designator specified is equated to the formal file designator that corresponds to that parameter. If no parameter is specified, a default actual file designator is assigned for the formal file designator. Brackets around a parameter or list of parameters indicate an optional item. Trailing commas must be omitted. The default actual file designators depend upon whether the context is a session or a batch job. Consult the *HP 3000 Multiprogramming Executive Operating System (03000-90005)* for further details.

SPL—COMPILE ONLY

:SPL [*textfile*] {, *uslfile*} {, *listfile*} {, *masterfile*} {, *newfile*}

:SPL compiles a source program from the *textfile*, with code generated to the *uslfile*, listing output on the *listfile*, and optional editing with *masterfile* and *newfile*. See the $EDIT compiler command for use of *masterfile* and *newfile*.

Parameter	Use	Formal File Designator	Default Actual Designator
textfile	Source program, corrections, compiler commands.	SPLTEXT	$STDIN
listfile	Output listing.	SPLLIST	$STDLIST
uslfile	Object code output	SPLUSL	$NEWPASS/$OLDPASS
masterfile	Old text for edit.	SPLMAST	$NULL
newfile	New text for edit.	SPLNEW	$NULL

The code generated by :SPL can be passed to the Segmenter for preparation with :PREP through the *uslfile*. Also the *uslfile* can be saved immediately following :SPL with the SAVE command referring to $OLDPASS.

EXAMPLE:

:SPL	Uses all default files, no editing.
:SPL PRQ27	Compiles from user textfile.
:SPL , , , PRQ27, PRQ28	Compiles from standard input, editing masterfile PRQ27 and storing the new program on newfile PRQ28.
:SAVE SCUSL = $OLDPASS	

SPLPREP—COMPILE AND PREPARE

:SPLPREP [*textfile*] {, *progfile*} {, *listfile*} {, *masterfile*} {, *newfile*}

:SPLPREP compiles from *textfile* into a temporary *uslfile* created by the compiler, with output listing on *listfile* and optional editing from *masterfile* to *newfile.* It then prepares the program from the *uslfile* to *profile.* Profile can then be executed using the :RUN command. The *uslfile* is still available after the command in $OLDPASS, unless the default is taken for *profile.*

All parameters except *progfile* are the same as :SPL.

Parameter	Use	Formal File Designator	Default Actual Designator
progfile	Destination for runnable program.	SPLPROG	$NEWPASS

EXAMPLE:

:SPLPREP	Uses all default files, no editing.
:SPLPREP SPRING, SUMMER	
:SPLPREP CORRECT , , , SPRING5,SPRING6	

SPLGO—COMPILE, PREPARE, AND EXECUTE

:SPLGO [*textfile*] {, *listfile*} {, *masterfile*} {, *newfile*}

:SPLGO compiles a source program from *textfile*, with output listing on *listfile*, and optional editing from *masterfile* to *newfile.* Both the *uslfile* used for object code and the *progfile* used for preparing and execution are temporary files created by the command. After the program runs, the *progfile* is still available in $OLDPASS, assuming the program itself did not open a $NEWPASS. All of the parameters are the same as described for :SPL.

EXAMPLE:

:SPLGO

:SPLGO TEXT2A, LP, OLDER, NEWER

APPENDIX E
HP 3000 Machine Instructions

This appendix consists of four parts:

Alphabetic listing of all HP 3000 machine instructions by mnemonic.

HP 3000 consolidated coding sheet.

Opcode formats for ASSEMBLE statement.

Functional cross-reference of all HP 3000 machine instructions.

