bcc	title: SPL Reference Manual	SPL/M-1.2
Larry L Barnes / checked: L. Peter Deutsch / approved: Mel	authors: Butler W. Lampson	revision date: 11/3/69 / classification: Manual / distribution: Company Private / pages: 69

ABSTRACT and CONTENTS

Reference manual for the initial version of the Model I System Programming Language. The syntax and semantics of the language are defined. Nothing is said about the command language for the processor, which is described in CSED/M-12, SPLDS/M-17 and SPLEX/M-32.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

1. 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               =  program:block / common: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;
```


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.

1.1 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, special characters and the sequences "<=" ">=" "**" "//".

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.

1.2 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 B if B appears in the identifier:list of an include:statement in the block.

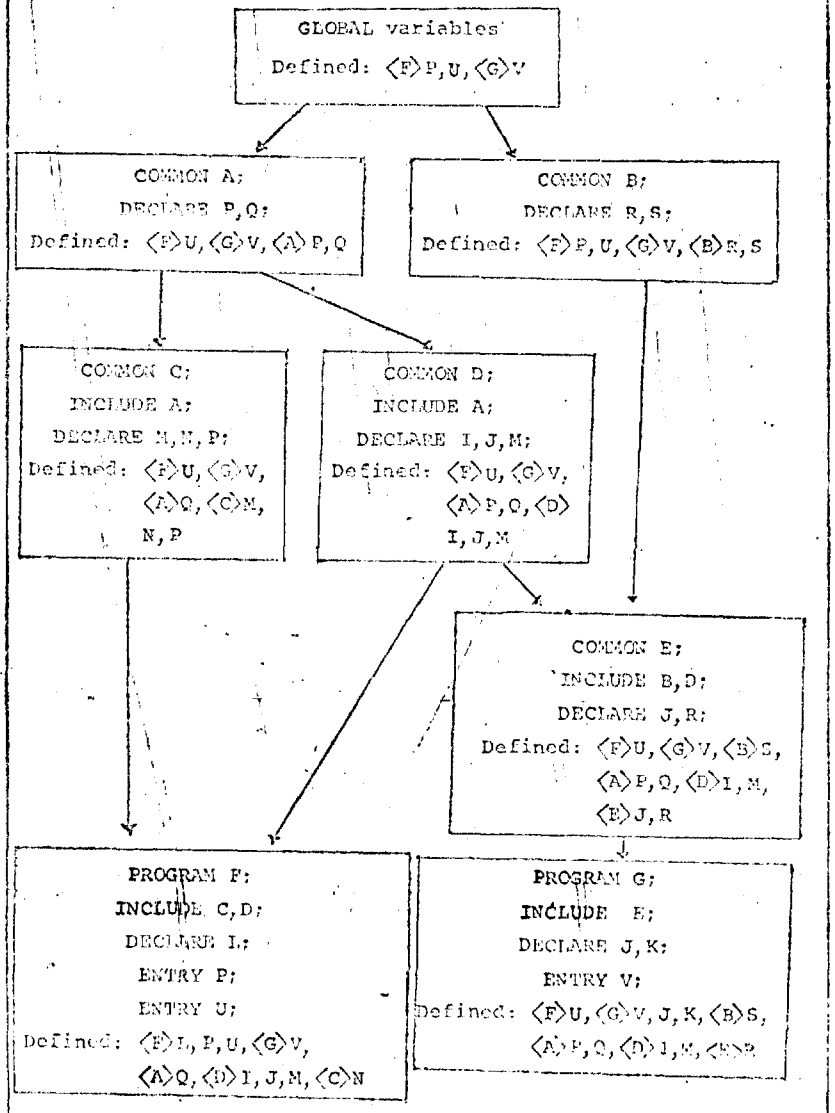
include:statement = "INCLUDE" identifier:list;

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 which contains no include:statements.

The effect of this convention is that declarations in COMMON blocks can be overridden by other declaration 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



⟨F⟩P means the P local to F

2. Declarations

2.1 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).

2.2 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 / "STRING" [length] ;

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

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

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

Note that FUNCTION ARRAY's and ARRAY FUNCTION's are both possible.

The type of scalar value determines its size: integer, function and field are one word; long, real; array, and label, two; string, double, longlong; and complex, four. 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 and so that the system can adjust the values of pointers when objects are moved around. It is important to declare as POINTER all integer variables which are to contain addresses during execution if it is desired to continue execution after modifying the program. 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 0), subscripted references to it will be compiled on the assumption that indirection through the descriptor with the subscript in IR 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- 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.

2.3 Attribute Modifiers

The shapes and sizes of things are specified by modifiers (dimensions, forms and lengths) 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" below for a discussion of what operands are regarded as constant.

2.3.1 Dimensions

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

Arrays of any dimensionality up to 7 are allowed. The 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" below 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 below under "Allocation."

2.3.2 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 "Allocation" for details.

2.3.3 Form

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

word:displacement = 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, 0 and 23 are used. A field may not cross a word boundary.

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

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

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

declare:statement = "DECLARE" declare:clause:list / macro:statement;

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

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

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 (LOCAL) or a COMMON block

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.

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.

2.5 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".

2.6 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 [-2000B, 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 in 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 promarily 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 FUNCTIONSs is done with a single name; otherwise the comments above apply. Initialization of FIELMs 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.

3 Constants

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

3.1 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 9-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.

3.2 Real Constants

```
simple:real:constant  =  digits "." $(digit) / "." digits
exponent              =  "E" sign digits;
```

```
sign            = ["+" / "-"];
real:constant   = simple:real:constant [exponent] /
                  digits exponent;
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.

3.3 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.

3.4 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.

3.5 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.

3.6 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.

3.7 Constant Expressions

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

4. 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 twos 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. Each word is in the form of a hardware string descriptor:

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
1:	reader pointer
2:	writer pointer
3:	end of 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:

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

Arrays of dimension >1 are handled by marginal indexing; see the discussion of arrays below.

