BCC 500
SPL LANGUAGE REFERENCE MANUAL

Chuck Wall
Judy Simon

THE ALOHA SYSTEM
University of Hawaii

Document No. BCC/M-2
December 31, 1973

Contract NAS2-6700
Department of Defense
Advanced Research Projects Agency
ARPA Order No. 1956

ABSTRACT and CONTENTS

This is the reference manual for the BCC 5ØØ System Programming Language (SPL). The syntax and semantics of the language are defined. This reference manual is an edited and updated version of a working paper written by Butler W. Lampson.

TABLE OF CONTENTS

1. Introduction 5

2. Meta Language 6

   2.1 Syntax for Meta Language Equations 7

   2.2 Syntax of Meta Language Tokens 8

3. Scopes and Program Format 10

   3.1 Lexical Format 11

   3.2 Scopes 12

4. Allocation 15

   4.1 Permanency of Storage 15

   4.2 Layout of Core 16

   4.3 Origins 18

   4.4 Fixed Environments 19

   4.5 Equivalence 20

   4.6 Fixed Fields 20

5. Declarations 21

   5.1 Names 21

   5.2 Declaration Statements 21

   5.3 Attributes 24

   5.4 Attribute Modifiers 27

      5.4.1 Dimensions 27

      5.4.2 Length 28

      5.4.3 Form 29

   5.5 Equivalence 29

   5.6 Initialization 30

5.7 Constants.. 32

5.7.1 Integer Constants...................................... 32

5.7.2 Real Constants... 33

5.7.3 Double Constants....................................... 34

5.7.4 Imaginary Constants.................................... 34

5.7.5 String Constants....................................... 34

5.7.6 Label Constants.. 35

5.7.7 Constant Expressions................................... 35

5.8 Data Formats... 35

5.9 Function Declarations.. 38

5.9.1 Formal Parameters...................................... 38

5.9.2 FRETURNs.. 39

5.9.3 Special Entry Points................................... 39

5.10 Macros.. 43

6. Action:Statements... 45

6.1 Expressions.. 45

6.1.1 Precedence of Operators................................ 46

6.1.2 Syntax of Expressions.................................. 47

6.1.3 Types of Operands...................................... 49

6.1.4 Semantics of Expressions............................... 53

6.1.5 Array Expressions...................................... 57

6.1.6 Function Call and Return Expressions................... 59

6.1.6.1 Arguments.. 59

6.1.6.2 Returns.. 60

6.1.6.3 Failure Exits...................................... 61

6.2 Expressions Used as Statements............................... 61

6.2.1 IF Statements.. 62

6.2.2 FOR Statements... 63

6.3 Assembly Language.. 65

7. Intrinsic Functions.. 67

7.1 Type Conversion Functions.............................. 70

7.2 String Functions.. 72

7.3 Storage Allocation Functions........................... 75

7.4 Miscellaneous Functions................................. 77

Appendix I - List of Keywords.............................. 78

1. Introduction

The BCC-500 Systems Programming Language (SPL) was designed for use in systems development, translator writing and high level sub-system development. It is sufficiently flexible and generates sufficiently efficient object code to replace assembly language in almost all situations. SPL is highly interactive, simple to use from a terminal and helpful in error situations.

This reference manual defines the syntax and the semantics of the language. Details about the command language, the editor and debugger will appear in another manual.

2. Meta Language

A syntax equation describes the form of some construction in a language. For example, the following equations describe simple arithmetic expressions.

```
expression = ["+/"-"] term $(("+"/"-") term);
term       = primary $(("*"/"/") primary );
primary    = symbol / number / "(" expression ")";
```

The first equation reads "an expression has the form of an optional plus or minus followed by a term followed by an arbitrary number (including Ø) of the constructs (denoted by '$'), either a plus or minus sign followed by a term." The last reads "a primary has the form of a symbol or a number or a literal left parenthesis followed by an expression followed by a right parenthesis." Symbols such as 'primary', 'term' and 'expression' are defined above and are called non-terminal symbols. 'Symbol' and 'number' are not defined above and are, therefore, called terminal symbols. The construct "(" is an example of a literal.

2.1 Syntax for Meta Language Equations

```
equation     = symbol "=" alternation ";";
alternation  = catenation $("/" catenation);
catenation   = term $term;
term         = primary / optional / repetition / negation;
primary      = symbol / literal / "(" alternation ")" /
               "<" text ">";
optional     = "[" alternation "]";
repetition   = [integer] "$" [integer] primary;
negation     = primary "-" primary;
```

The alternation indicates that any alternative is legal; a catenation indicates that the terms follow each other. The last alternative of the definition of a primary is included to allow an "escape" to normal English text. Naturally the text may not include a ">" character. This convention should be avoided if possible.

An 'optional' indicates zero or one occurrence of the alternatives enclosed in brackets. The meaning of the four kinds of repetitions are:

Form	Meaning
$ something	arbitrary number of 'something's'
1$ something	one or more 'something's'
$6 something	zero to six 'something's'
2$7 something	two to seven 'something's'

The negation construct is used to make an exception. It translates "any occurrence of the first primary except for an occcurrence of the second primary."

2.2 Syntax of Meta Language Tokens

Many languages are built out of small units, called tokens. The tokens of the metalanguage are symbols, integers, literals and certain punctuation characters such as "/", "$" and ";" (see below). In such languages blanks and carriage returns may appear between tokens without affecting the meaning of the information. The usual rule is that blanks and carriage returns delimit (or separate) tokens, but are otherwise ignored. Unless the contrary is explicitly stated in our documents, the reader may assume the treatment of blanks is as described above.

These equations define the tokens:

```
symbol  = word $(":" word);
word    = letter $(letter / digit);
integer = digit $digit;
literal = '"' $char1 '"' / "'" $char2 "'";
char1   = character - '"';
char2   = character - "'";
```

A symbol is a sequence of words separated with colons. A word is at least one letter optionally followed by letters or digit. An integer is represented in the normal way. A literal is a string of either char1's enclosed by double quotes or char2's enclosed by single quotes. The first form is preferred. 'char1' is any character except double quote; 'char2', any except single quote. Note that "?" "!" is not the same as "?!". The first form implies that the "?" and "!" may be separated by blanks, while the second requires that the two characters appear consecutively with no intervening blanks.

These three equations define the basic set of characters for defining tokens.

```
character = letter / digit;
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"/
            "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     = "Ø"/"1"/"2"/"3"/"4"/"5"/"6"/"7"/"8"/"9";
```

3. Scopes and Program Format

An SPL program is organized into blocks. A block begins with a PROGRAM or COMMON statement and ends with an END statement. It has a name which is given in its initial statement. Block names must be unique over the entire program. Thus the general format of a program is:

```
program               = $block;
block                 = common:block / program:block;
common:block          = "COMMON" identifier ";"
                        block:head
                        end:statement ";";
program:block         = "PROGRAM" identifier ";"
                        block:head
                        $($(label ":")
                        action:statement ";")
                        end:statement ";";
block:head            = $(include:statement ";")
                        $(allocation:statement ";")
                        $(declare:statement ";");
allocation:statement  = fixed:statement /
                        origin:statement;
label                 = identifier;
identifier            = name;
```

Thus, statements must occur in a block in the order:

PROGRAM or COMMON statement

include:statements

allocation:statements

declare:statements

action:statements

end:statement

A common:block must precede any blocks which INCLUDE it.

3.1 Lexical format

Every statement ends with a semi-colon.

Carriage returns and blanks are treated according to the following rules:

1) inside string constants or character constants blanks are treated like ordinary characters. Carriage returns are illegal in string and character constants (unless written with the '&' escape convention).
2) elsewhere a string of carriage returns and blanks is equivalent to a single blank.
3) a blank may appear anywhere except in the middle of a token. Tokens include names, reserved words, constants, the sequences "<=" ">=" "**" "//" that are SPL two character operators.

To summarize these rules somewhat sloppily we say that carriage returns and blanks are ignored except in string constants, names, and reserved words.

A comment has the form:

```
comment  =  <carriage return> "*" <arbitrary
            string of characters not including
            carriage return> <carriage return> /
            "/*" <arbitrary string of characters
            not including "*/" or carriage return>
            ("*/" / <carriage return>);
```

The first form of comments is exactly equivalent to a carriage return. The second form is equivalent to a blank if it ends with "*/", a carriage return if it ends with a carriage return; the difference is apparent only if it is immediately followed by "*".

Note that a multi-line comment must have an * or /* at the start of each line.

3.2 Scopes

Each variable is declared in some block and is said to be *local* to that block. The same identifier may refer to two different variables which are local to different blocks. The variable name together with the block name, however, is sufficient to identify the variable uniquely. A variable is said to be LOCAL in scope if it is local to a program block, COMMON if it is local to a common block.

Function names (i.e. names which appear immediately after FUNCTION or ENTRY) are GLOBAL in scope, however.