ALPHABETICAL LISTING OF INSTRUCTIONS

Mnemonic	Function	Format
ADAX	Add A to X	2
ADBX	Add B to X	2
ADD	Add	2
ADDI	Add immediate	4a
ADDM	Add memory	1a
ADDS	Add to S	7
ADXA	Add X to A	2
ADXB	Add X to B	2
ADXI	Add immediate to X	4a
AND	And, logical	2
ANDI	Logical AND immediate	7
ASL	Arithmetic shift left	3b
ASR	Arithmetic shift right	3b
BCC	Branch on Condition Code	1d
BCY	Branch on carry	3a
BE	Branch on equals	
BG	Branch on greater than	
BGE	Branch on greater than or equal	See BCC
BL	Branch on less than	
BLE	Branch on less than or equal	
BN	Branch on next equal	
BANCY	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

Mnemonic	Function	Format
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
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
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
DDEL	Double delete	2
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
DIV	Divide	2
DIVI	Divide immediate	4a
DIVL	Divide Long	2
DLSL	Double logical shift left	3b
DLSR	Double logical shift right	3b
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
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
FNEC	Floating negate	2

Mnemonic	Function	Format
FSUB	Floating subtract	2
HALT	Halt	6
IABZ	Increment A, branch if zero	3a
INCA	Increment A	2
INCB	Increment B	2
INCM	Increment memory	1b
INCX	Increment index	2
IXBZ	Increment X, branch if zero	3a
LADD	Logical add	2
LCMP	Logical compare	2
LDB	Load byte	1b
LDD	Load double	1b
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
LRA	Load relative address	1a
LSL	Logical shift left	3b
LSR	Logical shift right	3b
LSUB	Logical subtract	2
MOVE	Move words	8a
MPY	Multiply	2
MPYI	Multiply immediate	4a
MPYL	Multiply Long	2
MPYM	Multiply memory	1a
MTBA	Modify, Test, Branch, A	1e
MTBX	Modify, Test, Branch, X	1e
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
OR	Or, logical	2
ORI	Logical OR immediate	7
PAUS	Pause	6
PCAL	Procedure call	7
PLDA	Privileged load from absolute address	5
PSHR	Push registers	4a
PSTA	Privileged store into absolute address	5
RIO	Read I/O	6

Mnemonic	Function	Format
RMSK	Read Mask	6
RSW	Read Switch register	5
SBXI	Subtract immediate from X	7
SCAL	Subroutine Call	7
SCAN	Scan bits	3b
SCU	Scan until	8c
SCW	Scan while	8c
SED	Set enable/disable external interrupts	6
SETR	Set registers	4a
SIN	Set interrup	6
SIO	Start I/O	6
SIRF	Set external interrupt reference flag	6
SMSK	Set Mask	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
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
WIO	Write I/O	6
XAX	Exchange A and X	2
XBX	Exchange B and X	2
XCH	Exchange A and B	2
XCHD	Exchange DB	6
XEQ	Execute	6
XOR	Exclusive or, logical	2
XORI	Logical Exclusive OR immediate	7
ZERO	Push zero	2
ZROB	Zero B	2
ZROX	Zero X	2