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

5. 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"} ["=" expres-
                     sion];
```

For example

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

ENTRY and FUNCTION are synonyms.

5.1 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.

5.2 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.

5.3 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 occur.
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 moniter.
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	GRET	MCALL TV
UTILITY	UCALL	GRET	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	BLL	BLL	SP TV

Table 5.1 Summary of Function Call Conventions

Name of TV	Location and contents of descriptor	Contents of TV entry
MCALL	604000B; UB=MAXMCALL	Absolute address of function descriptor. Initialized to an error function.
UCALL	403014B; UB=MAXUCALL	As for MCALL.
POP	G'[0]; UB=MAXPOP	As for MCALL.
TRAP	G'[6]; UB=11B except for user ring, where UB=20B+MAXSYSPOP	Absolute address of code. Initialized to a trap routine.
FTRAP	604002B; UB=13B	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=0, 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

6. 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.

6.1 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 below. 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 function calls) during execution. This error is not 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.

6.2 Layout of Core

The arrangement of memory relative to G (the global environment) is designed to group read-only things 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:OS →    : OWGS → :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. Here are put all the writeable variables which appear in common blocks, together with fixed local environments. 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 in the initial implementation. Later versions will cause it to overflow into OWGS: Overflow writeable global storage, which is handled in the same way.

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 initialised 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 and dope) generated by one common block, are collected together and allocated continguously 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.

6.3 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 preceding block's space in CS extends part 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 common 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 form the control of the storage allocator.

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

6.4 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 WCS at its discretion.

6.5 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 41 and 42. The array itself is allocated according to the default rules. Restriction: the value of the equivalence must be in the range [0',0' 437777B]. 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.

6.6 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.

7. 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

7.1 Precedence of Operators

Expressions are made up of operators and operands. The operators, in order of precedence, are

FOR WHILE	loops
IF ELSE	conditionals
WHERE	sequential evaluation
&	sequential evaluation
RETURN FRETURN	function return
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, shift, cycle, logical and
**	exponentiate
+ - N' (unary)	unary + -, logical not

GOTO	transfer
← (on left)	
. $ @	field operations
$ @ (unary)	indirection, reference
[] ()	subscripting, function call

The operands are

constants

names

parenthesized expressions

7.2 Syntax of Expressions

The above list of operators by precedence, 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:

```
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   =  v:primary / <see p. 29> ;
a:primary       =  identifier / "(" a:tailing ")" ;
v:primary       =  constant / "(" expression ")"  ;
```

Note: this grammar is ambiguous because (A) can be parsed as both a:primary and v:primary. The intention is that the a-parsing be used if possible.

7.3 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 the following table, in which certain 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 imply that different types may appear. If the digits are the same, or there is no digit, the types must be the same.

A partial ordering on the numberic types is defined: I<R<D, R<C. Where two Ms or Ns 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. 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 below must be used.

An object of type U may be used where A appears in the following table. 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 it 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[1];
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	
*NS1	<,<=,>,>=	NS2	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.

7.4 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 under "statements" below

A1 IF I ELSE A2 — evaluates I. If it is ≠ Ø, evaluates A1 and returns its value, otherwise evaluates A2 and returns its value, or 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

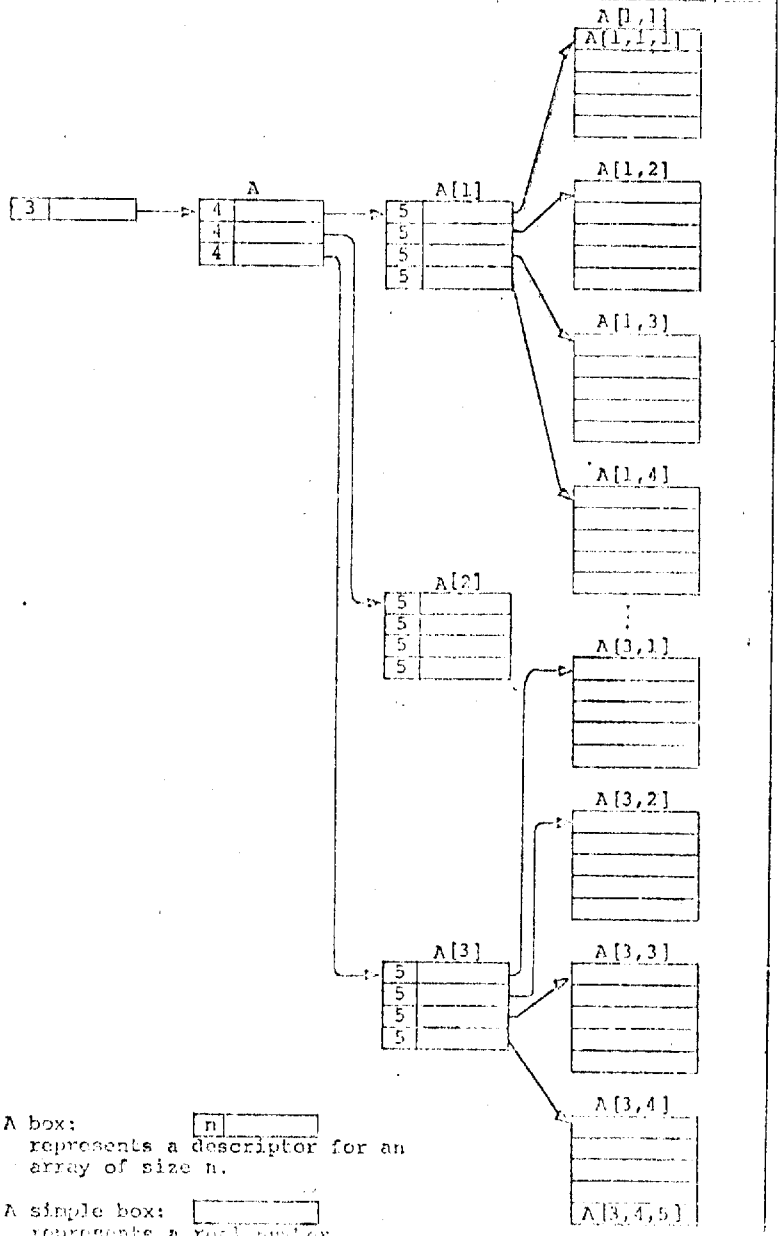
```
X ← (Y IF X < 3 ELSE Y+1)
```

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 "Function Calls" below
I1 OR I2	evaluates I1, returns 1 if it is ≠ Ø. Otherwise evaluates I2, returns 1 if it is ≠ Ø, otherwise Ø.
I1 AND I2	evaluates I1, returns Ø if it is =Ø. Otherwise evaluates I2, returns Ø if it is = Ø, otherwise 1.
NOT I	evaluates I and returns 1 if I = Ø, otherwise 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 that one 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 their operands
I1(LSH,RSH,LCY,RCY) I2	*these are 24-bit logical shifts (shift in Øs) or cycles

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 constant
(+,-)N N' I	obvious computes the bitwise (1's) complement of I
GOTO L	sends control to the statement labeled by A2. 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 A positioned in a word according to the field IF.



9. Function calls and returns

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 this case the type of the result is UNKNOWN and its mode is given by the type of the function.

9.1 Actual 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,0);
```

