CORTEX USERS GROUP

S.P.L.

USER GUIDE -1

SPL 1.30 User's Manual

by Mike Riddle

May, 1981

Information concerning this product, bug reports, and general comments may be addressed to:

Evolution Computing
1718 E. Campbell #33
Phoenix AZ 85016
ATTN: Mike Riddle

Evolution Computing, or its agents, may, at their option choose to extend assistance beyond that specified herein. In no event shall such extension of service be construed as an extension of liability or an obligation of further extension of service.

1. INTRODUCTION to SPL

SPL is an acronym for Systems Programing Language. It has been designed to make the job of writing systems programs such as interpreters, diagnostics, compilers, and systems utilities, easier. It is in the process of development, and the present implementation is not the complete language. However, it does represent a powerful, and often simple solution for many types of programs.

The fundamental idea behind SPL is to have a language rich in powerful constructs. Most "high level" languages require that the user write much of the code to perform certain simple operations, rather than just describe what operations are desired. The basic constructs are not powerful enough. For example, in most languages, one needs to write many loops to scan and process data. SPL in its full implementation should all but eliminate the need for the user to write loops directly - the language will create them as necessary from the statement of the problem and optimize them at the same time.

Several facilities are needed by systems programs that are not often needed in simple applications programs. Chief amoung these is an easy interface to assembly code. In SPL, you may use inline assembly language statements using the same identifiers as used by SPL.

Another needed feature of SPL is the ease of using indirect addressing, or pointer variables. The addresses of data may be calculated in an expression and the result used to get the data actually desired. This allows high level implementation of complex data structures.

Dynamic memory allocation is provided to be used with pointer variables and structures to allow efficient use of memory.

Subroutines may be coded in SPL or assembly language, and may be nested and recursive, if desired. They may respond to a varying number of calling parameters, determined each time the subroutine is executed. Also, subroutines may have multiple entry points with a varying number of parameters used by each.

Programmer written loops are supported with the DO statement and the PROCEDURE statement. Procedures may be executed in-line, as well as called remotely with the DO statement. The procedure that is the target of the DO statement may be specified by a pointer variable, allowing the use of assigned procedures, and seperation of procedures for accessing data and performing operations on it.

Formatted Input / Output is simplified by having the format specifications in the I/O data list, and by having the default specification be "free-form". Also, You may GET FROM and PUT TO string variables, and the I/O lists may contain IF-THEN-ELSE clauses for selective and adaptive formating.

Direct I/O is supported in multiples of 128 bytes. You may seek and mix reading and writing to any file. Direct I/O may be used with pointer variables and structures to speed up I/O access, as data does not need to be transfered from the I/O buffer, but can be worked on directly.

Expressions allow complete mixed-mode arithmetic, and provide a large number of pre-defined functions.

The one "restriction" of the language is that all variables must be declared as to type. No defaults are provided. Since SPL provides such freedom of use of data, it becomes necessary to have the compiler check usage. Because of this "restriction", many SPL programs will work the first time they compile correctly.

This manual describes the release version that is designed to operate under Marinchip System's MDEX operating system. It should work, with less efficiency during file I/O, under their Network Operating System. A stand-alone kernel is being developed for use with ROM-based products, so that an application may be coded in SPL, and the resultant code executed from a small ROM system. Disk file I/O will not be supported in the ROM kernel. Instead, XOP calling sequences will be defined for you to code your own file I/O drivers.

2. Using SPL

An SPL program is first prepared in a text file using the text editor. This file should be given the program name.SPL. Next This file is compiled by the SPL compiler, providing the output in relocatable format in a file with a name ending in ".REL". Then all necessary REL files are combined by the link editor using the command SPLINK.

You should create three files for each program you write in SPL. If you call your program TEST, for example, the three files, and their contents, are:

```
TEST.SPL    the SPL source text
TEST.REL    the compiler generated relocatable code
TEST        the executable program
```

After creating the necessary files, enter your source text using the EDIT command:

```
EDIT TEST.SPL=   the first time
 or
EDIT TEST.SPL    to make changes later
```

When you have finished editing the program text, compile it using the SPL command:

```
SPL TEST
```

And finally, use the SPLINK command to make the executable program:

```
SPLINK TEST
```

To run your program, simply enter it's name as a command:

```
TEST
```

The actual process followed by the computer is more complex, but you do not need to understand it to use SPL. The following description is provided for your information only.

1. SPL reads TEST.SPL and writes assembly code into TEMP1$
2. SPLOPT modifys TEMP1$ to optimize generated assembly code.
3. The ASM program is called to read TEMP1$ and make TEST.REL
4. SPLINK prepares a more complex LINK command and calls the
5. LINK command, which produces the TEST executable file.

The SPL command actually offers several options for compilation and listing of your program. The full format specification is:

SPL <reloc file>=<spl file>[,[<asm code file>][,<listing file>]

Where <reloc file> will have a type .REL if no type is specified, <spl file> will have a type .SPL if no type is specified, <asm code file> will contain the generated ASM code, and TEMP1$ will be used if no <asm code> file is specified, and <listing file> will contain a numbered source listing with error messages and expanded COPY file text. It may be CONS.DEV or PRINT.DEV, if desired.

If a single program name is given without types in place of <reloc file=<spl file>, or only a drive indicator is specified for <reloc file>, then the same file name with appropriate types will be used. Some examples:

SPL test	uses test.REL and test.SPL
SPL 1/=2/test	uses 1/test.REL and 2/test.SPL
SPL test,,cons.dev	uses test.rel and test.spl, and makes a listing on the console.

If you wish to use file names that do not have .TYPE extensions in the file name, simply end the file name with a period. As an example, to compile a program with source on drive 2 and executable file also on drive 2, but without saving the relocatable output, you could use the following sequence of commands:

SPL temp2$.=2/prog	uses temp2$ as .REL and 2/PROG.SPL
SPLINK 2/prog=temp2$.	

If you just wish to compile a program to check for errors, you may use the command form:

SPL =<program name>

This form will not assemble the compiled output.

To make programs that use subroutines, you list each subroutine name after the main program name on the SPLINK command. If more than two subroutines are used, then you will need to make your own LINK command. An example using two subroutines is:

```
SPLINK main,sub1,sub2
```

If you use more than two subroutine names, the SPLINK command will not work for you. In this case, copy the file SPLINK.LNK to a file set up for linking your program, perhaps ended with ".LNK". As an example of this, for the test program, you would:

```
CREATE TEST.LNK,2
EDIT TEST.LNK=SPLINK.LNK
t
i
in test.rel,sub1.rel,sub2.rel,sub3.rel,sub4.rel
<press return to finish inserting>
end
```

From now on, to link versions of the test program, use the command:

```
MAKE test
```

The make command generates the command:

```
LINK TEST=@TEST.LNK
```

which will perform the required linking operations.

3. META Notation

SPL is described in this manual in both english text and in Meta notation. Meta notation is a means of specifying syntax very precisely. To understand the statement descriptions, you should be aware of a few fundamental notations.

Mandatory choices are specified within braces {} with each choice seperated by a vertical bar |. You must use one and only one of the choices. As an example:

```
{ INTEGER | POINTER }
```

means that you must use one of the words, either INTEGER, or POINTER.

Things enclosed in brackets [] are optional and may be used or omitted. An example is:

```
[ <statement label> ]
```

The notation <term> is used to describe something defined elsewhere. All syntax rules specified in that definition are to be followed. In the above example, the statement label is optional, but if it is present, the rules for making a statement label must be followed.

Any term followed by ... may be repeated as often as desired. An example of this is:

```
<identifier> [ <integer literal> ] ...
```

which indicates that an identifier must be present, and may be followed by as many integer literals as desired.

Anything else that is used in a description may be considered a keyword or punctuation, which must appear exactly as described, except that upper and lower case do not matter except in string literals, and any number of spaces may be present wherever any space is present.