HP 3000 CONSOLIDATED CODING SHEET

0	1 2 3	4	5 6 7	8 9	10	11 12 13	14 15
00	STACK OPCODES ALL 64 STACK OPS MAY BE USED IN EITHER POSITION (STACK OP A OR B)		00	NOP		40	DEL
			01	DELB		41	ZROB
			02	DDEL		42	LDXB
			03	ZROX		43	STAX
			04	INCX		44	LDXA
			05	DECX		45	DUP
			06	ZERO		46	DDUP
			07	DZRO		47	FLT
			10	DCMP		50	FCMP
			11	DADD		51	FADD
			12	DSUB		52	FSUB
			13	MPYL		53	FMPY
			14	DIVL		54	FDIV
			15	DNEG		55	FNEG
			16	DXCH		56	CAB
			17	CMP		57	LCMP
			20	ADD		60	LADD
			21	SUB		61	LSUB
			22	MPY		62	LMPY
			23	DIV		63	LDIV
			24	NEG		64	NOT
			25	TEST		65	OR
			26	STBX		66	XOR
			27	DTST		67	AND
			30	DFLT		70	FIXR
			31	BTST		71	FIXT
			32	XCH		72	SPARE
			33	INCA		73	INCB
			34	DECA		74	DECB
			35	XAX		75	XBX
			36	ADAX		76	ADBX
			37	ADXA		77	ADXB
01	SUB OPCODE 1	X	00	ASL		SHIFT COUNT L	
		X	01	ASR		" " "	
		X	02	LSL		" " "	
		X	03	LSR		" " "	
		X	04	CSL		" " "	
		X	05	CSR		" " "	
		X	06	SCAN			
		I	07	IABZ	+/-	P RELATIVE DISPLACEMENT	
		X	10	TASL		SHIFT COUNT L	
		X	11	TASR		" " "	
		I	12	IXBZ	+/-	P RELATIVE DISPLACEMENT	
		I	13	DXBZ	+/-	" " "	
		I	14	BCY	+/-	" " "	
		I	15	BNCY	+/-	" " "	
		X	16	TNSL		SHIFT COUNT L	
		Y	17	SPARE			
		X	20	DASL		SHIFT COUNT L	
		X	21	DASR		" " "	
		X	22	DLSL		" " "	
		X	23	DLSR		" " "	
		X	24	DCSL		" " "	
		X	25	DCSR		" " "	
		I	26	CPRB	+/-	P RELATIVE DISPLACEMENT	
		I	27	DABZ	+/-	" " "	
		I	30	BOV	+/-	" " "	
		I	31	BNOV	+/-	" " "	
		X	32	TBC		BIT POSITION	
		X	33	TRBC		" "	
		X	34	TSBC		" "	
		X	35	TCBC		" "	
		I	36	BRO	+/-	P RELATIVE DISPLACEMENT	
		I	37	BRE	+/-	" " "	
02	SUB OPCODE 2	00	MOVE OPS	0 MOVE	PB/DB	RESERVED	SDEC
				1 MVB	"	"	"
				2 MVBL	0	"	"
				2 SCW	1	"	"
				3 MVLB	0	"	"
				3 SCU	1	"	"
				4 MVBW	CCG	CCE CCL	"
				5 CMPB	PB/DB	RESERVED	SDEC

0	1 2 3	4	5 6 7	8 9 10 11	12 13 14 15
02	SUB OPCODE 2	00	MINI OPS	14 RSW	RESERVED 0
				14 LLSH	RESERVED 1
				15 PLDA	RESERVED 0
				15 PSTA	RESERVED 1
		01	SPARE		
		02	LDI	IMMEDIATE OPERAND N	
		03	LDXI	" " "	
		04	CMPI	" " "	
		05	ADDI	" " "	
		06	SUBI	" " "	
		07	MPYI	" " "	
		10	DIVI	" " "	
		11	PSHR	DB DL Z	STA X Q S
		12	LDNI	IMMEDIATE OPERAND N	
		13	LDXN	" " "	
		14	CMPN	" " "	
		15	EXF	BEGINNING BIT #	# OF BITS
		16	DPF	" "	" " "
		17	SETR	DB DL Z	STA X Q S
03	SUB OPCODE 3	00	SPECIAL OP	00 SPARE	
				01 PAUS	K FIELD
				02 SED	" "
				03 XCHD	" "
				04 SMSK	" "
				05 RMSK	" "
				06 XEQ	" "
				07 SIO	" "
				10 RIO	" "
				11 WIO	" "
				12 TIO	" "
				13 CIO	" "
				14 CMD	" "
				15 SIRF	" "
				16 SIN	" "
				17 HALT	" "
		01	SCAL	STT ENTRY # N	
		02	PCAL	" " " "	
		03	EXIT	N FIELD	
		04	SXIT	" "	
		05	ADXI	" "	
		06	SBXI	" "	
		07	LLBL	PL- DISPLACEMENT N	
		10	LDPP	P+ DISPLACEMENT N	
		11	LDPN	P- DISPLACEMENT N	
		12	ADDS	IMMEDIATE OPERAND N	
		13	SUBS	" " "	
		14	TSBM	DB+ DISPLACEMENT N	
		15	ORI	IMMEDIATE OPERAND N	
		16	XORI	" " "	
		17	ANDI	" " "	