are function calls with 0, 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
"$" identifier or "$"  identifier "." full-word
  field
```

Then the value of each argument is stored in the corresponding formal argument of the function being called. No type conversion is done; nonmatched types are a (run time) error. Then control is passed to the function.

9.2 Returns

Return is done with an expression of the form

RETURN (expr, expr,...., expr) or RETURN expr or 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.

9.3 Failure Exits

If a failure exit is provided following the "//" in the call, a FRETURN will send control there. It may be a label, in which case control goes there, a RETURN, in which case a return is made from the function containing the failure exit, or VALUE expression, in which case the value of the expression becomes the value of the function. Just as for 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.

10. Statements

The following statements can appear in the body of a program block:

include:statement/allocation:statement
/declare:statement/action:statement

where

action:statement = "." assembler:statement/
for:statement/endfor:statement/
if:statement/elseif:statement/
endif:statement/
expression;

Statements of the first three kinds must appear in the prescribed order. Most of these have already been discussed. In this section we consider the restrictions on expressions used as statements and take up IFs and FORs.

10.1 Expressions 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 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:

1) the principal operator is

   a) <-, GOTO, RETURN, FRETURN, or a function call

   b) WHERE, &, FOR, WHILE, IF

2) If it 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

10.2 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 meaning should be obvious. 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 2 or 3 spaces.

10.3 FOR statements

The same thing can be done with FOR:

```
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 "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 ∅, 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 ∅, 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] ← ∅ FOR I←1 TO N FOR J←1 TO M;

10.4 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 MICPU/M-4 (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 10-23 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 MICPU/M-4 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]
IPD	$P[N]
BX	A[I]
BXD	($X')[I+N]
IM	N
IMX	X'+N
SR	R'[N]
ISR	$R'[N]
LR	L'[N]
ILR	$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.

11. 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 in-
                  cluding ";">
```

'Token' is defined in section 1.1.

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 in-
                  cluding "," 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. Beware. This glitch will be fixed at some far distant date.

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.

12. Intrinsic Functions

Figure 12.1 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 12.1, default values for arguments which the user is allowed to omit are given in parentheses after the argument type. For all functions which have returns, a routine which prints an error message and causes a subprocess trap will be supplied if the user fails to specify a failure action.

The remainder of this section describes the intrinsic functions in individual 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 12.1 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	X
APPEND	S,S	S	X	X
GC	S	I		X
STORINIT	I,I	I	X	
MARK	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 12.1 (continued)

12.1. 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 M1 CPU manual (M1CPU/M-4) 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 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.

CSE(S//F) expects to find a real number at the beginning of S_1, 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 of 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 0 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 bring the total number of characters written up to this many. 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 insuf-

ficient room to write the necessary number of characters onto S_2: in this case the writer pointer is unaffected.

CRS(R_1,S_2,I_3//F) appends the converted value of R_1 to S_2. Failure as for CNS. I_3 specifies the format in some as yet unspecified way: I_3 = 0, which is assumed if I_3 is omitted, results in some reasonable unformatted conversion.

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

12.2 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 M1CPU/M-4 for the exact specification of this operation, which is done with the ASP instruction.

INCDES(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 M1CPU/M-4 for details.

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

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

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

WCD(I_1,S_2//F) fails if S_2 is initialized, i.e. BP = RP. Otherwise it decrements RP and writes I_1 at the character position pointed to by the new value. 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, assumed 8 if missing. The value of SETUP is the string descriptor it creates. If I_3 is omitted, MAKE is called to assign space. BP = RP = WP is 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>INCDES(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≤I_2≤INCDES(BP,RP) leads to WP<RP<INCDES(BP,I_2); INCDES(BP,RP)≤I_2≤INCDES(BP,EP) leads to WP<INCDES(BP,I_2); and I_2>INCDES(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 only if there is not enough room in S_1; 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). Failure as for SCOPY.

12.3 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 user should keep his hands off it, under penalty of fouling up the operation of the allocator. 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 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 zone is created by the function

SETARRAY(A_1) which makes the space occupied by the array A_2 into a new zone by setting up some machinery inside it. CURRENT'ZONE is not set by this function. Blocks are released by

FREE(I_1,I_2) where the block pointer 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(I_1,I_2) which 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; the function

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; the description of the data structures, which will appear in the near future, should make this a simple task.

12.4 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) initialized 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.

13. Current Glitches

The following things do not work:

1) DECLARE name[subscript]<values to initialize further elements of an existing constant array

2) Multi-dimensional arrays

3) Deleting an entire block with the editor

The following things do not work properly, but are not complained about:

1) String arrays with both string length and array size given

2) Intrinsic functions called with BLL

3) Multi-line string constants (new feature, not documented)

4) FIXED and ORIGIN statements

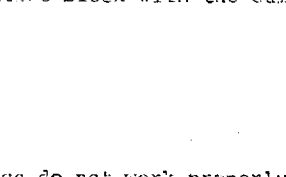
MEMORANDUM

TO: P DATE: November 7, 1969

FROM: Larry Barnes Larry

SUBJECT: CHANGE TO INCLUDE PROCESSING IN SPL

The current method of processing INCLUDE statements has some peculiarities. The following example indicates the problem. Suppose we have the following structure; where the labels are block names:



In SPL one would write:

```
COMMON A;
   ...
COMMON B; INCLUDE A;
   ...
COMMON C; INCLUDE B;
```

and so forth.

When processing the statement "INCLUDE E,C,D;", the blocks are included in the order D,B,A,C,E. This is peculiar in that I would like the blocks D,C,E included before A and B.

The following algorithm handles this problem. Build an include list for a block by:

i) inserting the blocks explicitly called for in reverse order;

2) scan this list and add any new blocks found in the scanned block's include list.

Thus our example would be processed:

(D,C,E)	(initial)
(D,C,E,B)	(B from D)
(D,C,E,B)	(B from C present)
(D,C,E,B,A)	(A from E)
(D,C,E,B,A)	(A from B present)

done

Then symbols would be defined starting with D and ending with A.

Peter claims this is a straight-forward change. I prefer these semantics to the current ones. If there are no objections, let's add this to the list of low-priority SPL improvements.

LB:cf

Bob Barnes

PCC	title: SPL EXECUTIVE COMMANDS	prefix-class-num-serial-rev: SPLEX/W-32
checked: Lampson, Schroeder, Barnes	authors: L. Peter Deutsch	approval date: 10/14/69
approved: Mit		classification: Working Paper
		distribution: Company Private; pages: 7

ABSTRACT and CONTENTS

This document describes the current (10-7-69) state of the SPL command language in those areas not covered by CSED/W-12 or SPLDS/W-17. In particular, it lists and describes all "executive" commands not discussed in the two earlier documents.

1. The Command Processor

SPL always returns to its command processor at the end of an operation. The command processor (abbreviated CP) identifies itself by printing a herald at the left margin: the herald character depends on the current mode of the CP as follows:

:	edit mode, expert
*	edit mode, beginner
-	debug mode, expert
=	debug mode, beginner

In the edit modes, a special group of miscellaneous commands is also recognized. The CP accepts a complete line at a time, which is collected using the standard line-edit. The "old line" for the edit is the previous command line, starting with the character after the herald; this is useful for correcting errors, or executing a command several times.

The syntax of commands depends on the mode: however, all modes except expert debug mode have essentially the same syntax for the command name, to wit: the command name is the first thing on the line. If the command takes arguments, they follow the name. In the beginner modes, the command name may be written in full or abbreviated as much as desired (even down to a single letter); if anything follows the command name (modes, arguments, etc.), one or

more blanks must intervene before it, and otherwise blanks following the command name are optional. In expert edit mode, the command name is always abbreviated to a single letter; blanks following it are always optional. Expert debug mode has its own peculiar syntax, which is described in SPLDS/W-17.

2. Special Commands

In the two edit modes, the CP also recognizes some special commands beginning with a dot(.). The shortest abbreviation for these commands is a dot and a letter, rather than just a letter as for edit commands; otherwise they behave just like beginner edit commands with respect to abbreviation and use of blanks. They perform functions of an executive or global nature which logically do not fall into either edit or debug categories. The formats and actions of these commands are itemized below.

2A. COMPILE Command

.COMPILE '

Compiles all blocks.

.COMPILE <list of block names separated by blanks>

Compiles the specified blocks.

.COMPILE

Compiles just those blocks needing compilation, i.e., those which have been edited since last being compiled, caused diagnostics when last compiled, etc.

2B. FINISHED Command

.FINISHED

Returns control to 940 DDT, under which SPL runs. SPL may be resumed by transferring control to .IN.

2C. Status Commands

.UNDEFINED

Lists all functions which have been called from some program which has been compiled, but which have not themselves been compiled.

.MAP

Produces a listing describing the layout of storage. The first two lines look like this:

USER RING: WGS 3 OF 22Ø-3777, RSGS 14 OF 34ØØØ-37777,
CS 4ØØØØ-4Ø135

This says that locations 22Ø through 3777 are currently assigned as WGS, of which only 3 cells are actually being used; 34ØØØ to 37777 are assigned to RSGS, of which 14 are being used; and 4ØØØØ through 4Ø135 are being used for CS. These two lines are followed by similar maps for the utility and monitor rings. All numbers are octal.

.STATISTICS

Print a variety of statistics about the current state of SPL. The printout looks something like this:

SPL 1Ø-3.7
34567 MAPS 23456 STORES 987 READS 876 WRITES
27 IB'S, 555ØØ-89Ø CELLS:
HEADERS: 5ØØ, TOKEN TABLES 2ØØØ-9Ø, SYMBOL TABLES 53ØØØ-8ØØ
2ØØØØ (SYMS) 3ØØØ (CONSTS) 3ØØØØ (PIT)

The first line identifies the assembly date (October 3)

and the patching level (7) of the version of SPL being used. The second line says that SPL has made 34567 references through its software map, of which 23456 were stores; read 987 pages from the drum; and written 876 pages. The third line says that there are 27 IBs (PROGRAMs or COMMON blocks) in the current ring, occupying 55000 useful cells and 800 cells of waste space. The last two lines break down these totals: 500 cells for headers (fixed overhead), 20000 useful and 80 waste cells for token tables, 53000 useful and 800 waste cells for symbol table and text, further broken down (useful cells only) into 200000 cells for symbol names and denotations, 3000 cells for constants, and 300000 cells for preprocessed text. All numbers are decimal.

2D. Saving and Retrieving Programs

.DUMP <file>

Dumps the entire contents of the current ring on the file. This includes source program, compiled code, global data, the status of the debugger, etc.

.RECOVER <file>

Restores the appropriate ring from the file, which must have been created with .DUMP (if this is not the case, there will be a diagnostic and no harm done). All previous information in that ring is lost.

2E. Changing Rings

.ADDRESS <digit>

Selects the user, utility, or monitor ring, according to whether the digit is 0, 1, or 2 respectively. The rings maintain an almost completely disjoint existence: source programs are permanently associated with the ring that was selected when they were read in. Similarly, no variable or address in one ring can be used to refer to information in another ring, e.g., if F1 is a function in one ring and F2 is a function in another, F1 cannot refer to F2 by name, and SPL will assume that F1 is referring to a function F2 in the same ring as F1 if the attempt is made.

.BOUNDARY <octal number without B>

Deletes all information in the ring in which the given address falls, then sets the boundary between RSGS and CS in that ring to the given address. There will be a diagnostic with no harm done if the address is illegal. Normal values for the boundary are 40000 in the user ring, 440000 in the utility, and 640000 in the monitor.

3. Useful Knowledge

SPL is started up by RECOVERing (in the 940 exec) from a rather large file, CONTAINing DDT, and starting up at .SPL. SPL will print its header (the first line printed by the STATISTICS command), some garbage about DUMMY COMPILING, and finally a ; herald indicating it is ready for use. Leaving SPL momentarily should be accomplished with the .FINISHED command. Since SPL currently has no logic for breaking into a long unwanted printout, the following ad hoc rules should be observed. There is no way to interrupt a compilation. To interrupt a long printout in edit mode, break out with control-A, put -1 into the call QUIT, then proceed with :P. If this gives an ILT trap, try :LIG. To interrupt a long printout in debug mode, try control-E followed by DSB XDECOM:0. Warning: if the P-counter is below 4000 when you do either of these things, the results will probably be disastrous unless the instruction being executed was a RSB 42B or 43B.

checked: Anne L. Barnes
checked:
approved: DLM
author: L. Peter Deutsch, C. K. Jones, R. K. Love

approval date/revision date: January
classification: Manual
distribution: Company Private
pages: 20

ABSTRACT and CONTENTS

This document serves as a user manual for the phase 1 language editor. The appendix documents the line-logic editing facility. Syntax and semantics appearing in this manual will be maintained for in house use and are not expected to change until the phase 2 version is released. Phase 2 will be a complete finished system for external use.

INTRODUCTION

The following pages are concerned with the MICS language editor for SPL and FORTRAN programs. This facility manifests itself to the user as a collection of commands and concepts in two flavors: basic and extended. The extended language editor is upward compatible to the basic version and provides the experienced user with quick and convenient ways to do complex editing.

Additionally, the distinction of phase 1 and phase 2 is necessary. The phase 1 language editor is an interim facility to be used in house only and will eventually be superseded by phase 2. The primary objective of the following pages is to document the phase 1 language editor as it is implemented. Phase 1 does not make a distinction between basic and extended versions. The phase 2 basic version will be a subset of what appears on the following pages, while the phase 2 extended version will be a superset.

GENERAL CONCEPTS

The basic difference between a language editor and a text editor is in the way the material to be edited is viewed. Usually, a text editor views its material as a collection of characters. On the other hand, a language editor has a higher level of understanding which allows it to view its material as a collection of tokens, where each token is a collection of characters. In other words, a text editor might view "132+TEMPERATURE" as 15 characters, whereas a language editor would view it as 3 tokens. A more sophisticated language editor would further recognize the 3 tokens as a number, an operator, and a symbol.

The MICS language editor views both SPL and FORTRAN programs as collections of tokens. Furthermore, it recognizes certain tokens and structural concepts. Structurally it is aware of lines and blocks, where the definition of block depends upon the programming language. The tokens recognized include block names, all FORTRAN labels, and SPL labels which appear as the first token on a line.

The purpose of this language editor is to provide a means for creating and altering programs. Consequently, there are ways to request editing actions to be performed as well as address physically where they are to occur in the program. The basic addressable quantity is a line. The language editor is always aware of a "current line" and allows addressing of the

subsequent line to be expressed as relative to the current line, relative to the current block, or absolute to the entire program. A complete discussion appears in the semantics section.

SYNTAX

The syntax appearing in this section is strictly for phase 1. It is anticipated that phase 2 will be an upward compatible extension. The character "¢" signifies the end of a line and represents a carriage-return-line-feed. Although the 14 commands appear in their verbose form, the language editor will recognize any contiguous subset of characters which starts with the initial character. Thus, SUBSTITUTE means the same as SUBST, SUB, and S. The first character following the command must be other than a letter or digit.