4. LANGUAGE ELEMENTS

Any programing language is made up of basic elements. These elements may be keywords, like READ, variable names and literals, and the punctuation required by the language. To learn any language, it is often best to first learn the rules for making and using the basic elements, and then learn how to combine them.

Each type of statement has its own required syntax. Syntax refers to the "grammer" of the statement. It simply means that certain special characters are used as punctuation to tell the compiler what you mean, and they must be used exactly as specified in this manual.

The basic elements in SPL are:

1. Keywords
2. Identifiers
3. Variables
4. Literals

They are used to make expressions and statements.

4.1. Keywords

A keyword is either a word or punctuation that must be used exactly as it appears in this manual. They are used to tell SPL what type of statement or option you wish to use, and to logically seperate parts of the program.

4.2. Identifiers

Identifiers are names that you make up for your program. Any identifier must fit the following pattern:

    An alphabetic character
    none or more alphanumeric characters

The identifier must not be the same as an SPL keyword. The character "underline" may be used to seperate part of a name for readability. For example, MYNAME and MY_NAME may be used interchangably. There is no preset limit on how many characters may be used in an identifier, but you must type each of them every time you use the name, so do not make names longer than necessary for clarity. Some valid identifiers are:

    I   JJ   PAYROLL   MASTER_DATA   NAME

4.3. Variables

Variables are holding places for your data, and you give a different identifier to each of them. The rules for making literals are different for each type of variable.

SPL currently supports the following types of variables:

16-bit INTEGER variables
8-bit BYTE variables
32-bit DOUBLE precision integer variables
64-bit REAL (floating point) variables
dynamically allocated STRING variables
address POINTER variables

4.4. Literals

Literals are the constants you use within the text of your source program to specify actual values to be used by the program. Examples of literals are:

```
3     -17     3.3E-2     "John's house"     adr(BETA)
```

As an example, an integer variable named I can hold several different values. If you wish to put the value 3 into the variable I, you could use the statement:

```
I=3;
```

In this example, the variable reference is I, and the literal is 3.

4.4.1. Integer Variables

Integer variables represent integer (no decimal fractions) data in the computer. In SPL, one may choose amoung three kinds of integer variables, BYTE, INTEGER, and DOUBLE.

BYTE variables represent values between -128 and +127, and require one byte of storage each. They may be used to contain character data when each character is to be manipulated seperately.

INTEGER variables represent values between -32,768 and +32,767. They use two bytes of storage each. Most integer arithmetic is performed using INTEGER type variables. INTEGER type variables always start on an even memory address boundary.

DOUBLE precision integer variables represent values between -2,147,483,648 and +2,147,483,647. Each DOUBLE type integer requires four bytes of storage. DOUBLE type variables may be used for accounting purposes, with large disk data files, and in other applications requiring a large numeric range and faster speed or better accuracy than REAL variables. DOUBLE type variables always start on an even memory boundary.

The rules for forming integer literals are identical for BYTE, INTEGER, and DOUBLE type variables, except for the range of values that may be used with each type. All integer literals must fit the following pattern:

Optional + or - sign
One or more digits
Ended by a character other than .

If the first digit is a zero, then the literal is considered a hexidecimal number, and valid digits are:

0 1 2 3 4 5 6 7 8 9 A B C D E F a b c d e f

If the first digit is not a zero, then the literal is considered a decimal number, and only 0 through 9 are considered valid digits.

Negative numbers are in twos complement form. This results in the following equivalences for 16-bit integers:

```
00000 to 07FFF   are positive values  0 to +32767
0FFFF to 08000   are negative values -1 to -32768
```

Some examples of valid integer literals are:

```
0   0FC3    1    -32767    +25    32767    45
```

4.4.2. Real Variables

Real variables are capable of representing numbers that are not exact integers, and numbers that vary from very large to exceedingly small values. However, they are not always represented exactly in the computer. For example, it is not possible to represent the value .1 exactly in binary. Real variables give you a large range of values, but loose the exactness of integers.

Real numbers may assume values between:

```
3.37350334183 * 10^-80
      and
7.23700557733 * 10^+75
```

Real literals must fit the following pattern:

An optional leading + or - sign
One or more digits and a required decimal point

optionally, an exponent specification which fits the pattern:

The character "E"
an optional + or - sign
one or more decimal digits
ended by any non-digit character

The literal may not have any embedded blanks.

All real variables in SPL are stored in 8 bytes of memory, always starting on an even address boundary. In the current implementation of SPL, they are stored in "System/370 Floating Point Format", since that computer first popularized this form of real number representation.

Picture each byte of a real variable as follows:

```
EE MM MM MM MM MM MM MM
```

The first byte stores several bits of information. Picture Each bit of this byte as follows:

```
MS E6 E5 E4 E3 E2 E1 E0
```

MS is the sign of the mantissa, and thus the sign of the entire numbers value. If MS=1, then the value of the variable is less than zero. If MS=0, then the value of the real number is greater than zero. Note that twos complement notation is NOT used. The MS bit has no effect on the mantissa digits MM .... MM.

The bits E6 through E0 are the exponent. It is expressed in what is called "excess 64 notation". That is, exponent values of 41 hex and larger indicate numbers greater than or equal to one. Exponent values less than 41 hex represent real numbers whose absolute values are less than one.

The exponent is the power of 16 that the mantissa should be multiplied by to get the real value. The "formula" for understanding the floating point representation is:

16^(EE-41 hex) * M.M MM MM MM MM MM MM

Some sample values are:

0	00 00 00 00 00 00 00 00
1	41 10 00 00 00 00 00 00
-1	C1 10 00 00 00 00 00 00
10	41 A0 00 00 00 00 00 00
-10	C1 A0 00 00 00 00 00 00
15	41 F0 00 00 00 00 00 00
16	42 10 00 00 00 00 00 00
255	42 FF 00 00 00 00 00 00
256	43 10 00 00 00 00 00 00
4095	43 FF F0 00 00 00 00 00
4096	44 10 00 00 00 00 00 00
.5	40 80 00 00 00 00 00 00
.1	40 19 99 99 99 99 99 99

Note that .1 is an irrational number in binary. This is one reason why you get answers like 1.99999999999 instead of 2 when you do some math operations on a computer.

4.4.3. String Variables

String variables are used to store text. A string is a variable length sequence of characters that may NOT contain an ASCII NUL (00) character. The length of the string may be zero, in which case it is called a NULL string. All string variables are initialized to null variables until they are used.

String literals must follow this pattern:

```
The character "
none or more ASCII characters
(the quote character " is represented by "")
ended by the character "
followed by any character except "
```

Some examples are:

```
"ALPHA # 3"   "123 S. 4th St."   "He said, ""Hello!"""
```

There are two types of strings: constants and dynamically allocated variables. A constant is stored in the program code itself and is never changed.

String space is dynamically allocated as needed from the unused memory space that exists between the end of the program and before the start of the operating system. Each string variable is assigned a two byte address pointer located on an even address boundary. This pointer contains a zero to indicate a NULL string, or the address of the appropriate text buffer.

The text buffer starts on an even address boundary. The first word is the number of active references to the text buffer. If this is zero, then the buffer must never be released into free space for reuse. This is used to indicate string constant storage within the program. Otherwise, it indicates a dynamically allocated string buffer, and the value of this word is the number of string variables that are currently assigned to this text. (If two or more strings are assigned to the same text, only one buffer is used, but both are counted as references. If both are reassigned to different values, then the text buffer area will be released to free memory.)

The second word of the text buffer is the count of the number of characters, not counting the NUL (00) end-of-string delimiter.

The actual text follows the count word and is ended with an ASCII NUL (00) byte.

5. POINTER Variables

POINTER variables are a form of INTEGER type variable that are used when working with memory addresses as data. They may be used exactly as you would use a normal INTEGER type variable. In addition, they may be used as an indirect address pointers to the data you actually wish to be used. Normally, A variable stores data to be used in it. A pointer variable's data is the address of the data you wish to use. This is a powerful tool that is usually only available in assembly language.