0	1 2 3	4	5	6	7	8 – 15
04	LOAD	X	I			PDQS ADDRESS MODE & DISPLACEMENT
05	TBA	0	0	0	+/-	P RELATIVE DISPLACEMENT
	MTBA	0	1	0	+/-	" " "
	TBX	1	0	0	+/-	" " "
	MTBX	1	1	0	+/-	" " "
	STOR	X	I	1		DQS ADDRESS MODE & DISPLACEMENT
06	CMPM	X	I			PDQS " " " "
07	ADDM	X	I			" " " " "
10	SUBM	X	I			" " " " "
11	MPYM	X	I			" " " " "
12	INCM	X	I	0		DQS " " " "
	DECM	X	I	1		" " " " "
13	LDX	X	I			PDQS " " " "
14	BR	X	I	0	+/-	P RELATIVE DISPLACEMENT
	BR	X	I	1		DQS ADDRESS MODE (INDIRECT) & DISPLACEMENT
	BCC	I	0	1		CCG CCE CCL +/- P RELATIVE DISPLACEMENT
15	LDB	X	I	0		DQS ADDRESS MODE & DISPLACEMENT
	LDD	X	I	1		" " " "
16	STB	X	I	0		" " " "
	STD	X	I	1		" " " "
17	LRA	X	I			PDQS " " " "

OPCODE FORMATS

I	Indirection
X	Index register
CAPITALS	literals, options, opcodes; non-variable items
[ ]	pick one item from within the brackets; the entire item is optional.
{ }	pick one item from within the brackets; the item is required.
opcode	one of the system/3000 opcodes that uses this format
label id	a label which is within range of the opcode.
variable id	a variable which is within range of the opcode
usi	an unsigned integer less than or equal to the specified number
procedure id	a declared procedure identifier
subroutine id	a declared subroutine identifier

Format 1

(1a)

```
          { label id     }
          { variable id  }
          { DB+ usi 255  }
          { P+  usi 255  }
 opcode   { P-  usi 255  }   [,I] [,X]
          { Q+  usi 127  }
          { Q-  usi 63   }
          { S-  usi 63   }
```

(1b)

```
          { variable id  }
          { DB+ usi 255  }
 opcode   { Q+  usi 127  }   [,I] [,X]
          { Q-  usi 63   }
          { S-  usi 63   }
```

(1c)

```
          { label id     }
 BR       { P+  usi 255  }   [,I] [,X]
          { P-  usi 255  }

          { DB+ usi 255  }
 BR       { Q+  usi 127  }    ,I  [,X]
          { Q-  usi 63   }
          { S-  usi 63   }
```

(1d)

```
          { label id     }
 opcode   { P+  usi 31   }   [,I]
          { P-  usi 31   }
```

(1e) *opcode* { *label id* | P+ *usi* 255 | P– *usi* 255 }

Format 2

opcode
or
opcode, opcode

Note: *opcode* must be stack op.

Format 3

(3a) *opcode* { *label id* | P+ *usi* 255 | P– *usi* 255 } [,I]

(3b) *opcode* *usi* 63 [,X]

Format 4

(4a) *opcode* *usi* 255

(4b) { EXF | DPF } *usi* 15: *usi* 15

Format 5

{ RSW | LLSH | PLDA | PSTA }

Format 6

opcode *usi* 15

Format 7

opcode *usi* 255
or
PCAL procedure id
LLBL procedure id

Format 8

(8a)

```
⎧ MOVE ⎫          ⎡ ,0 ⎤
⎨ MVB  ⎬   [PB]   ⎢ ,1 ⎥
⎩ CMPB ⎭          ⎢ ,2 ⎥
                  ⎣ ,3 ⎦
```

(8b)

```
          ⎧ A   ⎫
          ⎪ N   ⎪ ⎡ ,0 ⎤
MVBW      ⎨ AN  ⎬ ⎢ ,1 ⎥
          ⎪ AS  ⎪ ⎣ ,2 ⎦
          ⎩ ANS ⎭
```

(8c)

```
⎧ MVBL ⎫ ⎡ ,0 ⎤
⎪ MVLB ⎪ ⎢ ,1 ⎥
⎨ SCW  ⎬ ⎣ ,2 ⎦
⎩ SCV  ⎭
```

FUNCTIONAL CROSS REFERENCE

A (S–0 or TOS)