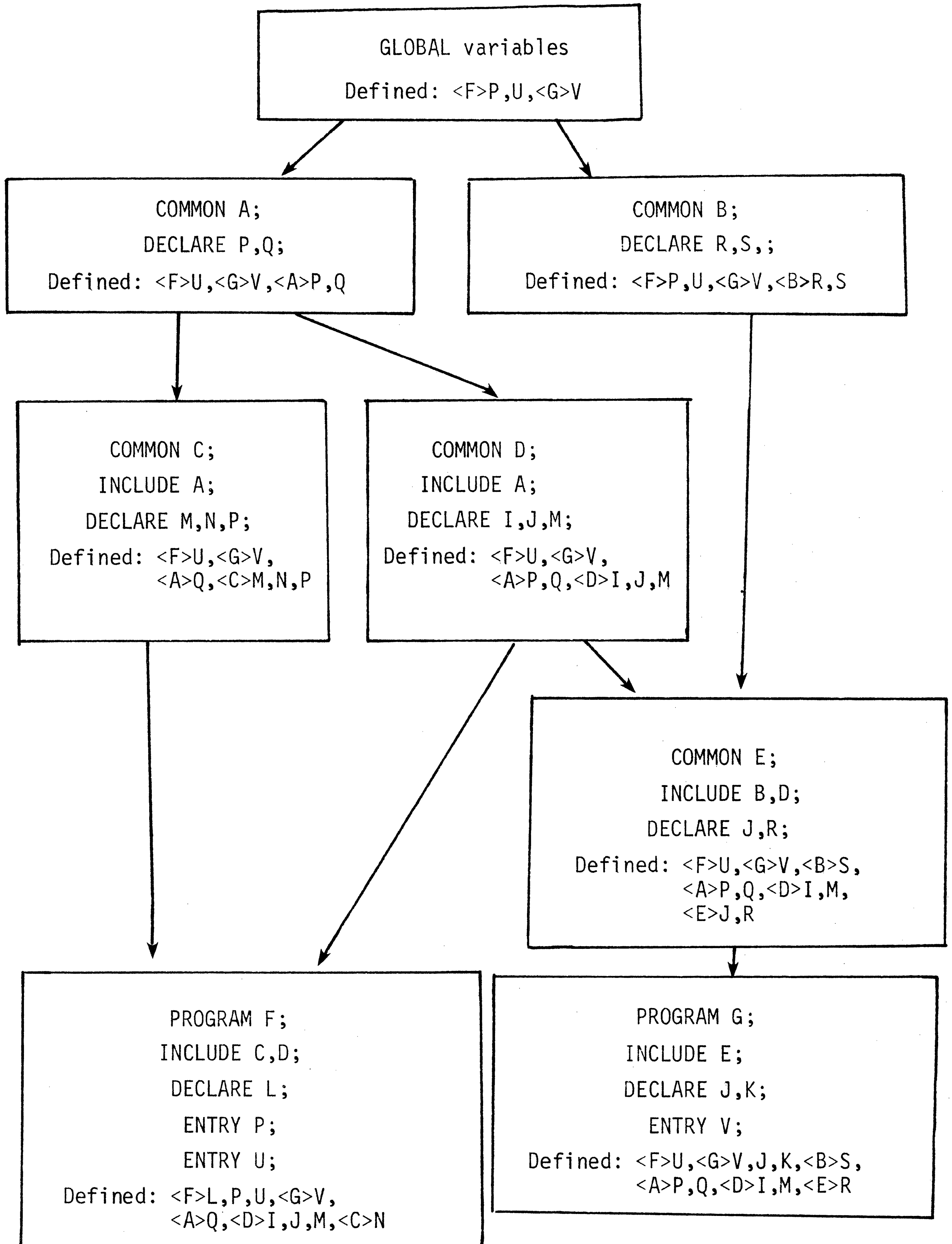
A variable may be referenced only in a block in which it is defined. Any variable is defined in the block to which it is local. Suppose that block C includes COMMON blocks Bi (i=1,...,n) in that order. Then a variable defined in Bj is also defined in C unless it is local to C or defined in Bi, i>j. A block includes Bi if Bi appears in the identifier:list of an include:statement in the block.

```
include:statement  =  "INCLUDE" identifier:list;
identifier:list    =  identifier $("," identifier);
```

All GLOBAL variables are considered to be defined in a GLOBAL COMMON block which is considered to be included in every block.

The effect of this convention is that declarations in COMMON blocks can be overridden by other declarations nearer the point of use. Exception: a MACRO name cannot be overridden. Note that if B includes A and C includes B, then the variables local to A are defined in C (unless variables of the same name are local to B or C). A declaration overriding an INCLUDE must occur before any reference to the variable involved. See figure 1. for an illustration.

Figure 1: Determining Defined Variables
(In the following figure <F>P means P local to F)



4. Allocation

SPL has a considerable amount of machinery for controlling the allocation of storage for programs and data. Much of this machinery is of limited interest, but a few parts of it are important to nearly all programmers. This section discusses the topics of general interest first, before going on to the others. The reader is advised to break off when he encounters material of no relevance to his needs.

4.1 Permanency of Storage

Data in SPL is of two kinds: permanent and stacked. Permanent data stays around for the life of a program, i.e. the value of a permanent data item, once set, survives until explicitly changed by the program. All data declared in COMMON blocks is permanent. Data declared in PROGRAM blocks is permanent if the block includes a

```
fixed:statement  =  "FIXED" ["," "ORIGIN" expr];
```

The function of the ORIGIN clause is explained in Section 4.3. This statement, if it is present, must appear between the include:statement and the declare:statements of the block. The FIXED program block may not be entered recursively (by funtion calls) during execution. This error is <u>not</u> checked for.

If a program block is not FIXED, all the variables local to it are stacked. This means that their values are undefined when the block is entered (by a call to one of its functions), may become defined by the action of the program and disappear when the function returns. The block may be entered recursively, and the values of its local variables for each level of recursion are completely distinct.

4.2 Layout of Core

The arrangement of memory relative to G' (the global environment) is designed to group read-only objects together and on separate pages from writeable things, so that the former can be protected by the hardware from modification. Later improvements will permit small programs to be packed together better.

Space is allocated in four main regions

G':WGS→ ←RSGS:G'+40000B:CS→ :377777B

WGS: Writeable Global Storage, starting at G'. This area is allocated by a general storage allocator in the compiler in a piecemeal fashion: no attempt is made to keep related things together. All the writeable variables which appear in common blocks, together with fixed local environments are put here.
Some of the first 128 words may also be used for field and array

descriptors, at the discretion of the compiler, except in the monitor ring, where this will never be done (unless forced by equivalences). The first few words, of course, are used for objects whose location is fixed by the hardware, like the stack descriptor. The allocation strategy for this area may be modified by ORIGIN statements; see below.

Collision of this area with RSGS is a fatal error.

The stack (where stacked data is stored) is allocated space at the end of this area. Its size depends on the number of non-FIXED PROGRAM blocks entered but not exited.

RSGS: Read-only Scalar Global Storage. Here are put the constant scalars (e.g. array descriptors and initialized scalars) from common blocks, as well as function descriptors. This area is allocated by another incarnation of the general storage allocation used for WGS and on the same piecemeal basis.

CS: Code Storage. Space here is allocated by block. All the code and constants generated by one program block, or all the non-scalar constants (strings, arrays, etc.) generated by one common block, are collected together and allocated contiguously in that region. Transfer vectors also appear here. If block A precedes block B lexically (in the source), then the

CS for A will precede the CS for B.

4.3 Origins

The origin:statement permits (most of the) storage of a block to be allocated at a fixed place.

```
origin:statement  =  "ORIGIN" [expr];
```

The expression, whose value is called the origin of the block, must evaluate to an integer at compile-time. The statement must appear in the block after any include:statement and before anything else.

If the block is a *program* block or a *common* block with no writeable variables declared, the origin tells where to start its space in CS. If the space for the preceding block in CS extends past the specified origin, an error is recorded and the statement is ignored. This implies that origined blocks must appear in order of increasing origins. Note that the scalar storage of a common block is allocated in RSGS and is not affected by origin:statements.

If the block is a <u>common</u> block with writeable storage, then the origin tells when to start this storage. Two restrictions apply

1) The block must have no requirements for CS.
2) All blocks with origined WGS must appear before any non-origined blocks which require WGS, so that the space taken by origined blocks can be properly removed from the control of the storage allocator.

All the WGS for an origined block is allocated together. A subsequent block may omit the expr from its origin:statement, in which case its WGS is allocated immediately following that of the preceding block.

4.4 Fixed Environments

The location of a fixed local environment may be specified by the fixed:statement, thus:

FIXED, ORIGIN expr;

The origin clause tells where to put the environment. The programmer is responsible for the security of the area he chooses, which is not checked by the compiler. In the absence of the ORIGIN, the compiler will allocate the storage in WGS at its discretion.

4.5 Equivalence

An equivalence can be used to fix the location of a scalar or an array descriptor by writing an integer-valued expression for the object of the equivalence. Thus

```
DECLARE A = 40B, ARRAY B[30] = 41B;
```

allocates A at 40 and the descriptor for the array B at locations 41 and 42 octal. The array itself is allocated according to the default rules. Restriction: the value of the equivalence must be in the range [G',G'+37777B]. An equivalence overrides all other methods of storage allocation. If a variable V has been equivalenced to a constant, or is declared in a common block, then @V is a constant whose value is the address assigned to V.

4.6 Fixed Fields

Descriptors for part-word fields are normally allocated in the first 128 words of the global environment by the compiler if there is room. This allocation can be suppressed and the field allocated in the function or common block like any other constant by prefixing FIXED to [SIGNED] FIELD in the declaration.