In SPL, there are two forms of pointers. First, you may use an integer as a pointer variable by preceeding it with the character "@". Second, you may use a STRUCTURE.

5.1. Integers used as Pointers

Any integer variable or expression that is preceeded by an @ character is used as an indirect reference. Thus, @1000 means the data stored at address 1000 in memory. @IX means to get the data in IX and use it as the address of the actual data.

This may be combined into expressions in several ways. @IX+2 means to take the data in IX and use it as the address of the actual data, to which 2 is added. If IX contains 1000 and memory location 1000 contains 25, then @IX+2 means 27.

@(IX+2) means to take the data in IX and add 2 to it, and use that as the address of the data. If IX contains 1000, and 1002 in memory contains 95, then @(IX+2) means 95.

5.2. Address Literals

If you are going to use pointers, then you also need a way to describe the address of a particular variable. This is similar to numeric literals. To do ordinary arithmetic, you need literals such as 19 and -25 as well as variables such as IX and J.

An address literal is written ADR(VARIABLE) where VARIABLE is the name of the variable whose address you desire. Note that you may NOT use an expression inside the parenthesis. If you desire the address of the location 2 bytes in memory after the variable IX, you would write ADR(IX)+2.

Typically, you might use a pointer variable as a switch between two different variables:

```
P=ADR(I); DO something; P=ADR(J); DO something;
```

If the procedure "something" uses P as a pointer variable, then the first time it is called, it will use I, and the second time it will use J.

5.3. Example of Pointer Nomenclature

As an example, assume that a program has three integer variables, I, J, and K, stored in memory as shown:

memory address	label	address contains
01000	I	47
01002	J	01000
01004	K	-19

In this case ADR(J) is exactly equal to the literal value 01002, J is equal to 01000, and @J is equal to the contents of memory location 01000, which is 47.

If the statement J=adr(K); is performed, then the following results:

ADR(J) is still equal to 01002, J is now equal to 01004, and @J is equal to -19 (the data stored in memory at the address which J contains as data).

5.4. Type of Data using Pointers

Normally SPL knows the type of data from the name of the variable since all variables have been declared in a TYPE statement, such as REAL, INTEGER, or STRING.

When using a pointer, you can specify any location in memory. The SPL compiler may not know how to handle the data it finds there. SPL will consider it an integer unless you specify a different type of data by using the characters ".", "$", "#" or "&".

```
@IX means the data is an integer
@.IX means the data is a real number
@$IX means the data is a string
@#IX means the data is a double precision integer
@&IX means the data is a byte integer
```

5.5. Use of Unsigned Multiply and Divide

Since pointer variables contain addresses, not signed integer values, it is necessary to perform any multiplication and division upon them with unsigned arithmetic. SPL provides the ** and // operators for this purpose. An example of their use might be to access an array element. In normal subscripting, you might specify TABLE(INDEX) for an element that is in the INDEX numbered position in TABLE. Using pointer variables, you could access it by:

```
@(ADR(TABLE)+(INDEX-1)**8)
```

This calculates the address of element number INDEX in TABLE when each element takes 8 bytes, (as REAL variables do).

6. EXPRESSIONS

Variables and literals may be combined with various operators into expressions that can be evaluated by the computer to produce a resulting value. An example of an expression is:

```
RATE*TIME-DEDUCTIONS
```

An expression is composed of TERMS combined with OPERATORS. A TERM represents a single value to be used in the expression, and an OPERATOR specifies how terms are to be combined.

6.1. Terms

In SPL, terms may be any of the following:

An integer or real numeric literal
A string literal
A predefined function
An indirect reference
a variable name
Another expression enclosed in parenthesis

6.2. Operators

Operators are assigned priorities similiar to those used in algebra. This means that multiplications are performed before addition, etc. The valid operators, listed in order of priority are:

1	Term evaluation	(including parenthesis)
2	^	(raise to a power)
3	unary + or -	(leading signs)
4	* /	(multiply and divide)
4	** //	(unsigned multiply and divide)
5	MOD	(modulus - the remainder after division)
6	+ -	(numeric addition and subtraction)
6	+	(string concatenation)
7	<= <> < >= > =	(logical relations)
8	NOT	(bit by bit inversion)
9	AND	(bit by bit AND)
10	OR XOR	(bit by bit OR and Exclusive OR)

6.3. Mixed mode expressions

Mixed mode expressions are allowed. This means that integer and real literals and variables may be used within the same expression. If either term combined by an operator is real, then the integer term will be converted into a real value before the operation takes place.

6.4. Logical Expressions

Logical expressions, including the relational operators, return an integer value. 0000 is False, and any other value is considered True. The relational operators return OFFFF for True.

The bit manipulation operations AND OR XOR and NOT perform their operation on each bit of an integer. Thus, 0005 AND 0006 is 0004:

```
        0000 0000 0000 0101  (0005 hex)
        0000 0000 0000 0110  (0006 hex)
        --------------------
   AND  0000 0000 0000 0100  (bit by bit AND function)
```

The modulus function returns the remainder after division. Thus 9 MOD 4 is 1, since 9/4 is 2, with a remainder of 1.

6.5. Data and Address expressions

Expressions that produce an address, such as pointer variable expressions, are refered to as address expressions. In the rest of this manual, an address expression means either an identifier or a pointer variable expression.

Data expressions are the normal type of expressions. They produce actual data as a result, rather than the address of the data.

6.6. Integer, Real, and String Expressions

In the rest of this manual, the term integer expression means an expression that has, or can have, an integer value as a result. For example, "ALPHA"=ANS produces an integer result, a logical value 0000 or OFFFF.

In a like manner, Real and String expressions refer to expressions that have the desired type of end result when evaluated.

6.7. Predefined Functions

SPL offers several predefined functions that may be used in expressions as terms. They are:

MATH FUNCTIONS

All of the following math functions that return a real result are accurate to only 7 significant digits in the current release of SPL, since they are implemented in 32-bit REAL arithmetic. This will be upgraded in a future release of SPL.

```
SQRT(X) real square root
LN(X) real natural logarithm of X
EXP(X) real exponential (e raised to the X power) of X
LOG(X) real common (base 10) logarithm of X
ALOG(X) real common exponential (10 raised to the X power) of X
ABS(X) real absolute value of X
RND real random number between 0.0 and 1.0
IRND(<integer limit>) integer random value with a range from
                      zero through limit-1

SIN(X) real sine of X in radians
COS(X) real cosine of X in radians
TAN(X) real tangent of X in radians
SEC(X) real secant of X in radians
CSC(X) real cosecant of X in radians
COT(X) real cotangent of X in radians
ASIN(X) real arcsine in radians of X
ACOS(X) real arccosine in radians of X
ATAN(X) real arctangent in radians of X
```

INTEGER SHIFT FUNCTIONS

All of the following functions take a 16-bit integer argument, perform the desired binary shift upon that value, and return an integer result.

```
SHIFTL(value,places) shift value left places, 0 fill on right
SHIFTR(value,places) shift value right places, 0 fill on left
SHIFTA(value,places) shift value right places, sign extend
                     (copy MSB) on left.
SHIFTC(value,places) shift value right places, circular fill
                     (bits shift out right go in left side).
```

I/O FUNCTIONS

CONIN returns an integer value of the next console key pressed. The character is NOT echoed to the console display.
CONRDY returns OFFFF if the console has an unread character available, and 0000 if no character is available.
IOSTAT returns an integer containing the system error status code of the previous file I/O operation.
FSIZE(<file name string expr>) size of file in sectors returned

ADDRESS (POINTER) FUNCTIONS

NRPARM integer number of parameters transfered to a subroutine
PARMADR(integer expression) address of the desired parameter
ADR(VARIABLE) integer address of the specified variable
IXADR(table,index,elementsize)
this function returns the address of the indexed element of table, where each element uses elementsize bytes.