- ADAX
- ADXA
- DECA
- DEL
- DUP
- INCA
- LDXA
- STAX
- XAX
- IABZ
- DABZ

Add

- ADD
- ADDI
- ADDM
- DADD
- FADD
- LADD
- ADXI
- ADAX
- ADBX
- ADXA
- ADXB
- INCM
- INCA
- INCB
- INCX

And

- AND
- ANDI

Arithmetic Shift

- ASL
- ASR
- DASL
- DASR
- TASL
- TASR

B (S–1)

- ADBX
- ADXB
- DECB
- DELB
- INCB
- LDXB
- STBX
- XBX
- ZROB

Bit

- SCAN
- TBC
- TCBC
- TRBC
- TSBC
- TSBM
- DPF
- EXF

Branch

- BCC
- BCY
- BNCY
- BOV
- BNOV
- BR
- BRE
- BRO
- CPRB
- DABZ
- IABZ
- DXBZ
- IXBZ
- MTBA
- MTBX
- TBA
- TBX

Byte

- LDB
- STB
- MVB
- MVBW
- SCU
- SCW
- BTST

Circular Shift

- CSL
- CSR
- DCSL
- DCSR

Compare

- CMP
- CMPM
- CMPI
- CMPN
- DCMP
- FCMP
- LCMP

Condition Code

- BCC
- BTST
- CMP
- DCMP
- DTST
- FCMP
- LCMP
- TEST
- TBC
- TCBC
- TRBC
- TSBC
- TSBM
- CMPI
- CMPN
- CMPB

Decrement

- DECM
- DECA
- DECB
- DECX
- DABZ
- DXBZ

Delete

- DEL
- DELB
- DDEL

Divide

- DIV
- DIVI
- DIVL
- FDIV
- LDIV

Double Integer Arithmetic

- DCMP
- DADD
- DSUB
- MPYL (double result)
- DIVL (double dividend)
- DDEL
- DNEG
- DOUP
- DZRO
- FIXR
- FIXT
- DFLT
- DXCH

Double Word

- LDD
- STD
- LDPP
- LDPN
- DDEL
- DDUP
- DXCH

Double-Word Shift

- DASL
- DASR
- DLSL
- DLSR
- DCSL
- DCSR

Duplicate

- DUP
- DDUP

Exchange

- DXCH
- XAX
- XBX
- XCH
- XCHD
- CAB

Exclusive Or

- XOR
- XORI

Field

- DPF
- EXF

Fix

- FIXT
- FIXR

Float

- FLT
- DELT

Floating-Point

See Real Arithmetic

Immediate

- CMPI
- CMPN
- ADDI

Immediate (cont.)

- SUBI
- MPYI
- DIVI
- ORI
- XORI
- ANDI
- ADXI
- LDI
- LDNI
- LDXI
- LDXN
- SUBXI

Increment

- INCM
- INCA
- INCB
- INCX
- IABZ
- IXBZ

Index Register

- ADXI
- SBXI
- INCX
- DECX
- ZROX
- LDX
- DXBZ
- IXBZ
- MTBX
- TBX
- ADAX
- ADBX
- ADXA
- ADXB
- LDXA
- LDXB
- STAX
- STBX
- XAX
- XBX
- LDXI
- LDXN
- CPRB
- PLDA
- PSTA

Integer Arithmetic

- CMP
- ADD
- SUB
- MPY

Integer Arithmetic (cont.)

- DIV
- INCA
- DECA
- ADXI
- SUBXI
- INCX
- DECX
- CMPI
- CMPN
- ADDI
- SUBI
- MPY
- DIVI
- CMPM
- ADDM
- SUBM
- MPYM
- INCM
- DECM

I/O-Interrupt-Common

- CIO
- CMD
- RIO
- RMSK
- SED
- SIN
- SIO
- SIRF
- SMSK
- TIO
- WIO
- RSW

Load

- LOAD
- LDX
- LDB
- LDD
- LDPP
- LDPN
- LRA
- LDXA
- LDXB
- LDXI
- LDXN
- LDI
- LDNI

Logical Arithmetic

- LCMP
- LADD
- LSUB

Logical Arithmetic (cont.)