5. Declarations

5.1 Names

A name is a sequence of not more than 16 characters starting with a letter, each of which must be either alphanumeric or an ' (apostrophe).

5.2 Declaration Statements

A declaration consists of a list of names together with specification of scope, type and mode, and possibly of initialization and equivalence. Thus:

```
declare:state-
ment              = "DECLARE" declare:clause:list /
                    macro:statement;

declare:clause:
list              = declare:clause $("," declare:clause);

declare:clause    = [type] [mode] item;

item              = identifier [form / [dimension] / [length]]
                    [equivalence]  [initialization];

equivalence       = "=" (identifier [subscript:list]/
                    ("L'" / "G'") subscript:list / expression);
```

```
subscript:list    = "[" expression $("," expression) "]";

initialization    = "←" (expression / "(" expression
                     $("," expression) ")" );
```

Where L' means local environment and G' means the global environment.

A declaration is processed from left to right. The attributes are initialized as follows:

scope	-	is determined by whether the declaration is in a PROGRAM block (LOCAL) or a COMMON block (COMMON)
type	-	INTEGER
mode	-	SCALAR

The values of the three attributes are called the state of the declaration. Occurrence of attribute specifiers may change the state. A name is given the attributes which are in the state when it is encountered, except that the form, dimension, or length, if appropriate, may follow the name as indicated in the syntax of item. FUNCTION, FIELD, and ARRAY are taken as specifying mode unless immediately followed by a mode word, in which case they specify type. Occurence of a type word sets the mode to SCALAR; occurence of a mode word leaves the type unchanged. UNKNOWN SCALARs are not allowed.

EXAMPLE:

```
DECLARE INTEGER A, B, STRING C, ARRAY D, E [5],
ARRAY [10] F, G (5:12);
```

declares integer scalars A and B, string scalar C, string arrays D, E, F, and G. The array D is not assigned any storage, but E is assigned 5 elements and F and G get 10. Except for G, no space is assigned for the string values and all the strings have 8-bit bytes; each element of G is assigned space for 5 bytes at 12 bits each. All these things are _local_.

5.3 Attributes

Every name has three attributes: scope, type, and mode. Each is chosen from a fixed set of alternatives:

```
scope          =   "COMMON" / "LOCAL" / "GLOBAL" ;

type           =   ntype / ptype / "STRING" [length] ;

   ptype       =   "PARAMETER" integer / "PARAMETER REAL" /
                   "PARAMETER DOUBLE" / "PARAMETER COMPLEX" ;

   ntype       =   integer / "REAL" / "DOUBLE" / "COMPLEX" /
                   "LABEL" / "LONG" / "LONGLONG" /
                   "FUNCTION" / "FIELD" / "ARRAY" /
                   "UNKNOWN" ;

   integer     =   "INTEGER" / "OCTAL" / "CHARACTER" /
                   "POINTER" ;

mode           =   "FUNCTION" / ["FIXED"] ["SIGNED"] "FIELD" [form] /
                   ("ARRAY" / "ARRAYONE") [dimension]/
                   "SCALAR" ;
```

Note that if the type is not specified "INTEGER" is assumed.

Certain constructions permitted by the above syntax are forbidden because no reasonable meanings can be attached to them.

1) Objects of type ARRAY or STRING with mode FUNCTION or FIELD do not need dimensions and lengths, and to give them as part of the item is an error.

2) A form may appear only if the mode is FIELD, a dimension only if the mode is ARRAY, a length only if the type is STRING.

The type of scalar value determines its size: integer, function and field are one word values; long, real, array, and label, are two words each; string, double, longlong, and complex, are four words each. An array is represented by a two-word descriptor, as is a label scalar. A function scalar is represented by a pointer to the two-word descriptor. A field scalar is either a constant, if its form is specified, or occupies a single word. The four cases of integer are included to permit intelligent printout of the value during debugging. The compiler recognizes only one type of integer, and the others will not be mentioned again.

If a name has mode ARRAY (or ARRAYONE; they are identical except that the latter causes subscripts to start at 1 rather than Ø), subscripted references to it will be compiled on the assumption that indirection through the descriptor will produce the effective address. It is also possible to subscript INTEGER SCALARS; such references will add the value of the name to the subscript to produce the effective address.

If a name is a field without a form, tailing (".", "$" or "@") will cause indirection through the location allocated to it. If it has a form, it is treated as a constant and

and the code compiled depends on whether it is full-word or part-word. If a field appears without tailing, it is treated as an integer whose value is the field descriptor if the field had a form, and the contents of the location allocated to it otherwise. If a field is SIGNED, the top bit will be copied into all the higher bit positions of a 24-bit word when the field is used to fetch a datum. Otherwise, these bit positions (if any) will be filled with zeros.

5.4 Attribute Modifiers

The shapes and sizes of arrays and strings are specified by modifiers (dimensions, lengths and forms) which have already appeared in the syntax for attribute names. Throughout, the expressions must evaluate to constants at compile time. This means that all the operands must be constant. See "Constants" in Section 5.7 for a discussion of what operands are regarded as constant.

5.4.1 Dimensions

```
dimension  =  "[" expr $("," expr)
              [":" [expr] ["," expr]] "]";
```

Arrays of any dimensionality up to 7 are allowed. The first expression following the colon specifies the number of words allocated to each element of the array; this makes it easy to create tables with multi-word entries. The size of an element is limited to 64 words. If it is not specified, it

is taken to be the size of the scalar object with the same attributes as the declared array. If an array is given an element size different from the one implied by its type, then subscripting it yields an expression of type UNKNOWN. See "Expression" (Section 6.1) for the implications of this.

The second expression following the colon tells where to allocate the first word of the array. If it is absent, the array is allocated using standard policies described in Section 4 "Allocation."

5.4.2 Length

```
length = "(" expr [":" [expr] ["," expr]] ")" ;
```

The string length is specified in bytes by the first expression. The second expression gives the byte size, chosen from 6, 8, 12, and 24; 8 is the default value. The third expression tells where to allocate the first word of the string.

If an array or string lacks dimension or length, no space is allocated and no descriptor created by the compiler. In this case an array is assumed to take one subscript.
If these elements are present, space is assigned to the local environment if the scope is LOCAL and the descriptors are initialized at function entry. If the scope is COMMON, space is assigned in the proper common block and descriptors compiled into this block. See Section 4 on "Allocation" for details.

5.4.3 Form

```
form              =  "(" word:displacement [ ":"
                     starting:bit "," ending:bit]
                     ")" ;

word:displace-
ment              =  expr;

starting:bit      =  expr;

ending:bit        =  expr;
```

A form specifies the word displacement and left- and right-most bits of a field. If the bit numbers are omitted, Ø and 23 are used. A field may not cross a word boundary.

5.5 Equivalence

An equivalence has the following meaning: the identifier following the "=", called the object, must be previously declared; if subscripts appear, it must be a dimensioned array and the number of subscripts must match the number of dimensions. The effect is to assign the same storage to the identifier preceding the "=", called the subject, as has already been assigned to the object. If the identifier in the object is L' or G', the subject is assigned to the designated location in the local or global environment respectively. If the subject is

a dimensioned array or a string, its descriptor is assigned to the same location as the object (to allocate the storage for array or string values, see above); otherwise the subject itself is assigned to the same location as the object. No account is taken of the possibility that the subject may occupy more space than has been allocated for the object. For some details and restrictions, see "Allocation" (Section 4).

5.6 Initialization

Initialization of SCALARs has the following meaning: if no equivalence is present, the identifier being declared becomes synonymous with the initialization quantity. For INTEGERs which lie in [-2ØØØB, 1777B], no space is allocated; for all other types, and for INTEGERs outside this range, space is allocated to hold the constant value in RSGS (for COMMON blocks) or CS (for PROGRAMs). If an equivalence appears, the object must be an absolute location (see "Allocation"), a scalar or array element with scope = COMMON, or an element of an initialized local array, and the initialization quantity will be stored into the variable, wherever it may be.

For each type of SCALAR, the initialization quantity must be a constant of that type. A LONG or a LONGLONG may be initialized with a string constant or with a list of integers; this is the only way of introducing constants of these types into an SPL program. An initialized STRING must not have a length.

Numeric initialized variables, if not equivalenced, may be re-initialized. This is primarily useful for things like defining fields, etc., using a compile-time counter. If block A includes block B, re-initialization by a declaration in A of a variable acquired from B has no effect on B or any other block that includes B.

Initialization of FUNCTIONs is done with a single name; otherwise the comments above apply. Initialization of FIELDs is illegal; the way to do this is to specify the form explicitly.

An ARRAY is initialized with a list of constants of the appropriate type. (Elements of the list are separated by commas and the list is enclosed in parentheses, as usual.) A FIELD ARRAY may be initialized with constant FIELD SCALARs; an ARRAY ARRAY may be initialized with names of arrays which have been declared with dimensions. The elements of the list go into successive elements of the array, starting with the first one. For multi-dimensional arrays, the last subscript varies most rapidly, just as the array is actually stored.