```
language:editor:command =
                "APPEND" [address] text
              | "CHANGE" [interval] text
              | "DELETE" [interval] ¢
              | "EDIT" [modes] [address] ¢ line ¢
              | "INSERT" [address] text
              | "LIST" [modes] [interval] ¢
              | "MODE" modespecs ¢
              | "NEXT" [modes] [integer] ¢
              | "PREVIOUS" [modes] [integer) ¢
              | "READ" file [address] ¢
              | "SUBSTITUTE" [modes] subspec [interval] ¢
              | "UNDO" ¢
              | "VALUE" [address] ¢
              | "WRITE" file [interval] ¢ ;
```

```
modes          = modespecs ":" ;
modespecs      = 1$(modespec) ;
modespec       = "A" | "B" | "C" | "I" | "N" | integer ;
text           = Ø $(line Ø) B^C
               | ["="] "(" interval ")" Ø ;
B^C            = <character code 142_8 = control B> ;
sign           = "+" | "-" ;
integer        = 1$(digit) ;
file           = <a 940 file name> ;
line           = $(character-Ø) ;
interval       = address
               | address "," address ;
address        = head $(tail)
               | [block] search $(tail) ;
head           = "."
               | [block] label
               | [block] "#" integer
               | [block] "$" ;
tail           = search
               | sign integer ;'
block          = "<" [name] ">" ;
search         = ["-"] token:search ;
token:search   = "/" [tokens-"/"] "/"
               | "*" [tokens-"*"] "*" ;
label          = name ;
name           = letter $(letter | digit | "'") ;
```