- LMPY
- LDIV
- NOT
- OR
- XOR
- AND
- ORI
- XORI
- ANDI

Loop Control

- MTBX
- MTBA
- TBX
- TBA
- CPRB
- IABZ
- IXBZ
- DABZ
- DXBZ

Memory Reference

- ADDM
- CMPM
- DECM
- INCM
- LDB
- LDD
- LDPN
- LDPP
- LDX
- LOAD
- LRA
- MPYM
- STB
- STD
- STOR
- SUBM
- TBA
- TBX
- MTBA
- MTBX
- BR
- BCC

Move

- CMPB
- MOVE
- MVB
- MVBW
- MVBL
- MVLB

Move (cont.)

- SCU
- SCW

Multiply

- MPY
- MPYM
- MPYI
- MPYL
- FMPY
- LMPY

Negate

- NEG
- DNEG
- FNEG
- NOT

Or

- OR
- ORI
- XOR
- XORI

Privileged

- CIO
- CMD
- RIO
- RMSK
- SED
- SIN
- SIO
- SIRF
- SMSK
- TIO
- WIO
- HALT
- PAUS
- SETR
- PLDA
- PSTA
- LLSH
- MVBL
- MVLB
- XCHD

Procedures

- PCAL
- EXIT

Program Control

- PCAL
- EXIT
- SCAL
- SXIT
- HALT
- PAUS
- XEQ
- LLBL
- NOP

Real Arithmetic

- FCMP
- FADD
- FSUB
- FMPY
- FDIV
- FNEG
- FLT
- DFLT
- FIXR
- FIXT
- DDVP
- DZRO
- DDEL
- DXCH

Register Control

- ADDS
- SUBS
- PSHR
- SETR
- XCHD

Scan

- SCW
- SCU
- SCAN

Shift

- ASL
- ASR
- CSL
- CSR
- LSL
- LSR
- DASL
- DASR
- DCSL
- DCSR
- DLSL
- DLSR

Shift (cont.)

- TASL
- TASR
- TNSL

Single-Word

- LOAD
- STORE
- MOVE
- MVLB
- MVBL
- PLDA
- PSTA
- LDX
- DEL
- XCH
- DELB
- DUP
- LDXA
- LDXB
- STXA
- STXB
- XAX
- XBX

Single-Word SHift

- CSL
- CSR
- ASL
- ASR
- LSL
- LSR

Special

- LLBL
- LLSH
- PLDA
- PSTA
- RSW

Stack op

- ADAX
- ADBX
- ADD
- ADXA
- ADXB
- AND
- BTST
- CAB
- CMP
- DADD
- DCMP
- DDEL

Stack op (cont.)

- DDUP
- DECA
- DECB
- DECX
- DEL
- DELB
- DFLT
- DIV
- DIVL
- DNEG
- DSUB
- DTST
- DUP
- DXCH
- DZRO
- FADD
- FCMP
- FDIV
- FIXR
- FIXT
- FLT
- FMPY
- FNEG
- FSUB
- INCA
- INCB
- INCX
- LADD
- LCMP
- LDIV
- LDXA
- LDXB
- LMPY
- LSUB
- MPY
- MPYL
- NEG
- NOP
- NOT
- OR
- STAX
- STBX
- SUB
- TEST
- XAX
- XBX
- XCH
- XOR
- ZERO
- ZROB
- ZROX

Store

- STOR
- STB
- STD
- STAX
- STBX

Subroutines

- SCAL
- SXIT

Subtract

- SUB
- SUBM
- SUBI
- DSUB
- FSUB
- LSUB
- SBXI
- DECM
- DECA
- DECB
- DECX

Test

- BTST
- TEST
- LTST
- TBC
- TCBC
- TRBC
- TSBC
- TSBM
- TBA
- TBX
- MTBA
- MTBX

Triple-Word Shifts

- TASL
- TASR
- TNSL
- CAB

Zero

- ZERO
- DZRO
- ZROB
- ZROX

APPENDIX F
BNF Syntax Index

This index lists all of the syntax productions or rules in this manual. They are in alphabetic order, with the enclosing less than (<) and greater (>) stripped off.