STRING FUNCTIONS

LEN(string expression) integer number of characters in string
CMDLINE string containing command line parameters
CHR(string expression,integer expression)
The character at the indicated position of the string is returned as an integer. Zero is returned if the position is past the end of the string.
STG(integer expression)
The integer is returned as a single character string having the ASCII value of the integer expression.
SUBST(string expression,from integer expression,length expression)
A string containing the specified number of characters from the specified string expression, starting at the given position, is returned.
UCASE(string expression)
A string containing the same characters, except that all alphabetic characters have been changed to upper case, is returned.
LCASE(string expression)
A string containing the same characters, except that all alphabetic characters have been changed to lower case, is returned.

7. ARRAYS and SUBSCRIPTING

7.1. Introduction

An array is a collection of related data items of the same type. Each array has a single name, and may have more than one variable in it. The size of an array is the total number of individual variables stored in it.

Picture an array of three integers, called A. It might look like this in memory:

```
234      -98      45
```

You may refer to a specific element (variable) in an array by using a subscript. This is an expression that specifies which element of the array you wish to use.

In the example above, A(1) is a variable that currently has the value 234 stored in it. A(2) holds -98, and A(3) contains 45.

Subscripts are written by entering the array name and then the character (followed by an expression indicating which element you wish to use, and then ending the subscript with the character).

For example, you may write A(NR) which means that the current value of the variable NR should be used to pick out which element of array A that you wish to use. If NR contains the value 2, then A(NR) refers to A(2).

7.2. Declaring an Array

To create and use an array in SPL, it must be declared in a type statement such as REAL, INTEGER, STRING, etc. You may list the array name and the largest value of each subscript. For example,

```
REAL A(5), B(4), C;
```

specifies that A is an array having 6 elements numbered 0 through 5, B is an array having 5 elements numbered 0 through 4, and C is a real variable.

7.3. Extended Use of Subscripting

By specifying the size of the maximum subscript, you will make room for the array. You are not required to do this, however. By not specifying a size for an array, you can use subscripting to access other variables, located after the "array" variable.

There is no checking performed when using subscripts. This can lead to trouble in poorly designed programs, but can also be used to advantage. Since all data is allocated space in order specified, you may set up a list of variables:

```
INTEGER A,B,C,D,E,F;
```

An then use them either in the normal way, or by making subscripts:

```
A(0) refers to A
A(1) refers to B
A(2) refers to C, etc.
```

The number of bytes used for each element of the array is determined by the type of the referenced data. Since A is INTEGER, 2 bytes are used for each element. If A were specified as a BYTE variable:

```
BYTE A(1); INTEGER B,C,D,E,F;
```

then the relationships would be:

```
A(0) and A(1) refer to the two bytes of A
A(2) refers to the first byte of B
A(4) refers to the first byte of C, etc.
```

It should be understood that the above "trick" should never be used just for its own sake. If there is no positive advantage to be gained from refering to variables in both an array subscript notation and by individual name, use only one form or the other. Good programing practice always dictates that the simplest and clearest techniques are better than the tricky or complex. SPL allows more freedom than should be necessary - don't abuse it.

8. SPL PROGRAM STRUCTURE

SPL program modules are classed as either a main program or a subroutine. Each module is seperately compiled then combined with the system LINK program.

A main program is one that receives control from the operating system when the program is to be run. Each executable program must have one and only one main program.

A subroutine is a module that receives control when CALLed from either a main program or another subroutine. When its END. statement is executed, control will return to the calling program.

Subroutines and programing related to them will be discussed in a seperate section of this manual. A main program follows this format:

```
Option statements, if any
The program statement
.
.
(as many SPL statements as needed)
.
.
END.   (marks the end of the main program.)
```

The program statement consists of:

```
PROGRAM  <program_identifier> ;
```

The program identifier is the label that the link editor will use for this module in its memory map. The word PROGRAM and the semicolon are keywords and must appear exactly as specified. The program identifier is also the statement label of the first statement you want executed. As an example:

```
PROGRAM TEST;
.......
TEST: % first statement % ......
.........
END.
```

The END. statement causes control to return to the operating system, and must be the last statement in the module. There must be one and only one "END." statement per module. Usually, any variable type declarations and procedures are included between the PROGRAM statement and the starting label.

8.1. Comments

Comments may appear any place a space may appear, except inside a string literal. Comments start with the character % and end with the next % character. It is possible to have several lines be one comment.

An example of a statement with a comment is:

```
PROGRAM UPDATE % update the transactions data file % ;
```

8.2. Statement Labels

All SPL statements except options, PROGRAM, SUBROUTINE, and CODE may have a statement label preceed them. A statement label is an identifier followed by a colon. Examples are:

```
TEST:      PROGRAM_ENTRY:      ERROR_ROUTINE:  alldone:
```

Also, labels may be declared externally (see SUBROUTINES) by placing an asterisk after the colon:

```
NEXTCOMMAND:*          ERRORPROCESS:*
```

These labels are used by statements that can transfer control, such as GO TO, IF, and DO, to indicate where the desired part of the program is.

8.3. Statement format

SPL allows the use of either upper or lower case letters, having identical meanings, except during string literals, when upper and lower case characters are considered different.

Most SPL statements end with a semicolon, and may occupy as many lines as necessary. No particular spacing or column alignments are necessary. The only line-oriented statement in the language are ASM, which lets the rest of the line be an embedded assembly language statement, and COPY, which lets you include multiple text files in your source program at compile time.

8.4. Options

Currently, the only option statements allowed are:

```
.TABS;   or   .NOTABS;
```

Note the period that starts all option statements. These two statements let you select whether the assembly language code generated by SPL will have tab expansion, which requires more disk space and time, or just a space between each field, which is the default option.

9. SPECIFICATION STATEMENTS

In SPL, all variables must be declared in a specification statement before being used. Normally, all specification statements in a program are placed immediately after the PROGRAM or SUBROUTINE statement, as the storage for the variables is located in the program exactly where the specification statement occurs. Since specification statements are NOT executable, you should not give them labels or GO TO them.

There is a specification statement for each type of data:

```
BYTE
INTEGER
DOUBLE
REAL
STRING
POINTER
```

There are also specification statements related to different data organizations, discussed later in this manual:

```
STRUCTURE
STACK
QUEUE
```

There is no required order to these statements and each may be used as many times as needed. The one rule concerning them is to never place them where they will be executed. The best place is at the start of the program, as discussed above.

Parameters of subroutines must be declared like any other variable, but must not be assigned data values by specification statements within the subroutine.

9.1. Specifying GLOBAL and EXTERNAL references

SPL allows variables to be used by more than one module of a program by using the LINK editor to communicate the address of variables to each module when the program is prepared. A GLOBAL variable is one that is defined and stored within the module itself, but is used by other program modules such as subroutines. An EXTERNAL variable is one that is defined in another module, and you wish to use it in the current module. Variables that are only used within the current module, (which includes most variables), are called LOCAL variables when discussing GLOBAL and EXTERNAL references.

You can specify that an identifier is GLOBAL by following the identifier with an asterisk, "*", when defining it in a specification statement, or before the colon if the identifier is a statement label. Local identifiers that are followed by an asterisk may be used by other modules of the program, since the * tells the LINK editor to allow this. A variable to be used by subroutines might be specified in the main program as:

```
POINTER MONITOR* 0F800;
```

You specify that a variable is EXTERNALLY defined by preceeding the identifier with an asterisk, "*", when it is specified. In a subroutine, MONITOR would be an external variable, and would be referenced by preceeding the identifier with an "*":

```
POINTER *MONITOR;

GO @MONITOR;
```

One very important point to remember is that data assignments in the specification statement may not be made for EXTERNAL variables. This is because the compiler generates a data statement for it, but when the variable is referenced externally, it has already been defined in the other module. In practical terms, if the asterisk preceeds the identifier, you may not follow it with literals in the specification statement.

9.2. The INTEGER and POINTER specification statements

Integer variables are declared using the INTEGER or POINTER statement. The only difference between INTEGER and POINTER in the current release of SPL is documentation. Declaring an integer a POINTER alerts the reader that that variable is going to be used for indirect addressing. In a future release of SPL, POINTERs may take 4 bytes, as they will store the type of the destination variable automatically.