A special feature allows initialization, at a later time, of further elements of an array some of whose elements have already been initialized. If X is a declared, initialized array, then the appearance of

X subscript:list initialization

as a declare:clause will cause the expression(s) in the initialization to be stored into elements of the array starting at the one designated by the subscript:list. Of course, all the subscripts must be constant.

5.7 Constants

For each type there is a syntax for constants representing values of this type.

5.7.1 Integer Constants

```
integer:constant  =  simple:integer / character:constant;

simple:integer    =  digit $(digit) ["B" [digit]];
```

If B appears, it causes the digits to be interpreted as octal; otherwise they are taken to be decimal. If a digit n follows the B, it is a scale factor, i.e., it is equivalent to n zeros preceding the B.

```
character:constant  =  ("6'" $4(pseudo:character)/
                        ("8'" / "'") $3 (pseudo:character))
                        "'";
pseudo:character    =  <character other than & or '>/
                        "&" letter/ "&&" / "&'" / '&"'/
                        "&" 3$3 digit;
```

A character constant allows up to three 8-bit or four 6-bit characters to be right-justified in a 24-bit word to make an integer. Pseudo:characters permit quotes and control characters to appear; the latter are specified by the letter whose code is less by 100B.

5.7.2 Real Constants

```
real:constant          =  simple:real:constant [exponent] /
                          digits exponent;
simple:real:constant   =  digits "." $(digit) / "." digits;
exponent               =  "E" sign digits;
sign                   =  ["+" / "-"];
digits                 =  1$(digit);
```

The meaning of this should be obvious; the given decimal approximation to a real number is converted to the closest approximation possible in the machine's 48-bit binary representation.

5.7.3 Double Constants

```
double:constant  =  (simple:real:constant / digits) "D"
                    ["+" / "-"] digits;
```

In this case the machine's 96-bit binary representation is used. Note that D must appear in a double constant, and that either . or E must appear in a real constant.

5.7.4 Imaginary Constants

```
imaginary:constant  =  (real:constant / digits) "I";
```

Complex constants may be constructed by arithmetic on real:constants and imaginary:constants; such arithmetic is performed at compile time, resulting in a single complex constant in the object code.

5.7.5 String Constants

```
string:constant  =  ('6"' / '8"' / '"') $(pseudo:character) '"';
```

The value is a string with the specified sequence of characters encoded in 8-bit (default case) or 6-bit bytes as specified.

5.7.6 Label Constants

```
label:constant = identifier;
```

The identifier must not appear in a DECLARE statement; it must appear as a label exactly once in the function, i.e., at the beginning of a statement and followed by a colon.

5.7.7 Constant Expressions

The compiler will evaluate any expression consisting entirely of constants and standard functions and thus will treat it like a single constant.

5.8 Data Formats

The formats of the various kinds of values (i.e. the binary representations) are in great part determined by the hardware of the machine. We summarize them here for completeness, and to specify a few conventions established by SPL. Refer to the CPU manual for the exact word layouts.

Integers are 24-bit two's complement.

Longs are 48-bit quantities. No operations are defined on them except general ones for moving and decomposing any data object.

Longlongs are 96-bit, but otherwise identical to longs.

Reals are 48-bit: sign, 11-bit exponent and 36-bit fraction.

Double precision real numbers are 96-bit; the format is identical to that for reals, except that the fraction is 84-bit.

Complex numbers are 96-bit and consist of two reals. The real part is the first, the imaginary, the second.

Strings are four-word (96-bit) objects called descriptors. Each word is a hardware string indirect address word:

Bits	Function
Ø-1	=2, to specify a string descriptor
2-3	byte size: Ø=6-bit, 1=8-bit, 2=12-bit, 3=24-bit
4-5	byte number in word, counting from left
6-23	word address

The four words are used as follows:

Ø: start of string, points to byte before first byte of string

1: reader pointer, points to last byte read

2: writer pointer, points to last byte written

3: end of string, points to last byte available in string

Labels use the hardware's BLL descriptor, which is too complex to be described here. Functions are represented by pointers to their BLL descriptors.

Fields use the hardware's field descriptor, which is a one word object with the following form:

Ø-1	=1, to specify a field descriptor
2	set for SIGNED field
3-7	length in bits
8-12	bit address of first bit
13-23	word displacement

Arrays use the hardware's array descriptor, which is a two-word object with the following form:

Ø : Ø-1	=3, to specify an array descriptor
Ø : 2	lower bound (Ø or 1) on subscript
Ø : 3	set for marginal index descriptors (see below)
Ø : 4	large element bit
Ø : 5-6 or 5-1Ø	multiplier (element size)
Ø : 7-23 or 11-23	upper bound on subscript
1 : 6-23	address of first word of array

Array of dimension >1 are handled by marginal indexing; see the discussion of arrays in Section 6.1.5.

Intrinsic functions will exist to decompose and construct all these descriptors.

5.9 Function Declarations

The syntax is

```
function:statement = ftype ("FUNCTION"/ "ENTRY")
                     identifier "("  [declare:clause:
                     list]  ")" ["," "FRETURN"] [function:
                     location];
ftype              = ntype / "STRING";
function:location  = "," ("MONITOR" / "UTILITY" / "POP" /
                     "TRAP'ENTRY" / "FTRAP'ENTRY" /
                     "SP'ENTRY" / "SYSPOP") ["←" expression];
```

For example

```
FUNCTION F (I, REAL J, STRING ARRAY K);
```

ENTRY and FUNCTION are synonyms.

5.9.1 Formal Parameters

The declare:clause:list must not include lengths or forms. It may include dimensions, but only the number of subscripts is counted, not the values, and the subscripts may be null (e.g. A[,] for a matrix). Arrays are assumed to have one subscript if no dimension appears.

Any identifiers in the declare:clause:list which have not already been declared are declared as though they had appeared in a DECLARE statement with the same attributes. If any such identifier has already been used, an error comment results. For each identifier which has already been declared either:

1) the attributes specified for it in the function declaration must exactly agree with the attributes already declared for it, or

2) no attribute specifiers may precede it in the declare:clause:list.

Otherwise there will be an error comment.

The identifiers in the declare:clause:list constitute the formal arguments in the order in which they are written. When the function is called (see "function calls" below) an equal number of actual parameters must be supplied, and they must agree in type and mode. No automatic conversions are done. The agreement is checked when the call occurs.

5.9.2 FRETURNs

The FRETURN clause must be included if the function returns with FRETURN. In this case it must always be called with a failure clause. If any function in a program block has a FRETURN, the first one must.

5.9.3 Special Entry Points

The function:location specifies that the function is to be entered in one of the system-defined transfer vectors at the location specified by the expression. In the case of

POP, SPL will supply a location if none is specified. The possibilities are:

POP	the function is to be callable as a POP
TRAP'ENTRY	the function is to be called when the specified (ring-dependent) hardware trap occurs.
SP'ENTRY	the function is to be called when the sub-process in which it runs is entered at the specified entry point.

The remaining ones are of interest only to system programmers:

FTRAP'ENTRY	the function is to be called when the specified (fixed) trap occurs.
MONITOR	the function is to be called when the specified MCALL is executed. These two make sense only if the function is in the monitor.
UTILITY	the function is to be called when the specified UCALL is executed. This makes sense only if the function is in a utility. If any function in a program block has a MONITOR or UTILITY function:location, the first one must have it.

SYSPOP the function is to be called when the specified syspop is executed.

The following tables summarize the treatment of the various special kinds of entry points.

Type of function	Call with	Return with	Put descriptor
Ordinary	BLL	BLL	--
MONITOR	MCALL	BLL	MCALL Transfer Vector
UTILITY	UCALL	BLL	UCALL TV
POP	Pop	BLL	POP TV
TRAP'ENTRY	Not applicable. A trap'entry is not really a function. It does not have arguments. The address of the first word of code should be put into the TRAP TV. It is a programming error to reference any local variables or do a return.		
FTRAP'ENTRY	As for TRAP'ENTRY, but put the address of the first word into the FTRAP TV.		
SYSPOP	As for TRAP'ENTRY, but put the address of the first word of code into the TRAP TV at 20B + syspop number.		
SP'ENTRY	SP'CALL	SP'RETURN	SP TV
	A sp'entry is not really a function. It does not have arguments. It is a programming error to reference any local variables. The only proper way to call an sp'entry is with an sp'call and to return with an sp'return.		

Table 5.1 Summary of Function Call Conventions

Name of TV	Location and contents of descriptor	Contents of TV entry
MCALL	6Ø4ØØØB; UB=MAXMCALL	Absolute address of function descriptor. Initialized to an error function.
UCALL	4Ø3Ø14B; UB=MAXUCALL	As for MCALL.
POP	G'[Ø]; UB=MAXPOP	As for MCALL.
TRAP	G'[6]; UB=12B except for user ring, where UB=20B+MAXSYSPOP	Absolute address of code. Initialized to a trap routine.
FTRAP	6Ø4ØØ2B; UB=14B	As for TRAP
SP	G'[12B]; UB=MAXSP	As for MCALL

All descriptors are normal IAWs with indirect addressing; they point to ARRAY IAWs with LB=Ø (LB=1 for TRAPS), MULT=1, BASE=indexed indirect source-relative pointer to the transfer vector, which is allocated in code space at the discretion of the compiler.