A

actual param, 4-6, 5-25, 5-33
actual param part, 4-6, 5-25, 5-33
addop, 3-12, 4-11
address part, 5-2
address specification, 3-22
adr mode, 5-2
aexp, 4-11
arg 1, 5-3
array init, 3-17
ASCII characters, 1-3
ASSEMBLE statement, 5-2
atype, 3-16, 3-22, 3-29, 3-38

B

base, 2-2
based integer, 2-2
base part, 2-2
base register reference, 3-14,3-18, 3-23
bit cat field, 4-8
bit extract field, 4-8
branch subopl, 5-3
branch word, 5-17
btestword, 4-15
byte ref, 4-15, 5-21, 5-30

C

case body, 5-8
CASE statement, 5-8
ccf, 5-4, 5-22
char, 5-30
character, 2-7
character string, 2-7
comment, 1-3
composite integer, 2-2
compound statement, 1-1, 5-1, 5-8
compound tail, 1-1, 3-30, 5-1, 5-8
cond clause, 5-17
cond elem, 5-17
cond factor, 5-17
cond primary, 5-17
cond term, 5-17
conjunction, 4-14
const, 5-4
constant, 2-1
count, 4-15, 5-21
ctype, 3-28, 3-38

D

data declaration, 3-1
data group, 1-1, 3-1
db, 3-16
decimal integer, 2-2
define declaration, 3-11
define identifier, 2-8, 3-11
define invocation, 3-11
definition, 3-17
DELETE statement, 5-10
deposit field, 5-6
deposit field length, 5-6
digit, 2-2, 2-8
disjunction, 4-14
DO statement, 5-12
double integer, 2-2

E

E-dec, 3-17
ELSE part, 5-17
empty, 1-1
entry declaration, 3-27
entry identifier, 2-8
equate, 3-12
equate declaration, 3-12
equate expression, 3-12
equate identifier, 2-8, 3-12
entry identifier, 3-27
equate invocation, 3-12
equate primary, 3-12
equate term, 3-12
expr, 4-1
extract field length, 4-8

F

FOR clause, 5-13
formal param, 3-28, 3-37
formal part, 3-28, 3-37
format-1, 5-2
format-2, 5-2
format-3, 5-3
format-4, 5-3
format-5, 5-3
format-6, 5-3
format-7, 5-3
format-8, 5-4
format-9, 5-4
FOR statement, 5-13
fraction, 2-4

G

g-array dec, 3-16
G-dec, 3-16
global array declaration, 3-16
global attribute, 3-14, 3-22
global head, 1-1
global pointer declaration, 3-22
global simpvar declaration, 3-14
GO statement, 5-15

I

identifier, 2-8
I-field, 5-2
IF statement, 5-17
index, 4-3
indexed identifier reference, 3-17
indexed ident reference, 3-22
indirect base register reference, 3-18
initial value, 3-14, 3-17, 3-18, 4-15, 5-22
instruction, 5-2
instruction slist, 5-2
integer, 2-2
integer field, 2-2
integer variable, 4-3
intrinsic declaration, 3-36
intrinsic identifier, 2-8, 3-36

K

K-field, 5-3

L

label declaration, 3-25
label identifier, 2-8, 3-25, 3-26
label ref, 5-15
l-array dec, 3-17
left deposit bit, 4-8, 5-6
left extract bit, 4-8
letter, 2-8
lexp, 4-14
listelm, 3-17
listelmt, 4-15, 5-22
local array declaration, 3-16
local pointer declaration, 3-22
local simpvar declaration, 3-14
logical addop, 4-14
logical elem, 4-14
logical expr, 4-1
logical factor, 4-14
logical mulop, 4-15
logical primary, 4-14
logical term, 4-14
logical value, 2-6
long real number, 2-4