The form is:

```
{ INTEGER | POINTER } <integer specification>
     [ , <integer specification> ] ...  ;
```

An integer specification is:

```
<identifier> [*] [ <data specification> ]
 or
* <identifier>
 or
<identifier> [*] ( <integer literal> )
 to reserve <integer literal>+1 words of integer space
 for array subscripting or list use.
```

And a data specification is:

one or more integer literals or address constants of the form:

```
ADR ( <identifier> )
```

Some examples are:

```
INTEGER i,j,k 34, h OFF, table 1 2 3 4 5 6;
INTEGER room(20);
POINTER aptr,bptr, roomptr ADR(room);
POINTER iptr ADR(i), jptr ADR(i) ADR(j);
```

If the identifier is preceeded by *, it is defined EXTERNALLY. If it is followed by an *, then it is a GLOBAL variable which may be referenced by other program modules.

9.3. DOUBLE specification statement.

Double precision integer variables are specified with the DOUBLE statement. The form is:

```
DOUBLE <dpi spec> [ , <dpi spec> ] ;
```

where <dpi spec> is:

```
<identifier> [*] [ <data specification> ]
  or
* <identifier>
  or
<identifier> [*] ( <array size> )
```

The array size specification, if present will allocate (<array size>+1)*4 bytes of storage, as array subscripts start with 0.

A <data specification> is one or more integer literals, seperated by spaces.

9.4. The BYTE specification

BYTE variables are defined by the BYTE specification statement. Its form is:

```
BYTE <bytevar spec> [ , <bytevar spec> ] ;
```

where <bytevar spec> is:

```
<identifier> [*] [ <data specification> ]
  or
* <identifier>
  or
<identifier> [*] ( <array size> )
```

The array size specification, if present will allocate (<array size>+1) bytes of storage, as array subscripts start with 0.

A <data specification> is one or more integer or string literals, seperated by spaces.

9.5. REAL variable specification

Real variables are specified by using the REAL statement. The form is:

```
REAL <real specification> [ , <real specification> ] ... ;
```

where a real specification is:

```
<identifier> [*] [ <real literal> ... ]
 or
* <identifier>
 or
<identifier> [ * ] ( <integer literal> )
 to reserve real table space
```

Note that several real literals may be specified, seperated by spaces, to create real data tables. Some examples are:

```
REAL a,b,c,d 34.2 table -1.0 2.4 3.09 3.1E-4 ;
REAL alpha, beta, gamma, greek_sums;
```

If you follow the identifier with an *, then the variable is defined for the LINK editor to communicate to another module. To allow other subroutines to reference a real variable, you might use:

```
REAL pi* 3.14159;
```

A subroutine that wishes to use the above defined variable should place an * BEFORE the identifier name. A subroutine that references PI, above, would specify:

```
REAL *pi;
```

Note that data may not be specified when defining an external variable reference.

9.6. STRING Variable Specification

The STRING statement is used to specify string variables. The form is:

```
STRING <string specification> [ , <string specification> ] ... ;
```

where a string specification is:

```
<identifier> [*] [ <string data specification> ]
 or
* <identifier>
```

and a string data specification is either a string literal, or an integer literal. If an integer literal is used, a single character string having that value is defined. This is one way of specifying ASCII control characters.

Some examples are:

```
STRING name "MIKE", other_name, ctlc 03;
STRING alpha;
```

If the identifier is followed with an *, then the variable is defined externally for the LINK editor to communicate to other modules, so that they may reference the variable. An example is:

```
STRING company_name*  "XYZ Mfg. Co.";
```

To reference this string in another module, include the * BEFORE the identifier name. A subroutine that uses the above string would specify:

```
STRING *company_name;
```

Note that such external specifications can not specify data.

A string specification may also be an array dimension:

```
<identifier> [*] ( <array size> )
```

Note that no data may be assigned to a string array within the specification statement. Each element of the array takes two bytes and contains the buffer pointer address for that string element of the array.

10. EXECUTABLE STATEMENTS

Executable statements are those statements that will generate executable code for the processor to perform. An executable statement is performed when control is transfered to it.

Normal SPL programs specify some program label that is to receive intial control, or begin the program execution. From that statement on, each statement is executed in the order it physically occurs in the program, unless that statement transfers control to a label of another statement, (as GO TO does, for example).

If a statement can transfer control to a different statement label, it is considered a CONTROL statement. All other statements only transfer control to the next statement in the program when they finish their task. The most common example of this type of statement is the variable assignment.

10.1. The Assignment Statement

The assignment statement allows you to evaluate an expression and assign its value to a variable. The syntax is:

```
[label[*]:] <address expression> = <data expression> ;
```

The character "*" after the label specifies the label as a GLOBAL symbol. Other program modules can reference the label.

An address expression is a variable name or an indirect address expression that identifies the desired variable that is to be changed.

The data expression is evaluated, and the resulting value is assigned to the variable identified by the address expression. Any necessary mode conversions between real and integer will be performed. A string variable may not be assigned a numeric expression and a numeric variable may not be assigned a string expression.

Some examples are:

```
ALPHA=3.2 * SIN(C)^2.1 ;
@data_pointer=B+@(ADR(TABLE)+2);
C=D;
```

If the destination variable is not the same data type as the expression, the value of the expression will be converted to the type of the destination variable after the expression is completely evaluated.

10.2. GO statement

You may transfer control to any labeled executable statement with the GO statement. Its form is:

```
{ GO ¦ GOTO ¦ GO TO } <label address expression> ;
```

As an example:

```
GO test;
.....% other statements %
test: a=a+1;
.....
```

The GO statement will cause execution to continue with the statement labeled test:.

As another example, the use of pointer variables allows assigned GO TO branches:

```
ptr=ADR(labela); GO @ptr;
.....
ptr=ADR(labelb); GO @ptr;
```

The GO @ptr; statement will transfer control to either labela: or labelb:, depending on which label has been assigned to ptr.

10.3. The Computed GO Statement

SPL also offers a computed GO statement that lets an integer-valued expression select one of several program labels and GO to that label. The form of the computed GO is:

```
GO (<label> [ , <label>] ... ) <integer expression>;
```

Note that only program labels may be used in the label list. Address expressions and pointer variables may NOT be used. As an example:

```
GO (alpha,beta,gamma) control;
delta: ........
```

This statement will go to alpha if control=1. If control=2 then it will GO TO beta. Likewise, if control=3 then it will GO TO gamma. If control is less than one or more than 3, the statement labeled delta will be executed, and no branch will be taken.

10.4. IF...THEN...ELSE

The IF statement is the main decision making statement in SPL. It has the form:

```
IF <integer expression> THEN <statement> [ ELSE <statement> ]
```

Note that no ; is required after the IF statement since it is ended by another statement, which will have its own ;.

Any executable SPL statement may be used as the conditional statement. If you desire more than one statement to be performed as a result of the IF test, you may enclose them in {BRACES}. As an example:

```
IF cmd="GO" THEN {flag=1; GOTO process;}
            ELSE flag=0;
```

10.5. LINK statement

Normally, when a program ends, the computer operator must type in a command to specify what should be done next. The LINK statement lets an SPL program send a command to the operating system directly, and then ENDs the SPL program to perform that command. The form of the LINK statement is:

```
LINK <string expression> ;
```

As an example, the statement:

```
LINK "DIR 2/" ;
```

will end the program and display the directory of drive 2. Note that the string expression may be any valid operating system command.

10.6. PROCEDUREs and the DO statement

Often, a group of statements must be performed at several different places in a program or performed many times. SPL allows this within a program by setting up the group of statements as a PROCEDURE, and then DOing the procedure.

The form of a procedure statement is:

```
[<statement label> : ] PROCEDURE <identifier> [*] ;
```

The optional asterisk allows the procedure to be executed from outside the module it is defined in.