The MAX symbols are, for the moment, built into the compiler with the following values:

MCALL = 400B, UCALL = 400B, POP = 100B, SYSPOP = 100B, SP = 20B.

Table 5.2: Transfer Vectors

5.10 Macros

The language allows a simple form of token-substitution macro. A macro is defined by a

```
macro:statement = "MACRO" macro:name ["(" formal:list ")"]
                  "←" macro:body;

macro:name      = identifier  ;

formal:list     = [formal $("," formal)]  ;

formal          = identifier  ;

macro:body      = compact:token:string;

compact:token:
  string        = <arbitrary string of tokens not including ";">
```

Once a macro:name has been defined (i.e. has appeared in a macro:statement) it can only be used in a macro:call. A macro:call may appear anywhere except in a string or character constant. It is

```
macro:call      = macro:name ["(" actual:list ") "]  ;

actual:list     = [actual$("," actual)]

actual          = balanced:token:string

balanced:token:
  string        = <compact:token:string balanced with
                  respect to parentheses, and not including
                  "," except in parentheses, or carriage return>;
```

The actual:list must be present if and only if the formal:list was present in the macro:statement, and must be of the same length as the formal:list. The macro:call is replaced by the macro:body, except that each occurrence of a formal in the macro:body is replaced by the corresponding actual. The result is then scanned again for further macros.

Macros in a macro:body are expanded at definition time (unless they have not yet been defined, in which case they are expanded at call time according to the rescanning rule stated above). If expanded at call time, their actuals must not include any formals.

Note that a macro is expanded strictly by token substitution: there is no requirement that any of the token strings involved make syntactic or semantic sense.

6. Action:Statements

An action:statement is defined by:

action:statement = expression/

if:statement/elseif:statement/endif:statement/

for:statement/endfor:statement/

"." assembler:statement/

6.1 Expressions

This section provides the following information about SPL expressions:

approximate syntax, based on the precedence of the operators

exact syntax

rules for types of operands

the semantics of the various operators

6.1.1 Precedence of Operators

Expressions are made up of operators and operands. The operators, in order from low to high precedence, are:

Operator	Meaning
FOR WHILE	loops
IF ELSE	conditionals
WHERE	sequential evaluation
&	sequential evaluation
RETURN FRETURN	function returns
OR	boolean "or"
AND	boolean "and"
NOT (unary)	boolean "not"
= # > >= < <=	relations
← (on right)	assignment
MOD	modulo or remainder
+ - V' E'	add, subtract, logical or, logical exclusive or
* / LSH RSH LCY RCY A'	multiply, divide, left shift, right shift, left cycle, right cycle, logical and
**	exponentiate
+ - N' (unary)	unary + -, logical not
GOTO	transfer
← (on left)	assignment
. $ @	field operations
$ @ (unary)	indirection, reference
[] ()	subscripting, function call

The operands are:

constants

names

parenthesized expressions

6.1.2 Syntax of Expressions

The section 6.1.1 Operators by Precedence List, while convenient for quick reference, does not suffice to specify the syntax of expressions. We therefore state the complete syntax; explanations of the meaning of the operators follow in section 6.1.3:

```
expression      =  forexp;
forexp          =  ifexp $( "FOR" forclause / "WHILE" ifexp);
forclause       =  identifier "←" remainder ([","
                   alternation] "WHILE" ifexp / ["BY"
                   ifexp] ["TO" ifexp]) ;
ifexp           =  whrexp ["IF" whrexp ["ELSE" ifexp]];
whrexp          =  catexp ["WHERE" whrexp];
catexp          =  retexp $("&" retexp);
retexp          =  alternation / ("RETURN" / "FRETURN")
                   (alternation / "(" ifexp $("," ifexp)
                   ")");
alternation     =  conjunction $("OR" conjunction) /
                   "GOTO" tailing;
conjunction     =  negation $("AND" negation);
negation        =  ["NOT"] relation;
relation        =  assignment [("=" / "#" / ">" / ">=" /
                   "<" / "<=") assignment];
```

```
remainder          =  sum $( "MOD" sum);
assignment         =  remainder / a:tailing "←" assignment;
sum                =  term $(("+" / "-" / "V'" / "E'") term);
term               =  factor $(("*" / "/" / "LSH" / "RSH" /
                      "LCY" / "RCY" / "A'") factor);
factor             =  ["+" / "-" / "N'"] power;
power              =  tailing[  "**" factor];
tailing            =  a:tailing / v:tailing ;
a:tailing          =  indirection $(tail) / reference
                      $("." field) ;
v:tailing          =  reference $(tail) ;
tail               =  ("." / "$" / "@") field ;
indirection        =  1$("$") arrayref ;
reference          =  ["@"] arrayref / function:call ;
arrayref           =  a:primary $( "[" expression $(","
                      expression) "]" ) ;
function:call      =  a:primary / v:primary ;
a:primary          =  identifier / "(" a:tailing ")" ;
v:primary          =  constant / "(" expression ")" ;
```

Note: this grammar is ambiguous because a function:call can be parsed as both a:primary and v:primary. a:primary parsing will be used if possible.

6.1.3 Types of Operands

The various operations have various requirements for the types of operands permitted and the type of result produced. The permitted combinations are summarized in Table 6.1, in which the following conventions are used.

type abbreviations:	I	integer
	G	long or longlong
	R	real
	D	double
	C	complex
	S	string
	L	label
	U	unknown
other abbreviations:	F	suffix means mode = FIELD
	T	suffix means mode = FUNCTION
	Y	suffix means mode = ARRAY
	S	suffix means mode = SCALAR
	A	means any type
	N	means I, R, D or C (i.e. number)
	M	means I, R or D

Where A, N or M is suffixed with a digit, different digits indicate that different types may appear. See, for example, "WHERE": argument 1 and argument 2 may be different types. If the digits are the same, or there is no digit, the types must be the same.

Note: A partial ordering on the numeric types is defined: $I<R<D$, $R<C$. Where two Ns or Ms appear, the lower is converted to the higher

before the operation is evaluated. If the result is N or M, it has the higher type also. See, for example, "+". If NS1 were "I" (integer) and NS2 were "R" (real), then NS1 would be converted to "R" and the result would be "R". It is illegal to have one D argument and one C argument. Where A appears, the mode is free except as fixed by suffixes. In all other cases mode = SCALAR.

Constants receive special treatment. Any type N constant is automatically converted to a higher type if that is required for an assignment to be legal. This is not done for variables; the explicit transfer functions described in the definition of action:statement at the start of Section 6 must be used.

An object of type U may be used where A appears in Table 6.1. It may also be used as one of the operands in the lines marked *, in which case it is assumed to have the type of the other.

Note the treatment of ARRAYs, FIELDs and FUNCTIONs of type ARRAY, FIELD or FUNCTION. When such variables are applied to subscripts, pointers or function arguments, they yield results of type UNKNOWN and mode given by their type. Normally such results must be assigned to something of known type before they can be used because of the restrictions on the use of type UNKNOWN; thus, for example, if we want A to be an ARRAY of REAL FUNCTIONs we would write

```
DECLARE FUNCTION ARRAY A, REAL FUNCTION RA;
   :
RA ← A[I];
RA(X,Y + 5);
   :
```

ARG1	OPT	ARG2	RESULT	NOTES
A	IF	I ELSE A	A	The A's are required to be the same only if the value of the IF is used.
A1	WHERE	A2	A1	
A1	&	A2	A2	
*IS	OR	IS	IS	
*IS	AND	IS	IS	
-	NOT	IS	IS	
*NS1,AS	=,≠	NS2,AS	IS	
*MS1	<,<=,>,>=	MS2	IS	
*A	←	A	A	
*MS1	MOD	MS2	MS	
*NS1	+,-,*,/	NS2	NS	
*NS1	**	NS2	NS	but see details below
*IS	SHIFT, A',E',V'	IS	IS	
-	N'	IS	IS	
-	+,-	NS	NS	
-	GOTO	LS	-	
IS	.	AF	AS**	
AS	$	IF	IS	
IS	@	IF	IS	
-	$	IS	US	
-	@	A	IS	
AY	[IS...,	IS]	AS**	
IS	[IS]		US	
AT	(A2, ..., An)		AS**	

*: one operand may be U, and is assumed to have the type of the other.

**: if A is ARRAY, FIELD or FUNCTION, the result is type U, mode A.

Table 6.1 Permitted Operands and Type of Result

6.1.4 Semantics of Expressions

We now complete the discussion of operations with comments on the evaluation of each one, together with some remarks which may clarify the syntax and type conversion rules given above. The operands are referenced by the symbols which stand for them in the expression schemata on the left.

FOR,WHILE	are discussed in Section 6.2.2
A1 IF I ELSE A2	evaluates I. If I ≠ Ø, evaluates A1 and returns its value, otherwise evaluates A2 and returns its value, or returns Ø if the ELSE is missing. Typical usage is: F(X) IF X < 4 ELSE G(X) IF X< 5 ELSE H(X); Note that: X ← Y IF Y < 3 ELSE Y+1 ; alters X only if Y < 3. Therefore write: X ← (Y IF X < 3 ELSE Y+1) ; if this is intended.
A1 WHERE A2	evaluates A2, then evaluates A1 and returns its value.