```
subspec        = "/" [tokens-"/"] "/" [tokens-"/"] "/"
               | "*" [tokens-"*"] "*" [tokens-"*"] "*"
tokens         = token $(token)
token          = <defined by the language's syntax>
```

SEMANTICS

APPEND: Appends the text after the address. If no address is specified, appends after the current line. The last line appended becomes the current line. If no lines are supplied, the addressed line becomes current.

CHANGE: Replaces the interval by the text. If no interval is specified, replaces the current line. The last line of the text becomes the current line. If no lines are supplied, the first line of the interval becomes current. The number of lines changed will be printed if the interval consisted of two addresses rather than one. CHANGEs may not extend across block boundaries. The only way to CHANGE the first line of a block is to CHANGE the entire block.

DELETE: Deletes the interval. If no interval is specified, deletes the current line. The line before the interval becomes current. The number of lines deleted will be printed if the interval consisted of two addresses rather than one. The only way to delete the first line of a block is to delete the entire block. Deletions may not extend across block boundaries.

EDIT: Uses the addressed line as the "old line" for the line editor. Editing conventions for the line

editor are in the appendix. If no address is specified, the current line is used. The EDITed line becomes current. Meaningful modes are "A", print the new line after EDITing: and "B", print the old line before EDITing. Modes appearing with the EDIT command are temporary and do not disturb the permanent modes set by the MODE command. For a complete discussion of modes, see the semantics of the MODE command.

INSERT: Inserts the text before the specified line or before the current line if no address. The last line inserted becomes current. If no lines are supplied the addressed line becomes current.

LIST: Prints the interval. If no interval, prints the current line. The last line actually printed becomes current. "I", interpret, is the only meaningful mode (see MODE Semantics). Modes appearing with the LIST command are temporary and do not disturb the permanent modes set by the MODE command.

MODE: The editor executes certain commands in different ways depending on a set of internal state variables called modes. A permanent set of modes are always in effect and can be set and reset by the MODE command. The permanent modes can be over-ruled for the duration of one command with

temporary modes as in EDIT, SUBSTITUTE, and LIST. Basically, there are five modes:

1) A (after): if on, causes lines being affected by EDIT and SUBSTITUTE to be printed after the operation is complete.

2) B (before): if on, causes lines being affected by EDIT and SUBSTITUTE to be printed before the operation commences.

3) C (confirm): if on, causes lines being affected by SUBSTITUTE to be printed and requests confirmation prior to actual substitution. Permission to SUBSTITUTE is granted by typing "Y" (yes) and denied by typing "N" (no).

4) I (interpret): if on, control-L will print as code 154_8 (in phase 2 it will cause a page eject); otherwise, "control-L" prints as "^L".

5) integer: the value of the integer determines the maximum number of SUBSTITUTEs which can occur. If more than this value are attempted, a message will be printed which indicates the SUBSTITUTE command was limited to this number.

One additional letter may appear with modes: "N". When "N" is encountered, all alphabetic modes following it are reset. Initially, A, B, C, and I are reset (off) and the substitution limit is set to 50. Thus, "MODE 100INABC" sets the substitution limit to 100, sets Interpret, and resets After, Before, and Confirm.

NEXT: Prints the next N lines to the current line just as LIST would, where N is the number following the NEXT command. If N is omitted, the next line is printed. The last line printed becomes the current line.

PREVIOUS: Prints the previous N lines to the current line just as LIST would, where N is the number following the PREVIOUS command. If N is omitted, the previous line is printed. The last line printed becomes the current line.

READ: Reads the file and INSERTs it before the addressed line. The last line read becomes the current line. If no address is specified, the file is APPENDed to the end of the entire program. The file name must be either surrounded by single quotes or terminated by a blank or end of line.

SUBSTITUTE: Searches the interval for occurrences of the second set of tokens and SUBSTITUTEs the first set of tokens for each occurrence. If no interval is specified, the current line is used. The last line having a substitution made in it becomes the current line. Note that the second set of tokens will match regardless of spacing. That is, /␣C-A/␣A␣+␣C␣/ will find a match in "X-A+C:". The first set of tokens is inserted exactly as

stated. The replacement in the example will result in "X←␣C-A;". Simply stated, the first set of tokens is inserted in the line as if it were a string of characters. Comments may not appear in the first set of tokens. If the first set of tokens is null, the first set from the previous SUBSTITUTE will be used.

UNDO: This command will undo the deletion caused by the last CHANGE, DELETE, and EDIT: provided no commands affecting text have been executed since. Thus, if a grievous mistake has been made, the drudgery of restoring the old lines is alleviated. Note, however, that the new line insertions made by CHANGE and EDIT are not undone and therefore must be normally attended to. The old lines will be physically located, as a group, following the new lines. It is recommended that one not grow too accustomed to this command, as its usefulness will be compromised. Executing two UNDOs in a row will not restore the state which existed two changes previous. The current line is left unchanged.

VALUE: This command will print the editor address of the specified line in two forms. For example, "#4 = %#57" would mean line #4 on the block

containing the line and line #57 of the whole program. If no line is specified, the current line is used. The addressed line becomes the current line.

WRITE: Writes the specified interval on the file. If no interval is specified, the entire program is written. The last line written will be the new current line.

text: Basically, there are two ways of specifying text. The first is just a series of lines entered from the teletype under the control of the line editor. The old line is always the previous entered line except for the first line, which has a null old line. See the Appendix for a discussion of the line editor. The second method for specifying text allows one to use lines which are already in the program. The lines to use are specified by the interval. Additionally, if the "=" is present, the specified lines are deleted.

interval: An interval specifies a group of one or more contiguous lines. The second address must have an absolute line number which is not less than the first. A block specified in the first address will be used for the second address if not overridden.

address: A line address is composed of a starting point

(head or search) possibly followed by a series of line increments (tail).

head: This specifies a specific line. The current line is referenced as ".". The appearance of a block permits a line other than one in the current block to be specified. "$" will address the last line of the appropriate block. "#" followed by an integer will select the line numbered as the integer. The first line of a block is #1. A line may also be addressed by its label.

tail: This takes the line specified by head and increments forward or backward accordingly. The signed integer increments that number of lines. The search is explained below.

block: This defines the scope in which lines are addressed. The presence of a name confines the line selection to the block of the same name. If "<>" appears, the selection is over the entire program.

search: Searches are normally forward unless the "-" is present, in which case, they are backward by line. Associated with a search is a starting point and a scope. The starting point in the address syntax is the line adjacent to the current line if either no block or the unnamed block is specified; and the line adjacent to #1

if a specific block is named. The search is circular within the scope, looking at the first line after the last line if the search is forward, and vice versa if backward. The scope is defined by the semantics of block, if block is present. Otherwise, the current block defines the scope.

token:search: This will find the first occurance of the specified set of tokens and address that line. Spaces are ignored and SPL comments are illegal.

label: The only SPL labels recognized by the language editor are those which appear as the first token on a line followed by ":" on the same line.

QUIT

Typing QUIT while the language editor is in control will function as a break facility in phase 1. This can be used to safely terminate the current action. Actions which are QUITable are:

1) LIST, PREVIOUS, NEXT, and WRITE: will terminate after the line being processed when QUIT occurs.

2) APPEND, CHANGE, INSERT: will terminate after the text line, being processed when QUIT occurs, is completed.

3) SUBSTITUTE: will terminate after the substitution, being processed when QUIT occurs, is completed.

4) search: will terminate after the line, being searched when QUIT occurs, is [illegible] not to contain the searched for item.

RESTRICTIONS

Unfortunately, there are currently some problems associated with block boundaries. These are itemized as follows:

1) You cannot DELETE lines from more than one block at a time.
2) You cannot DELETE the first line of a block unless you DELETE the entire block.
3) You cannot EDIT the first line of a block.
4) You cannot CHANGE lines from more than one block at a time.
5) You cannot CHANGE the first line of a block unless you CHANGE the entire block.
6) You cannot introduce text which has SPL COMMON, PROGRAM, or END in it unless it will go in between already existing blocks--at the very end, or at the very beginning. This applies to APPEND, CHANGE, INSERT, and READ.
7) You cannot SUBSTITUTE a string which has SPL COMMON, PROGRAM, or END in them for anything.

APPENDIX

LINE EDITOR CONCEPTS

The line editor is the input interface between the teletype and MICS. When one of the MICS subsystems needs a line of teletype input, the line editor receives and retains control until the user is done composing a new line, at which time control and the entire new line are returned to the controlling subsystem.

Instead of typing in all the characters, the user may compose the new line by editing the "old line." The content of the old line is determined by the controlling subsystem and is usually the previous new line received by that subsystem.

Both the new line and old line have character pointers associated with them; initially these are set to the first character position. As characters are typed in from the keyboard, both character pointers are advanced. Thus, if the old string initially has "ABCDE" in it, the new string nothing, "XYZ" is typed and the editing facility is used to copy the next character from the old to the new string; the resultant new string will contain "XYZD".

The user communicates with the editing facility by typing control characters. In some instances, the editing facility also listens to one character following a control character (indicated by C below). The list below gives the different

control characters and their resultant actions. A control character is typed by depressing the CTRL key while typing a normal character. Using control A as an example, control characters are signified as A^C. Note that normal character typing advances the pointers of both the old and new strings except during insert mode (between E^C brackets).

LINE EDITOR COMMANDS:

A^C Backspace one character in new string and print "←".

B^C Print CR-LF and finish.

C^C Copy one character from old string to new string and print copied character.

D^C Copy rest of old line into new line and finish, printing copied characters and CR-LF.

E^C Initiate and terminate insert mode, print "<" or ">". Characters typed after "<" and before ">" will not advance the old line character pointer.

F^C No type version of D^C.

G^C (Nothing)

H^C Copy rest of old line into new line, printing copied characters. Just like D^C except CR-LF is not printed and line is not finished.

I^C Insert spaces in new line up to next tab stop, printing them; advance old line that number of spaces. NOTE: if current character is in column #4 and tabs are 5 & 10, I^C will insert 5 spaces. The first tab is set at 9 and there-

after at five space increments (8, 13, 18, 23...63, 68).

J^C Puts CR-LF into new string and continues to accept input after printing CR-LF.

K^C (Nothing)

L^C (Nothing)

M^C Print CR-LF and finish.

N^C Backspace one character in both old & new string and print "↑".

$O^C C$ Copy characters up to C in old line into new line printing; if the very next character is C, the following one is used to terminate the copy.

$P^C C$ Skip over characters in old line up to C printing "X" for each character; if very next character is C, the following one is used to terminate the skip.

Q^C Restart line anew, printing "←". Reset old and new string character pointers.

R^C Retype unaligned by printing LF, rest of old line, CR, LF, all of new line; continue to accept input.

S^C Skip one character in old line, print "/".

T^C Retype aligned (like R^C) - Note, control characters printed by the "^C" convention will count as 1 character; thus, the number of characters unaligned indicates the number of control characters in the line.

U^C Copy, to next tab, characters from the old line into the new line -- just like I^C only print copied characters instead of spaces.

$V^C C$ Take C literally unless it is less than 100_8 in which

case add 100_8, print C, advance old line character pointer by 1.

W^C Backspace new line to first blank preceding a non-blank. Thus, "ABC DEF W^C" will end up as "ABC " as will "ABC DW^C".

$X^C C$ Skip through C - like P^C.

Y^C Concatenate new and old strings into old string and re-edit, print CR-LF.

Z^C Copy through C, like O^C.

DCC

title: SPL COMMAND LANGUAGE AND DEBUGGING SYSTEM

prefix/class-number/revision: SPLDS/W-17

author: M. Greenberg
Roger Sturgeon

approval date: 9/10/69

revision date:

classification: Working Paper

distribution: Company Private

pages: 11

ABSTRACT and CONTENTS

The debugging language for the M1 Compiler System is described in detail sufficient for the user. Informal syntax and semantics are given for all commands, including a careful treatment of possible program states with respect to suspended tasks.

The command processor will exist at any time in one of several modes. These are:

executive	&
verbose editor	*
quick editor	:
expert debugger	-
beginner debugger	=

Each of these modes will have its own herald as indicated. SPL is entered in executive mode and will thus type an &.

To change modes merely type the herald for the desired mode immediately after another herald and continue. This new mode will stick until another herald is typed.

The executive and editor modes will be discussed elsewhere. The debugger is now described.

A command in beginner mode is of the general form

=<command> [<modifiers>:] [arguments]

By <command> is meant a sufficient number of characters to distinguish the command from all others, terminated by a blank. The modifiers are a collection of single letters terminated by a colon. The nature of the arguments depend on the command. The entire command is terminated by a carriage return.

A command in expert mode is of the general form

-[<modifiers>] <command> [<args>]

The command will be a single punctuation character and will thus serve to terminate the modifiers and eliminates the need to type a blank after the command.

Command descriptors

MODE #

This command takes no arguments (other than modifiers) and sets the debugger permanently into all the specified modes.

GOTO $

Takes two optional arguments. First is the editor address of the statements to transfer control to. Second is number of break points to pass through before control is returned to the debugger.

CONTINUE ,

Takes one optional argument, the number of break points to pass through. This command will ignore a break on the first statement executed.

STEP +

Takes one optional argument, the number of statements to execute.

BREAK .

Takes one argument, which is an editor address or interval at which to set a breakpoint(s).

KILL [

Takes one argument, an editor address or interval of breakpoint(s) to clear.

DISPLAY]

No arguments, lists all breakpoints.

TRACE :

Takes two arguments, the first is an editor address. The trace breakpoint is set to this address and the program 'continues.' The second argument specifies the number of breakpoints to pass through.

REPORT (

The user may specify that the value of any number of expressions be printed out at a break. Executing the REPORT command (takes no arguments) puts the debugger in a state where it is editing a line interpreted as a list of expressions separated by commas, whose values should be printed in the break message.

NOTES:

Whenever a program breaks, a break message is printed. The message will always start with the break address. The statement where the break occurred becomes the "current line" for the editor. To cause the source statement where the break occurred to be printed, use the L modifier with the program execution commands (CONTINUE, STEP, TRACE, GOTO). To suppress this, use the Q ("quiet") modifier.

A conditional breaking facility can be evoked in the following way; If at the time control is transfered to the user program, a function with name BREAK' is defined, then the debugger will cause the program to be interpreted and the function BREAK' will be called after every statement is executed. Thus any user specified condition can be tested for after every statement.

To clear all break points use the modifier A with the KILL command.

To set or clear the trace break point use the modifier T with the BREAK or KILL command.