A procedure has the form:

```
PROCEDURE statement
other SPL statements
END;
```

If control is transfered to the PROCEDURE statement's label, or "falls through" from the statement before the procedure, then the procedure will be performed as if PROCEDURE and END; were not present. This allows the procedure to be executed in-line.

To perform the procedure, the DO statement is used. Two forms of the DO statement are allowed. The first is:

```
DO <procedure identifier> ;
```

This form will perform the procedure and then return to the statement following the DO. Note that the identifier used by DO is the identifier that follows the word PROCEDURE, not the statement label of the PROCEDURE statement.

The second form allows looping and performing the procedure several times. Its form is:

```
DO <procedure identifier> :
   <index address expression> = <initial value expression>
   , <limiting value expression> [ , <increment expression> ] ;
```

The index variable is set equal to the initial value, and that value is compared to the limit value. If the increment is positive, then index must be <= limit for the DO to execute the procedure. If the increment is negative, then the index must be >= limit for the DO to execute the procedure. Each time the procedure is performed, the increment, or 1 if no increment is specified, is added to the index, and the test is made again. This allows the procedure to be performed several times with one DO statement.

All expressions are evaluated only once, at the start of the DO statement, and changes to variables referenced in those expressions by the procedure will not affect the execution of the DO statement.

As an example:

```
PROGRAM loop_demo;
INTEGER ix;

PROCEDURE test;
PUT "test procedure. IX=",ix;
END;

loop_demo: DO test: ix=1,4;
END;
```

will print out:

```
test procedure. IX=1
test procedure. IX=2
test procedure. IX=3
test procedure. IX=4
```

If the procedure identifier is omitted from the DO statement, then the procedure immediately follows the DO statement and ends with the next END; statement:

```
DO : I=1,3;
PUT I;
END;
```

will print out:

```
1
2
3
```

10.6.1. Exiting from a DO loop

Normally, a procedure, whether specified in a procedure statement or by an in-line procedure as shown above, MUST exit through the associated END; statement. To exit a DO procedure without going through the END; statement, use the EXIT; statement. This will abort any additional executions of the calling DO loop. As an example:

```
DO : I=1,10;
put i;
if I=last then exit;
put "more to go";
end;
```

This would perform 10 iterations of the code, unless the variable LAST were less than 10. If so, the DO statement would be aborted when the EXIT; statement is executed.

11. CONSOLE output of an integer as a character

Direct console output may be used to send a control sequence to the console or echo characters read in using the CONIN function. The format is:

```
CONOUT <integer data expression> ;
```

For example, to send a backspace to the console:

```
CONOUT 8;
```

12. The COPY statement

Sometimes you will wish to use exactly the same statements in several different programs or subroutines, for example, in defining a disk data record, or common variables. You can put the SPL source code into a file and then include it in each SPL module with the COPY command. Its form is:

```
COPY  <file name>
```

Note that there is no ; at the end of the COPY statement. Also, nothing can follow the file name on the same line, and the COPY statement must all be on the same line. It will be replaced with the text of <file name> during compilation. An example is:

```
COPY setup.spl
```

13. TRACE facilities

SPL provides two forms of the TRACE statement to assist in debugging your programs. TRACE LABELS will display a message on the console for each statement label that is executed, to show you the path your program is taking.

The second form will trace the execution of each machine instruction as it is executed, showing you the instruction mnemonic, address, the contents of any registers, and the results. It is helpful to have a listing of the compiled assembly language code handy to interpret what you will see. If you did not specify an intermediate file on the SPL command, then the compiled assembly code will be in file TEMP1$ for you to print out.

The form of the TRACE statement is:

```
TRACE { LABELS | EXECUTION } { ON | OFF } ;
```

Trace statements cause the final program to run much slower and require more memory space.

14. Embedded ASM language statements

SPL allows you to include assembly language statements in-line with your SPL statements by using the ASM statement. It starts with the keyword ASM. The next character following the M of ASM is ignored as a delimiter. The rest of the line will be the assembly language statement.

As an example:

```
PROGRAM asmtest;
INTEGER i,j;
asmtest:
GET i,j;
% Shift I right J bits %
ASM shift  mov i,r0   get i
ASM        srl r0,1   shift it
ASM        mov r0,i   put it back
j=j-1;
IF j<>0 GO shift;
PUT i;
END.
```

Registers R0 through R7 may be used freely, although any SPL statement may change them. Register R8 is a base index register for subroutine parameters, R9 is a subroutine parameter address stack, and R10 is the utility stack pointer. R11 is used for BL instructions, R12 is used during PUT statements for editing, and R13, R14, and R15 are used by BLWP instructions. Any assembly language statement may use any register, but r8,r9, and r10 must be left unchanged when the next SPL statement occurs. You must consider that any SPL statement may change all registers.

15. TEXT INPUT/OUTPUT

SPL Provides two forms of Input/Output statements. Text oriented I/O is used for console and disk text file use, and DIRECT I/O is used to access disk or device files in multiples of 128 byte sectors, without any editing.

Text I/O involves editing operations that convert data between ASCII character representations and the internal forms data are stored in.

The GET statement is used for text input, and the PUT statement is used for text output. Also, you can GET FROM a string, scanning the string and converting it into other data forms. You may also PUT TO a string using the PUT editing facilities to create text strings.

GET and PUT may also use disk text files by using the OPEN and CLOSE statements to select disk files.

The forms of the GET statement are:

```
% Read input from the console %
GET <input edit list> ;

% Read input from a string in memory %
GET FROM <string expression> : <input edit list> ;

% Read from a disk text file %
GET ( <file descriptor> ) <input edit list> ;

% Read from a non-string location in memory %
GET FROM ( <address expr>,<length expr> ) <input edit list>;
```

This last form is used to edit text contained in direct disk I/O buffers into normal variable forms. This allows SPL to work with data stored in almost any format.

The address expression gives the starting location in memory of the input text, and the length expression is equal to the maximum number of characters to be scanned. There is no requirement on the data format except that an ASCII NUL (00) will always force input editing to terminate as if that character were the end of the "line". An ASCII RETURN (13) will also effect an end of line condition. In all other cases, the line will be ended by the number of characters specified being scanned.

The forms of the PUT statement are:

PUT [:] <output edit list> ;

PUT to the console device. A colon immediately after the keyword PUT causes the output to remain on the current line after the end of the put statement. This is used to provide input "prompts" for following GET statements.

PUT TO <string address expression> : <output edit list> ;

The line of text resulting from the PUT editing is assigned to the specified string. No carriage return is at the end of the string.

PUT (<file descriptor>) <output edit list> ;

The text line, with carriage return, is sent to the specified text file.

A file descriptor is:

<integer file pointer> [\ <eof and error label>]

All disk files are referenced through the use of an integer pointer variable assigned by the OPEN statement. The optional eof and error label is the program label to branch to if an eof or error condition is detected during I/O.

PUT TO (<address expr>,<length expr>)<output edit list>;

This allows you to edit variables into a string of text and then place that text at any selected memory address, without storing it in the form used by SPL string variables. This is useful in working with direct I/O disk buffers and other structured data organizations.

The number of characters specified will be stored in memory. No return or line feed or Null will be appended at the end of the line. If the resulting output string is shorter than the number of characters specified, then blanks will be appended on the right. If longer, characters on the right will be truncated.

15.1. Input Edit Lists

During input text editing, the source text line is scanned under control of the input edit list. The form of this list is:

    <input edit term> [, <input edit term>] ...

There are several forms of input edit terms.

15.1.1. Integer or Real variable address references

If you specify the address of a numeric variable, the input text line is scanned from its current position, skipping blanks. If a numeric literal is encountered, its value is assigned to the specified variable. The scan cursor is positioned to the character following the character used as a delimiter to end the literal. If no literal is encountered, the cursor is positioned to the first non-blank character.

As an example, If the input line is:

    123,19XYZ

is read by the statement:

    GET I,J,further editing terms;

Then I is assigned the value 123, J is assigned the value 19, and the cursor is positioned to the character Y in the data for further editing.