A1 & A2	evaluates A1, then evaluates A2 and returns its value. Several &'s may be strung together.
RETURN, FRETURN	See Section 6.1.6
I1 OR I2	evaluates I1, returns 1 if it is ≠ Ø. Otherwise evaluates I2, returns 1 if it is ≠ Ø, otherwise returns Ø.
I1 AND I2	evaluates I1, returns Ø if it is =Ø. Otherwise evaluates I2, returns Ø if it is = Ø, otherwise returns 1.
NOT I	evaluates I and returns 1 if I = Ø, otherwise returns Ø.
A1 (=,#,>,>=,<,<=)A2	*evaluates A1 and A2 and then evaluates the relation. The value is Ø if the relation does not hold, 1 if it does. Note that only = and # are legal on non-M types.
A1 ← A2	evaluates A2 and stores the resulting value into A1. They must agree in type and mode except for the special treatment of constants, and except that one operand may be of type U.
M1 MOD M2	*evaluates M1 and M2, and returns M1-FIX(M1/M2)*M2
N1(+,-,*,/) N2	*obvious
I1(A',V',E')I2	*compute the bitwise "and," "or," or "exclusive-or" of the operands
I1(LSH,RSH,LCY,RCY) I2	*these are 24-bit logical shifts (shift in Øs) or cycles

* see end of table.

N1 ** N2	*obvious, except that I1 **I2 is an error unless I2 is positive. The error is not caught until runtime if I2 is not a constant.
(+,-)N	obvious.
N' I	computes the bitwise (1's) complement of I.
GOTO L	sends control to the statement labeled by L. If this was passed as a parameter, the correct environment is restored.
I. AF	*evaluates I, takes it as an absolute address A, and references the bits of A + word:displacement (AF), from starting:bit (AF) to ending:bit (AF). The result may appear on either side of an assignment. If the field is SIGNED, the starting bit is copied into all the higher bit positions when the result is used as a value; otherwise these positions are filled with zeros.
A $ IF	*references the bits of the value of A specified by IF. The word displacement of IF should not be greater than the number of words in the value of A. (4 at most if A is a variable). Sign extension is handled as for "." above.
I @ IF	*returns T, where T is the result of: T ← Ø T$IF ← I i.e., the value of I positioned in a word according to the field IF.

* see end of table.

$I	references the value addressed by the value of I taken as a hardware indirect word. Normally the top 6 bits of I should be off, since the hardware uses them to select the type of indirection rather than to specify the address. The value is of type U.
@A	returns the address of the value of A. It makes sense (and is legal) only if A can appear on the left of ← or is a label constant.
AY[I,...,I] IS[I]	references the element of the array A specified by the subscript I, as described in Section 6.1.5. If the first operand is IS, only 1 subscript is allowed. This construct is equivalent to (IS + I).WØ, where we have declared FIELD WØ(Ø).
A1(A2,...,A2)	returns the value of the function A1 after calling it with parameters A2, as described in Section 6.1.6.
AF(I)	equivalent to I.AF

An * preceding the description means that the order of evaluation of simple operands (see Section 6.1.6.1) is undefined. Compound operands are always evaluated left-to-right if there are more than one. If the * is lacking, the operands are always evaluated left-to-right.

6.1.5 Array expressions

Arrays of any dimensionality from 1 to 7 are allowed. If the array has n dimensions, then a reference to it with n integer subscripts yields a scalar of the same type, unless the type is ARRAY, FIELD or FUNCTION. In this case the type of the result is UNKNOWN and its mode is the type of the array. Thus, after declaring:

1) INTEGER I, J, K, REAL ARRAY A[3, 4, 5]

we know that:

2) A [I, J+1, K**2]

is a REAL SCALAR. It is also possible to write:

3) A [I, J+1]

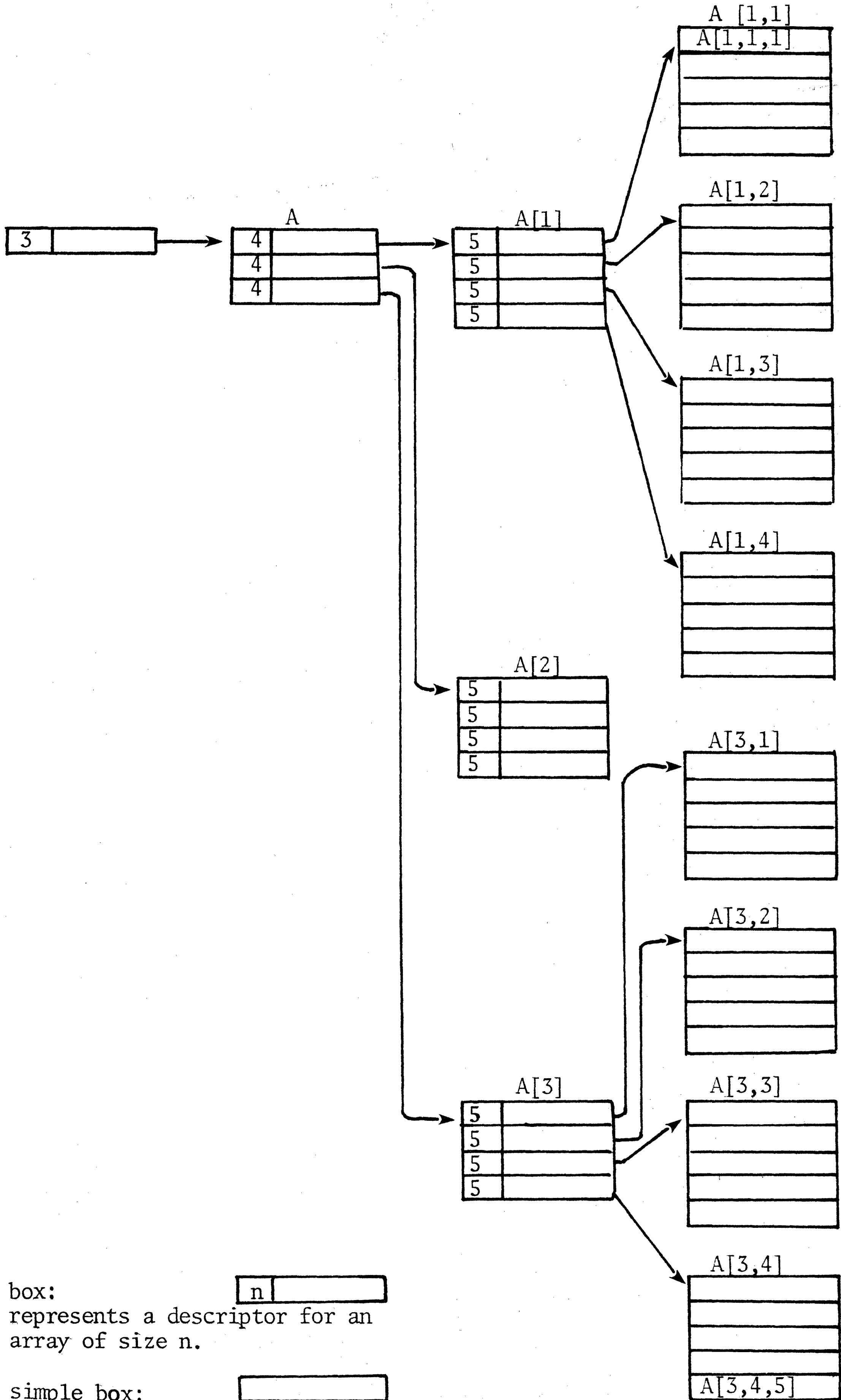
which is an UNKNOWN ARRAY. If it is assigned to the REAL ARRAY B, then:

4) B [K**2]

references the same scalar referenced by (2). It is probably not useful to do anything with an UNKNOWN array except to assign it to something.

Marginal indexing is used to access arrays. In the above example (1), the value of A is a descriptor for an array with three entries. Each entry of this array is a descriptor for an array with four entries. Each entry of the four-entry array is a descriptor for an array with five entries, each of which is a real number. The Figure 6.1.5 illustrates this. The 120 words allocated for the real numbers are contiguous in storage and in the order indicated. Note that Fortran arrays vary the first subscript first and are therefore incompatible.

FIGURE 6.1.5



A box: represents a descriptor for an array of size n.

A simple box: represents a real number

6.1.6 Function call and return expressions

The syntax for a function call is

```
a:primary "(" [expr $("," expr)] [stores]
   ["//" failure:result [stores]]")" ;
stores = ":" identifier $("," identifier) ;
failure:result = ["GOTO"] identifier / ("RETURN" /
   "FRETURN") [expr / expr:list] / "VALUE" expr ;
```

The a:primary must have mode=FUNCTION. The value of the function is taken to be a SCALAR of type equal to the type of the a:primary, unless this type is ARRAY, FIELD or FUNCTION. In the latter case, the type of the result is UNKNOWN and its mode is given by the type of the function.

6.1.6.1 Arguments

The arguments immediately follow the function name. There is no restriction on their number or type, except that an initialized LOCAL label or string array may not be used.

```
F(); F(X); F(X,Y(1,2),Z[3]**5,W,Q);
```

are function calls with Ø, 1 and 5 arguments respectively. Arguments are evaluated as follows. All the arguments which are compound are evaluated, left to right, and their values are saved. An argument is simple if it is one of the

following, compound otherwise:

    constant
    identifier
    identifier "[" identifier "]"
    identifier "." field
    "$" identifier
    "S" identifier "." full-word field

The value of each argument is then stored in the corresponding formal argument of the function being called. No type conversion is done; nonmatched types are a run-time error. Control is then passed to the function.