To cause a break message at every breakpoint passed through use an E modifier with the STEP, CONTINUE, TRACE, and GOTO commands. To cause a break message only at the end use an N modifier. If no modifier is used, then the current mode will take effect.

The user can specify that the GOTO, TRACE and CONTINUE commands only count the breakpoint on the current statement when counting the number of breakpoints passed through, by using the H modifier. The G modifier causes any breakpoint encountered to be counted.

Notes on transfers of control to user program: The debugger has the facility for maintaining two programs at once. They

will hereafter be referred to as Task 1 and Task 2. The debugging may exist in a state called the zapped state (reset state) in which the call stack and hardware stack are initialized and the debugger is not aware of any tasks. Using a Z modifier with the mode command will zap the state. When a transfer of control statement is executed a number of things must be considered before transfer of control is allowed:

1) Do any statements or functions need recompiling? If so, they are recompiled. If the break statement has moved the P-counter must be changed accordingly.
2) Is continuing legal? It isn't if the break statement has been modified or if the call stack cannot be unwound correctly (see below).
3) Under what task should the program be run?
4) Should the call stack be unwound and how much?

Control can be transfered to the program by the GOTO, CONTINUE, TRACE and STEP commands or by a direct statement. CONTINUE, TRACE and STEP are legal only if

a) a Task 1 program exists
b) the P-counter is not in the middle of the statement Task 1 broke in, if the statement has been modified since the break.

To transfer control, first all modified statements or functions are recompiled. Then the action taken is determined by the following diagram.

	call stack not empty	call stack empty
GOTO command or direct statement containing GOTO or RETURN	unwind stack transfer control as Task 1	illegal
otherwise	unwind stack transfer control as Task 2	transfer control as Task 1 with call stack set to dummy local environment

Unwind stack: The stack must be unwound far enough so that any call on the stack, in a statement that was subsequently modified, is removed from the stack. Once this is accomplished the stack must be further unwound until the top function on the stack is the same as the current editor function. Unwinding the call stack requires user confirmation. If the stack is unwound the P-counter is set to the instruction after the top call on the unwound stack.

Run as Task 2: This means the state of Task 1 and the hardware stack pointer are saved. When Task 2 terminates for any reason the saved state and pointer are restored.

EXAMINE /

This command takes Ø, 1, or 2 arguments.

If Ø, then the last quantity printed is printed again

If 1, then the value of the expession is printed

If 2, then the arguments are interpreted as bounds and everything in between is printed inclusive.

The value of an expression can be printed in one of the following formats

signed integer	I
unsigned integer	U
real number	R
6-bit characters	6
8-bit characters	8
pointers (with B, S, I, U)	P
double precision (with R or 0)	D
complex	X
unsigned octal	O
field descriptor	F
label or function	B
string	S
machine code	M
longlong	W

The type of the root operand in the parse tree will determine the format in which the value of an expression will be printed

unless a permanent mode is set. To suppress the permanent mode use a V modifier.

Another format may be specified instead by using one of the above modifiers. The examine command will abort if what is requested doesn't make sense.

For the two argument variety of the command both arguments must be references (in the sense of the SPL manual) i.e., be simple or have a principal operator which is indirection or subscripting, to be meaningful. Further, both arguments must specify addresses in the same environment, i.e., the same common block or function, if they are both simple variables or subscripted variables.

The two arguments otherwise are interpreted as absolute addresses and the contents of the cells between the two addresses are printed in the specified mode.

NEXT >

This command takes one optional argument, n. It prints the next n quantities following the last quantity (location) printed. If the argument is missing, it is taken to be 1. The format used for each quantity printed is taken from the symbol table unless overridden by modifiers in the command. Reaching the end of an environment (function, common block, absolute) terminates the command.

PREVIOUS <

This command works the same as NEXT except that the n preceding, rather than following, quantities are printed.

LEVEL ↑

This command takes two arguments. The first specifies the number of levels of local environment on the stack to jump back. A negative number means go forward on the stack. The second argument if specified is a function name. In this case jump n levels of that function.

FIND)

Not yet specified.

All state registers, etc., will be put in fixed places in the current global environment by the debugger and may be referred to with built-in symbols, to wit (in order):

PC'	program counter
AR' BR' CR' DR'	4-word accumulator
XR'	index register
LR'	local environment
GR'	global environment
SR'	status register

block name

TABLE OF MODES

Mode	Used With	Meaning
A	KILL	clear all breakpoints
B	EXAMINE, MODE	label or function
C		
D	EXAMINE, MODE	double precision (R,O)
E	GOTO, CONTINUE, STEP, TRACE, MODE	print message at every breakpoint passed
F	EXAMINE, MODE	field descriptor
G	GOTO, CONTINUE, TRACE, MODE	all breakpoints encountered are counted
H	GOTO, CONTINUE, TRACE, MODE	only breakpoint on current statement is counted
I	EXAMINE, MODE	signed integer
J		
K		
L	GOTO, CONTINUE, STEP, TRACE, MODE	print source at break
M	EXAMINE, MODE	machine code
N	GOTO, CONTINUE, STEP, TRACE, MODE	print message only at end
O	EXAMINE, MODE	unsigned octal
P	EXAMINE, MODE	pointers (B,S,I,U)
Q	GOTO, CONTINUE, STEP, TRACE, MODE	do not print source at break
R	EXAMINE, MODE	real number
S	EXAMINE, MODE	string
T	BREAK, KILL	set/clear trace breakpoint

U	EXAMINE, MODE	unsigned integer
V	EXAMINE, MODE	suppress permanent mode
W	EXAMINE, MODE	longlong
X	EXAMINE, MODE	complex
Y		
Z	MODE	map state
6	EXAMINE, MODE	6-bit characters
8	EXAMINE, MODE	8-bit characters

M E M O R A N D U M

TO: P\ DATE: January 16, 1970

FROM: Peter Deutsch L Peter Deutsch

SUBJECT: LATEST UPDATES TO SPL

Effective immediately (i.e. the first version of SPL dated 1-14-70 or later), numbers input to executive commands are normally taken as decimal. A number followed by B is taken as octal. This applies to .ADDRESS, .BOUNDARY, .OCTAL, and the new commands listed below. For example, to list the utility ring in octal, use .0 4ØØØØØB 5777773.

The following commands are added:

.GO — Starts execution on the bare machine. See below for details.

.LIST PHYSICAL — Lists the physical map of the simulated CPU. Entries whose existence is implied by the software tables (map, FMT, CHT) but have not actually been loaded into the map are enclosed in parentheses.

.MOVE n1 n2 — Moves (copies) the contents of page n1 into page n2. A positive page number refers to a page in the normal virtual memory; a negative number, to a page in the simulated real memory. Each of n1 and n2 may be either a page

number (between -31 and 127) or a page address (between -174ØØØB and 774ØØØB).

.BOUNDARY n1 n2 — Sets the boundary between RSGS and CS at n1, and also sets the G-Register value for the appropriate ring to n2.

.SET FORMAT n — Sets the spacing and indentation format according to n, as follows:

n	spacing	indentation
1	standardized	as input
2	as input	as input
3	standardized	standardized
4	as input	standardized

.SET MARGIN n — Sets the position of the first column for standard indentation to n. The left edge of the paper is position Ø.

.SET INDENT n — Sets the amount of indentation per level of logical nesting to n. With standardized indentation, a non-comment line is listed with $n_1 * n_2$ leading blanks, where n_1

is the MARGIN parameter, n_2 is the INDENT parameter, and k if the number of enclosing IF and FOR blocks.

.SET OFFSET n — Continuation lines are indented by $n_1+kn_2+n_3$, where n_3 is the OFFSET parameter.

The parameters listed above are printed out by .LIST FLAGS and are initialized as follows: FORMAT = , MARGIN = 1Ø, INDENT = 3, OFFSET = -1. .ZERO FORMAT will reset FORMAT to its initial value, etc.

The following glitch list includes all presently known problems which are either (a) commonly encountered, or (b) scheduled to be fixed. In view of the increased work load I am presently handling, no other glitches will be fixed beyond those marked with $ on the list, except in cases of extreme hardship.

1) Glitches 1, 4, 5, 6, 9 from the memo of 12-19-69.

$ 2) Glitches 2, 3, 7, 8, 13 from the memo of 12-19-69.

$ 3) Intrinsic functions called with BLL may not be defined by the user.

Glitches 10, 11, and 12 from the memo of 12-19-69 have already been fixed.

Bare machine mode has been mostly specified and coded. The comments about disk and drum simulation in the memo of 12-19-69