M

main body, 1-1
memory ref opcode, 5-2
MOVE statement, 5-21
MOVE-WHILE stmt, 5-21
muldiv, 3-12
mulop, 4-11

N

non-branch subopl, 5-3
nonref var dec, 3-14
number, 2-1
number of bits, 2-2

O

opcode format, 5-2
option, 3-29
option part, 3-29
own array dec, 3-17

P

pcount, 5-29
pointer dec, 3-22
pointer init, 3-22

power, 2-4
proc body, 3-29
proc data declaration, 3-29
proc data group, 3-29
PROCEDURE call statement, 5-25
procedure declaration, 3-28
procedure group, 1-1, 3-1, 3-29
procedure identifier, 2-8, 3-28
proc group, 1-1, 3-1, 3-30
proc head, 3-28
proc identifier, 3-28
program, 1-1
PUSH and SET statement, 5-27

R

real number, 2-4
reference param, 4-6, 5-25, 5-33
reference part, 3-17
register spec, 5-27
relop, 4-14, 5-17
RETURN statement, 5-29
right part, 5-6

S

sadmode, 5-4, 5-22
scanop, 5-4
SCAN statement, 5-30
SCAN-UNTIL stmt, 5-30
SCAN-WHILE statement, 5-30
sdec, 4-15, 5-4, 5-21, 5-30
sdeca, 4-15, 5-4, 5-21, 5-30
shift count, 4-9
shift op, 4-8
sign, 2-2, 2-4, 3-12, 3-18, 3-23, 5-3
simpvar init, 3-14
sindex, 5-15
special op, 5-3
specification, 3-29, 3-37
specification part, 3-28, 3-37
stacked param, 4-6, 5-25, 5-33
stack opcode, 5-3
statement, 5-1
STEP clause, 5-13
string, 2-7
stype, 3-38
sub body, 3-37
subdata declaration, 3-2
subdata group, 1-1, 3-1
sub head, 3-37
sub memret op, 5-2
sub move op, 5-4
subop3, 5-3
subprogram, 1-1
subr identifier, 2-8, 3-37
SUBROUTINE call statement, 5-33
subroutine declaration, 3-37
subroutine identifier, 2-8, 3-37
sub subop2, 5-3
switch declaration, 3-26
switch identifier, 2-8, 3-26

T

T array identifier, 2-8, 3-16
T assignment statement, 5-6
T bit operation, 4-8
T bit shift, 4-8
T data identifier, 3-14, 3-18, 3-23
testword, 5-30
text, 3-11
T function designator, 4-6
T identifier, 2-8, 3-14
T IF expr, 4-1
T_{ilb} bit concatentation, 4-8
T_{ilb} bit extraction, 4-8
T_{irbde} aexp, 4-11
T_{irbde} expr, 4-1
T_{irbde} factor, 4-11
T_{irbde} primary, 4-11
T_{irbde} term, 4-11
T_{irlde} dest ref, 5-21
THEN part, 5-17
T left part, 5-6
T pointarr, 5-21
T pointer identifier, 2-8, 3-22
T proc identifier, 2-8
T simpvar identifier, 2-8, 3-14
T subr identifier, 2-8
T variable, 4-3
type, 3-14, 3-16, 3-22, 3-28, 3-38

U

udb, 3-17
unsigned integer, 2-2
unsigned long real number, 2-4
unsigned real number, 2-4
usi, 3-14, 3-18, 3-23
usi 31, 5-3
usi 63, 5-3
usi 255, 5-2

V

value param, 4-6, 5-25, 5-33
value part, 3-28, 3-37
var dec, 3-14
var identifier, 5-2
var reference, 3-14, 3-17, 3-23
vb, 3-17

W, X

WHILE statement, 5-35
X-field, 5-2

APPENDIX G
Building an Intrinsic File

The program BUILDINT is used to build or change intrinsic disc files. The program uses $STDIN for input and $STDLIST for list output. The intrinsic data file is opened as SPLINTR.

The command to execute the program is

:RUN BUILDINT

The input data consists of SPL/3000 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
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
```