6.1.6.2 Returns

A return is done with an expression of the form:

    RETURN (expr, expr,..., expr)/
    RETURN expr/
    RETURN;

The expression list is treated exactly like the actual parameter list of a function call. The value of the first expression becomes the value of the function; it and subsequent values are stored in the corresponding identifiers following the ":" in the call, exactly as actual parameter values are stored in formal parameters.

6.1.6.3 Failure Exits

If a failure exit is provided following the "//" in the call, an FRETURN will send control as specified in the failure exit. It may be a label, in which case control goes to that address; a RETURN, in which case a return is made from the function containing the failure exit; or a VALUE expression, in which case the value of the expression becomes the value of the function. Just as for the RETURN, any number of values may be returned; they are stored in the corresponding local:identifiers following the ":". When a function has a failure exit the normal or success return is with RETURN, exactly as for a function with no error exit.

6.2 Expressions used as Statements

In order to catch some common errors in which the user inadvertently writes an expression statement which does nothing, a set of rules is enforced. They insure that evaluation of the expression results in some change in the state of the world; such an expression is called an _action expression_. Here the principal operator is the one of lowest precedence (i.e. first on the list in the Section 6.1.1 on "Precedence of Operators"), except that any operator enclosed in n sets of parentheses is of higher precedence than any operator enclosed in fewer than n sets of parentheses.

An expression is an action expression if the principal operator is:

a) ←, GOTO, RETURN, FRETURN, or a function call

b) WHERE, &, FOR, WHILE, IF

If the operator is in group (b) then:

a) for "WHERE" or "&", both operands are action expressions,

b) for "FOR" or "WHILE", the body (first operand) is an action expression,

c) for "IF/ELSE", both of the consequents are action expressions, or the second consequent is missing.

6.2.1 IF statements

We have seen that IF can be used as an operator. It can also be used in the following way:

```
IF expression DO;
   ...
ELSEIF expression DO;
     ...
    .
    .
    .
```

```
ELSE DO;
   ...
ENDIF;
```

Any number of ELSEIFs are allowed. The "..." may be replaced by any sequence of statements balanced with respect to IFs and FORs. The ELSE may be omitted. The integer expressions after the IF and ELSEIFs are evaluated in turn until a non-zero one is found. The statements between it and the next ELSEIF, ELSE or ENDIF are then executed, and control goes to the statement following the ENDIF. THE ELSE DO is equivalent to ELSEIF 1 DO. If none of the expressions are non-zero, nothing is done. It is good practice to indent the statements represented by "..." uniformly 3 or 4 spaces for clarity.

6.2.2 FOR statements

The same thing can be done with "FOR" as with "IF"

```
for:statement;
   ...
ENDFOR;
```

Here we have

```
for:statement = ("FOR" for:clause / "WHILE" expression)"DO";
for:clause = identifier "←" (expression1["BY" expression2]
  ["TO" expression3] / expression1["," expression2]
  WHILE" expression3);
```

If the BY/TO form is used, the identifier must be of type M. If BY is omitted, BY 1 is assumed. If TO is omitted the loop can only terminate by an explicit transfer out.

The effect is that the statements represented by ... are executed repeatedly for successive values of the controlled variable. In the first case the variable starts at expression1. On each successive loop expression2 is added until the variable passes beyond expression3. The definition of "beyond" depends on the sign of expression2. If expression1 is beyond expression3, the loop body will not be executed at all. If the expressions are compound (see Section 6.1.6 on "Function Calls"), they are evaluated before the loop starts; if simple, then each time around.

The second form initializes the controlled variable for expression1. Then it tests integer expression3. If it is Ø, control passes beyond the ENDFOR. Otherwise the loop body is executed, the value of expression2 (or expression1 if expression2 is omitted) is assigned to the variable, and the test is made again. The expressions are re-evaluated each time around the loop.
A WHILE statement simply loops until the integer expression is Ø, without modifying anything.

When "FOR" or "WHILE" is used as an operator, exactly the same facilities are provided. The first argument is evaluated each time around the loop. The value is undefined. Thus

```
A[I,J] ← Ø FOR I←1 TO N FOR J←1 TO M;
```

would initialize the array A.

6.3 Assembly Language

An assembler:statement consists of one or more machine instructions according to the following syntax:

```
assembler:statement    =  "." machine:instruction $(','
                            ["."] machine:instruction);
machine:instruction    =  opcode [address];
opcode                 =  identifier / simple:integer;
address                =  expression;
```

Since opcodes appear in a restricted context, the symbols used for opcodes in the CPU Manual (which are all recognized by SPL) may be used freely for other purposes as well. If an opcode is an identifier and not predefined, it must be an INTEGER constant. Such opcodes, as well as opcodes which are written as integers, are treated as follows: if no address appears, the value of the opcode is placed directly in the compiled program; if an address does appear, bits 18-23 of the opcode value are placed in bits 3-8 of the instruction word and bit 17 of the value is placed in bit 9 (the programmed operator bit).

Any expression may appear as an address as long as it is logically equivalent to either a constant (of any type and mode) or one of the addressing formats of the CPU. These

formats are described in detail in the CPU Manual and are listed below, together with the usual way of generating them. Note the existence of the four reserved symbols X', L', G', and R'.

Addressing format	Normal syntax
D	G'[N]
I	$G'[N]
X	X'[N]
PD	P[N]
PDI	$P[N]
BX	A[I]
BXD	($X')[I+N]
IM	N
IMX	X'+N
SR	R'[N]
SRI	$R'[N]
LR	L'[N]
LRI	$L'[N]

In the above list, N stands for an INTEGER constant quantity, P and I stand for INTEGER SCALAR quantities, and A stands for an ARRAY quantity. BX or PD addressing may also result from tailing. Since the determination of the addressing format is done on semantic, not syntactic, grounds, the exact rules are quite complex.

7. Intrinsic Functions

Figure 7.1 lists all the intrinsic functions in SPL.

An intrinsic function is one which:

1.) Is recognized by the compiler without the need for any declaration by the user;

2.) May have default argument values automatically supplied by the compiler;

3.) Has the types of its arguments checked at compile time;

4.) May compile into special open code.

In figure 7.1, default values for arguments which the user is allowed to omit are given in parentheses after the argument type. For all functions which have failure returns, a routine which prints an error message and causes a sub-process trap will be supplied if the user fails to specify a failure action.

The remainder of this section describes each intrinsic function in detail. Type letters with subscripts will be used to refer to the arguments of a function: e.g. the arguments of CNS will be referred to as I_1,S_2,I_3, and I_4.

NAME	ARGUMENT TYPES	RESULT TYPE	FRETURN?	OPEN CODE?
FIX	R	I		X
ENTIER	R	I		X
FLOAT	I	R		X
DFLOAT	I	D		X
RE	C	R		
IM	C	R		
CSN	S,I(1Ø)	I	X	
CSR	S	R	X	
CSD	S	D	X	
CNS	I,S,I(Ø),I(1Ø)	S	X	
CRS	R,S,I(Ø)	S	X	
CDS	D,S,I(Ø)	S	X	
INCDES	I,I	I		X
LNGDES	I,I	I		X
GCI	S	I	X	X
WCI	I,S	I	X	X
GCD	S	I	X	X
WCD	I,S	I	X	X
SETUP	S,I,I,I(8)	S		X

I = integer C = complex

R = real A = array

S = string D = double

Figure 7.1 List of intrinsic functions

NAME	ARGUMENT TYPES	RESULT TYPE	FRETURN?	OPEN CODE?
SETS	S,I(Ø),I(Ø)	S		X
SETR	S,I(Ø)	S		X
SETW	S,I(Ø)	S		X
LENGTH	S	I		X
SCOPY	S,S	S	X	
APPEND	S,S	S	X	
GC	S	I		X
STORINIT	I,I	I	X	
MAKE	I,I(Ø)	I	X	
SETZONE	I	I	X	
SETARRAY	A	I	X	
FREE	I,I(Ø)	-	X	
EXTZONE	I,I	-	X	
FREEZONE	I,I(Ø)	-	X	
BCOPY	I,I,I(-1)	-		X
BSET	I,I,I(-1)	-		X

I = integer   C = complex

R = real   A = array

S = string   D = double

Figure 7.1 (continued)

7.1 Type Conversion Functions

FIX(R_1) converts R_1 to an integer by truncation towards zero.

ENTIER(R_1) converts R_1 to the nearest integer.

FLOAT(I_1) converts I_1 to single-precision floating point.

DFLOAT(I_1) converts I_1 to double-precision floating point.

The four operators above are converted directly into machine instructions. For details consult the part of the CPU Manual which deals with handling of floating point numbers.

RE(C_1) gives the real part of C_1 in single-precision floating point.

IM(C_1) gives the imaginary part of C_1 in single-precision floating point.