15.1.2. Integer or Real Fixed length references

You may also specify a fixed field width specification. That many characters will be scanned for a valid number. No matter how many characters of that field are used by the resulting number, the scanner will be positioned after the entire field for the next input edit term. For fixed field specifications only, leading zeroes do NOT indiciate hexidecimal notation. This allows a string of digits to be broken into seperate components. For example:

```
GET I\2,J\3;
```

will read the data 01045 and set I=1 and J=45.

On real variables, only the field width may be specified. No decimal place default is available.

15.1.3. String Address Specifications

If a string address is specified, then the rest of the input line from the current cursor position is assigned to the string variable.

For example, the statement:

```
GET LINE;
```

will read an entire line into the string LINE.

15.1.4. String Address with length specifications

If the specification is of the form:

<string address expression> \ <integer expression>

The integer expression will determine the number of characters, starting at the current cursor position, to assign to the string variable. Also, the cursor will be advanced by that number of characters. If there are not enough characters remaining on the input line, then the result string will be padded with blanks on the right end to make it the desired length. If the cursor is at the end of the input text, then a null string will be assigned to the string variable.

For example, the input text:

```
1234567890 ALPHA
```

when read by the statement:

```
GET S\4,T;
```

where S and T are string variables will result in S being set to "1234" and T being set to "567890 ALPHA".

15.1.5. String address expression with delimiter

You may also specify that a string variable is to be set to the characters starting at the current cursor position and continuing until a certain character occurs. This character is called the cursor. It is not part of the assigned string value, and the cursor is positioned after it for further scanning. If the delimiting character does not occur, the cursor position remains unchanged, and the string variable is set to a NULL string.

This is specified by:

<string address expression> 'c

where c is the delimiter character. For example, the input line:

01234567890 ALPHA: BETA

When read by the statement:

```
GET R,S':,T;
```

Where R is a real number, and S and T are strings, results in R=1234567890, S="ALPHA" and T=" BETA".

15.1.6. Tab to position

The scan cursor can be set to any position either forward or backward by the TAB term:

<integer expression>

The value of the expression determines the next column to be scanned. The first column of the input line is numbered 1.

15.1.7. Mark Column Position

The current column number that is pointed to by the scan cursor may be saved in an integer variable by using the term:

? <integer address expression>

Note that this may be used to scan the same information twice:

```
GET S':,?IPOS,T,#IPOS,R;
```

The above example, assuming that S and T are strings, IPOS and R are integers, will let you scan to the position of a delimiter, and then set a string equal to the rest of the line, and then return to the position after the delimiter and scan for an integer value.

15.1.8. Skip columns

The specification:

```
\ <integer expression>
```

causes the input scan cursor to advance the indicated number of position, + or -, from its current position. If this would take it past the end of the input line, then the cursor is positioned to the end of the input line. If this would cause the cursor to be positioned before the start of the line, then the cursor will be set to the first character in the line.

15.1.9. Skip to character pattern

The specification:

```
= <string expression>
```

will cause the cursor to advance until the string expression has been matched in the input line. In that case, the cursor will be positioned to the character after the matched string occurance. If it does not occur in the remaining part of the input line, the cursor is not moved.

15.1.10. SKIP specified character

It is possible to cause all occurances of a character from the current scanner position to be skipped, by specifying:

```
-c      skip any occurances of the character "c"
        starting at the current scanner position
```

Skipping of characters stops when any character other than "c" is found. The following example would get a name, after skipping leading blanks:

```
get - ,name;
```

15.1.11. Term Class Delimited string specifications.

In SPL you may specify that a string is to be read consisting of characters that fit (or do not fit) the rules for certain selected types of terms. The specification is:

```
<string variable reference>=<testname>
     or
<string variable reference>#<testname>
```

where = specifies all characters that fit the desired term type, and # specifies only characters that do NOT fit the desired term type.

The following term types are provided for <testname> :

DIGIT	only character 0 through 9 are accepted
ALPHA	only letters of the alphabet are accepted
ANCHR	letters or digits are accepted
LABEL	The first character must be a letter, but any folowing characters may be letters or digits.
INUM	characters in a sequence making an integer literal are accepted. This includes leading zeroes allowing hexidecimal specifications.
STLIT	The first character must be ". All characters after that until the next " are included. Two " characters in a row will be included and not end the string.

The test names are actually the names of assembly language subroutines, so you may write your own tests if desired. The rules for such a subroutine are:

Only r1, r2, and r3 may be used. r3 will be set to zero for the first test of a string, and may be used to count characters if desired. The subroutine will be called with a BL instruction for each character, and must test the character addressed by *r4. Before returning, r1 must be set to either OFFFF if the character is to be included, or zero if it is not to be included in the string.

15.1.12. IF...THEN...ELSE

SPL also allows conditional formatting by letting you use the IF THEN ELSE logic construct inside edit lists. It is used in the following manner:

```
IF <integer expression> THEN <input edit term>
   [ ELSE <input edit term> ]
```

To allow more than one edit term per condition, you may use {braces} to contain an input edit list as if it were a single input edit term. An example of the use of this might be to allow two different types of "commands" to be input to a program. One form might be:

```
TEXT:35,HELLO
```

to specify TEXT on line 35 of a doccument, and

```
VALUE:35,192
```

to specify that two values are to be read in.

The following GET statement will do this for you:

```
GET CMD':, IF CMD="TEXT" THEN {NR,S} ELSE {I,J} ;
```

This will assign NR and S if the command is "TEXT", and will assign I and J if it is not.

15.2. Output Edit Lists

Output editing builds a line from specifications given in the output edit list. The form of an output edit list is:

```
<output edit term> [ , <output edit term> ] ...
```

There are several forms of output edit terms.

15.2.1. String data expressions

A string expression may be output by using the form:

```
<string data expression> [ \ <integer length expression> ]
```

If a length is specified, only that many characters of the string expression will be output. If the string is too small, it will be padded with blanks. If no length is specified, then the entire string will be output.

When a fixed length field is specified for a string, the string is normally left-justified in that field. Left or right justification may be specified by following the \ character with < for left justification, and > for right justification. An example, used for column headings, is:

```
PUT "NAME"\<20,"AMOUNT"\>10;
```

15.2.2. Integer data expressions

Integer data expressions may assume the following forms:

```
<integer data expression> [ \ <integer length expression> ]
        or
* <integer data expression> [ \ <integer length expression> ]
```

The leading * specifies hexidecimal output. If it is not used, decimal form is used. If no length is specified, then only the number of characters necessary are used, followed by one blank. If a length is specified the integer value is right-justified in a field of the specified size. Blanks are used to fill a decimal field, and zeros are used to fill a hexidecimal field.

An example, assuming that I contains 10 and J contains -1, is:

```
PUT I,J,*I,J\6;
```

This will output the text "10 -1 0A   -1".

15.2.3. Real data expressions

Real valued expressions are output with the form:

```
<real data expr> [ \ <integer size expr> [ : <integer dp expr> ] ]
```

If no field width is specified, then only the necessary number of characters are used to represent the value. If a fixed field size is specified, then the value will be right-justified in the field and left-padded with blanks as necessary. If the number of decimal places is not specified, then zero decimal places will be used. Scientific notation will only be used if the number can not be expressed any other way. A blank will follow a real numeric field if no field length is specified. If an integer data expression is used with a decimal places specification then the decimal point will be edited into the field in that position arbitrarily.

As an example, if R=1.23456 then:

```
PUT R,R\4,R\4:2;
```

will output the text string "1.23456   1 1.23".

15.2.4. Tab to selected column

The term:

```
# <integer data expression>
```

allows you to specify what character position in the edit line will be used by the following edit term. Column positions are relative to the start of the edit list, not to physical columns on the I/O device, since it is possible with PUT to start another "line" of output on the same physical line used by an earlier PUT: statement.