CSN(S_1,I_2//F) expects to find an integer as the beginning of S_1, with syntax ['+' / '-'] 1$<digit in base I_2>. Digits above 9 are allowed if I_2>1Ø: the next digit after 9 is A, and so on. I_2 is taken as 1Ø if not supplied. CSN returns the integer, which it reads off the string, advancing the reader pointer so that the next character to be read is the non-digit which ends the integer, or to

the end of the string. CSN fails if S_1 does not begin with an integer in the proper format, leaving the reader pointer unchanged.

CSR(S_1//F) expects to find, at the beginning of S_1, a real number in any of the formats allowed by SPL for REAL quantities. It returns a single-precision floating point number. Otherwise the action is the same as for CSN.

CSD(S_1//F) is the same as CSR except that it returns a double-precision floating point result. Either SPL REAL or DOUBLE syntax is acceptable; in the former case, the number is accumulated in double precision.

CNS(I_1,S_2,I_3,I_4//F) converts the value of I_1 to a string of characters, which it appends to S_2. The radix is I_4, assumed to be 1Ø if omitted. If bit Ø of I_3 is on, I_1 is converted unsigned (e.g. -2 will appear as 77777776 in radix 8); otherwise, a '-' precedes the converted absolute value if I_1 is negative. Bits 18-23 of I_3 give the number of characters to generate: enough blanks are written before the converted value to give the total number of characters required. If the converted value does not fit into this many characters, it is truncated on the left with no error indication. If the character count is Ø, the converted value is neither padded nor truncated. I_3 is taken as Ø (signed, no formatting) if omitted. CNS fails only if there is insufficient room to write the necessary number of characters onto S_2: leaving the writer pointer is unaffected.

CRS(R_1,S_2,I_3//F) appends the converted value of R_1 to S_2. The failure condition is the same as for CNS. I_3 specifies the format in some as yet unspecified way; $I_3 = \emptyset$ is assumed if I_3 is omitted, results in some reasonable unformatted conversion.

CDS(D_1,S_2,I_3//F) is exactly like CRS except that the converted value is in DOUBLE rather than REAL format.

7.2 String Functions

In this section the following abbreviations are used: BP = beginning pointer, RP = reader pointer, WP = writer pointer, EP = end pointer. These correspond to the 4 words of an SPL string descriptor, in order.

INCDES(I_1,I_2) assumes that I_1 is a character pointer (hardware string indirect word), such as one of the 4 words in an SPL string descriptor. The value is I_1 incremented by I_2 character positions. See CPU Manual for the exact specification of this operation, which is done with the ASP instruction.

LNGDES(I_1,I_2) assumes that I_1 and I_2 are both character pointers. It returns the length of the string which they bracket. See the CLS instruction in CPU Manual for details.

GCI(S_1//F) fails if S_1 is empty, i.e. RP = WP. Otherwise it increments RP by one character position then returns the character pointed to by RP.

WCI(I_1,S_2//F) fails if S_2 is full, i.e. WP = EP. Otherwise it increments WP then writes I_1 at the character position pointed to by WP. The value of WCI is I_1.

GCD(S_1//F) fails if S_1 is empty. Otherwise it returns the character pointed to by WP and then decrements WP.

WCD(I_1,S_2//F) fails if S_2 is initialized, i.e. BP = RP. Otherwise it writes I_1 at the character position pointed to by RP and then decrements RP. The value of WCD is I_1.

SETUP(S_1,I_2,$I_3 I_4$) puts into S_1 a string descriptor for a string of I_2 characters starting with the first character of the word pointed to by I_3. The character size is I_4, defaulting to 8. The value of SETUP is the string descriptor it creates. If I_3 is omitted, MAKE is called to assign space. BP = RP = WP in the created string descriptor.

SETS(S_1,I_2,I_3) is exactly equivalent to SETW(S_1,I_3) followed by SETR(S_1,I_2): see below. I_2 and I_3 are taken as Ø if omitted.

SETR(S_1,I_2) sets S_1's RP to point I_2 characters beyond BP. IF $I_2 < Ø$, it is taken as Ø; if $I_2 >$ LNGDES(BP,WP), it is taken as this quantity; if I_2 is omitted, it is taken as Ø. The effect is that the RP remains between the BP and the WP.

SETW(S_1,I_2) sets S_1's WP to point I_2 characters beyond BP. There are four cases: $I_2 < Ø$ leads to WP←RP←BP; $Ø \le I_2 <$ LNGDES(BP,RP) leads to WP←RP←INCDES(BP,I_2); LNGDES(BP,RP) $\le I_2 \le$ LNGDES(BP,EP) leads to WP←INCDES(BP,I_2); and $I_2 >$ LNGDES (BP,EP) leads to WP←EP. Again, the effect is to guarantee the correct order of BP, RP, WP, and EP.

LENGTH(S_1) gives the number of GCI's that can be done on S_1 without failing, i.e. LNGDES(RP,WP).

GC(S_1) returns the character pointed to by RP. This is garbage if S_1 is empty, but no check is made.

SCOPY(S_1,S_2//F) copies the string S_2 into the string S_1. S_2 is not affected; for S_1, RP←BP and WP←INCDES (BP, LENGTH(S_2)). Failure occurs only if there is not enough room in S_1 and no pointers are affected.

APPEND(S_1,S_2) appends the string S_2 to the string S_1, advancing S_1's WP by LENGTH(S_2). The failure condition is the same as for SCOPY.

7.3 Storage Allocation Functions

There is a standard mechanism for allocating and releasing arbitrary-sized blocks of storage in arbitrary order called the storage allocator. It is driven by the following standard functions:

STORINIT (I_1, I_2) initializes the storage allocator to use an area of storage beginning at I_1 and occupying I_2 number of words for its machinations. It is not necessary to call STORINIT; a standard area will be reserved if STORINIT has not been called when the first request is made for a block. The value of STORINIT is a pointer to the zone just created; this pointer is also put into the predeclared global pointer variables INFINITY'ZONE and CURRENT'ZONE.

MAKE(I_1, I_2) creates a block of storage of I_1 words and returns a pointer to it. An extra cell is assigned by the system; it immediately precedes the block and contains the length in the bottom 18 bits and flags in the top 6. The cell is for system use only. Space is normally allocated directly from the area specified by STORINIT (or the standard default area); this area is called the infinity zone. The user may set up zones of his own; for example, if he wishes to create some fairly complex temporary structure and then delete it in its entirety,

it is more efficient to create it in a separate zone and then release the entire zone. I_2, which is optional, is a pointer to a user zone; if it is omitted, the zone pointed to by CURRENT'ZONE is used. CURRENT'ZONE is set by the function SETZONE(I_1) which provides compatibility with the (hardware) storage allocator. A user zone is created by the function STORINIT.

SETARRAY(A_1) which makes the space occupied by the array A_1 into a new zone by setting up some machinery inside it. CURRENT'ZONE is not set by this function. Blocks are released by the function FREE.

FREE(I_1,I_2) where the block pointed to by I_1 must fall within the zone optionally given by I_2. When a zone is full, i.e., a call on MAKE finds insufficient space, an overflow function is executed. The address of the descriptor for this function is in word 1 of the zone; it is initialized to a system error routine when the zone is created. The user, of course, may change it at any time. The function receives the arguments of MAKE as its arguments. Frequently the proper course of action is to acquire additional space and attach it to the zone: this is done by EXTZONE.

EXTZONE(I_1,I_2) adds all the space in the block I_2 to the zone pointed to by I_1. When a zone reaches the end of its usefulness, all the space occupied by the zone must be released using the function FREEZONE.

FREEZONE(I_1,I_2) releases all the space (including extra extensions) occupied by the zone I_1 into the zone I_2. If the extensions were allocated out of more than one zone, the user must release them individually with FREE.

7.4 Miscellaneous Functions

BCOPY(I_1,I_2,I_3) copies I_3 words starting at I_2 to I_3 words starting at I_1. Copying is done in the appropriate direction (i.e. starting at the beginning or the end of the block) to ensure that no information is lost. If I_3 is omitted, I_2.SIZE is used, where FIELD SIZE (-1:6,23); this is where the storage allocator hides the block size. The intention is that I_3 should be omitted if the block pointed to by I_2 was created with MAKE, since other objects in SPL such as arrays and strings do not have this word.

BSET(I_1,I_2,I_3) initializes I_3 words starting at I_1 to the value I_2. If I_3 is omitted, I_1.SIZE is used as in BCOPY.

Appendix I
List of Keywords

A'
AND
ARRAY
ARRAYONE

BY

CHARACTER
COMMON
COMPLEX

DECLARE
DO
DOUBLE

E'
ELSE
ELSEIF
END
ENDFOR
ENDIF
ENTRY
EXTERNAL

FIELD
FIXED
FOR
FRETURN
FTRAP'ENTRY
FUNCTION

G'
GOTO

IF
INCLUDE
INTEGER

L'
LABEL
LCY
LONG
LONGLONG
LSH

MACRO
MOD
MONITOR

N'
NOT

OCTAL
OR
ORIGIN

PARAMETER
POINTER
POP
PROGRAM

R'
RCY
REAL
RETURN
RSH

SCALAR
SIGNED
SP'ENTRY
STRING
SYSPOP

TO
TRAP'ENTRY

UNKNOWN
UTILITY

V'
VALUE

WHERE
WHILE

X'