15.2.5. Skip columns

The specification:

```
\ <integer specification>
```

will cause the desired number of columns, + or -, to be skipped over, and the output column pointer repositioned. If this would position it beyond the start or end of the edit line, then the column pointer is positioned at that end.

15.2.6. Mark present column position

The term:

```
? <integer address expression>
```

allows you to save the current edit cursor position in an integer variable, in the same way that such a term is used by the GET statement. This can be used to line up information with free form output:

```
PUT NAME$," ",?NPOS,ADRS0;
PUT #NPOS,ADRS1;
PUT #NPOS,PHONE_NUMBER;
```

so that the result might look like:

```
MARK HAMMERSCHLINGER  1234 W. 6th St.
                      Eureka, ZZ 99900
                      (602) 279-4545
```

15.2.7. IF...THEN...ELSE

The PUT statement also allows you to use the IF THEN ELSE logic construct to provide variable formatting, in an identical manner as used by GET. The format is:

```
IF <integer expression> THEN <output edit term>
     [ ELSE <output edit term>]
```

You may use { <output edit list> } to allow more than one term to be used with THEN or ELSE.

An example of this might be:

```
PUT NAME\14,IF CREDIT=BAD THEN "pick up card" ELSE LIMIT\7:2;
```

could print out lines like:

```
ARTHUR J. JAY 1000.00
BORG ALLISON   500.00
BLACK PETE    pick up card
MICKEY MOUSE  5000.00
```

16. DIRECT FILE INPUT / OUTPUT

Disk files may be accessed using GET and PUT for text files, and READ, WRITE, and SEEK for direct access files, by first opening the file. OPEN locates the file on the disk and prepares pointers and buffers in memory for use with the file.

An integer variable is used as an address pointer to this "file buffer" area of memory. The OPEN statement assigns a value to this integer file variable, and that value should not be changed until after the file is CLOSED.

Closing a file forces any data in the buffer out to the file and releases the buffer space to free memory for other use. You must close all files that you use before ENDing a program. SPL will NOT close them for you.

16.1. The OPEN Statement

The syntax of the OPEN statement is:

```
OPEN <integer address expression>
FOR <string file name expression>
{ INPUT | OUTPUT | DIRECT }
[ \ <error address expression> ;
```

INPUT specifies a text input file to be used with GET. OUTPUT specifies a text output file to be used with PUT. DIRECT specifies a binary file used with READ, WRITE, and SEEK.

16.2. The CLOSE Statement

```
CLOSE <integer address expression>
      [ \ <error address expression> ] ;
```

If no error address is specified, then CLOSE will ignore the error. This is done so that all files may be closed, even if one CLOSE statement has an error.

As an example to read lines of text from the file "MYDATA" and print them on the console, until the end of file, you might use the program:

```
PROGRAM type;
POINTER i; STRING s;
type: OPEN i FOR "MYDATA" INPUT;
loop: GET (i\done) s;
      PUT s;
      GOTO loop;
done: CLOSE i;
      END.
```

The keywords appear in uppercase and identifiers in lower case for clarity. This is not required.

16.3. The READ and WRITE statements

Binary file I/O is performed by using the READ and WRITE statements. A buffer space sufficient to hold all of the sectors read or written at one time must be provided. The buffer will contain the exact data that is on the disk. No formatting or other changes to the data will be made by READ or WRITE. Pointer variables and structures may be used to access data contained in the buffer. Disk I/O buffers may be ALLOCATEd and used through a pointer variable in the READ or WRITE statement.

The format of a READ or WRITE statement is:

```
READ <file reference> <buffer address reference>
     [ , <integer sector count expression> ] ;

WRITE <file reference> <buffer address reference>
     [ , <integer sector count expression> ] ;
```

Where <file reference> is:

```
( <file identifier variable reference>
  [ \ <error or EOF branch address expression> ]  )
```

A file indentifier variable is a pointer variable that is used by SPL to relate the information contained in the OPEN statement to the READ and WRITE subroutines.

As an example of direct disk I/O, the following program will copy information from FILE1 to FILE2.

```
PROGRAM directcopy;
STRING name1 "FILE1", name2 "FILE2";
POINTER fromfile, tofile;
INTEGER sectorbuffer(128);

directcopy:
OPEN fromfile FOR name1 DIRECT;
OPEN tofile FOR name2 DIRECT;
loop: READ (fromfile\done) sectorbuffer;
      WRITE(tofile) sectorbuffer;
      GOTO loop;
done: CLOSE fromfile;  CLOSE tofile;
      END.
```

16.4. The SEEK statement

The seek statement allows you to position a file to any desired sector for the next READ or WRITE. Sectors are numbered starting with 0. The format is:

```
SEEK ( <file reference> ) <integer sector expression> ;
```

The statement:

```
SEEK (file\error) 25;
```

Tells the operating system to seek to the 25th sector of FILE, and if an error occurs, to GO TO ERROR.

16.5. CREATE files

To create a file, use:

```
CREATE <string expression for file name>
       [ ,<integer expression for file size in sectors > ]
       [ \ <error address> ] ;
```

If no length is specified, the file size will be determined by the operating system in use. Under MDEX, a default file size will be assigned. Under NOS, the file will be automatically created as it is written. If the file already exists, the error exit will be taken.

16.6. DELETE file

To delete a file, use:

```
DELETE <string expression for file name>
       [ \ <error address> ] ;
```

16.7. RENAME file

To rename a file, use:

```
RENAME <string expr for old file name> TO
       <string expr for new file name>
       [ \ <error address> ] ;
```

example: RENAME "master" TO "mbackup";

16.8. File Size Function FSIZE

The function FSIZE will return the number of sectors allocated to a specified file. The argument is a string expression specifying the file name. As an example:

```
NSEC=FSIZE("temp2$")
```

will place the numbr of sectors allocated to file TEMP2$ into the integer variable NSEC.

16.9. Error Handling with SPL

If a file I/O error occurs, and an address for handling the error is not specified by the file I/O statement, then and error message is generated and the program terminates. The exception to this is CLOSE. If there is an error closing the file and no error address is specified, then the error is ignored. This is because several files may be closed at the end of the program. An error closing one file should not prevent the program from trying to close the other files.

In your error handling routines, you may use the function IOSTAT to get the error status code. This integer value is identical in meaning to the error codes specified in the Termination error messages. The first two digits of the error code specify the type of statement that was executing when the error was detected. The next two digits specify the Operating System error code that initiated the error response. These codes are specified in the appropriate operating system manual. Some of the more common codes are listed on the next page.

*** ERROR CODES RETURNED IN IOSTAT and ERROR MESSAGES

0	no error from system - used to indicate actual write length < desired write length.
1	end of file
2	Unrecoverable device error
3	file not found
4	bad file index
5	bad file name syntax
6	bad subfunction (probably an SPL bug)
7	file not executable (on a Link statement)
8	too may files open
12	Not enought room on disk for create
13	No room in directory for create
18	File already exists - can't create

The statement type codes are as follows:

01xx	Direct Read
02xx	Direct Write
03xx	Direct Seek
04xx	Direct Open
05xx	Text Output Open
06xx	Put Text Write
07xx	Text Input Open
08xx	Get Input Read
09xx	Direct Close
0Axx	Text Close
0B00	No Memory for Buffer Allocate (See ALLOCATE command)
0Cxx	Create file error
0Dxx	Delete file error
0Exx	Rename file error
0E00	Rename - drives not the same
0Fxx	Size file FUNCTION error on directory read

As an example, error 0801 is an end of file while reading a text file with a GET statement.

17. DATA ORGANIZATION TOOLS

SPL provides several statements that allow data to be organized in a variety of ways.

17.1. Structures

A structure is a dummy map of an area of memory that describes the relationships between different types of data. This map may be placed anywhere in memory because the start of the structure is treated like a pointer variable.

As a simple example, consider:

```
POINTER MAP;
STRUCTURE (MAP) I 0, J 2, K 4, REAL A 6, B 14;
```

Now if you say MAP=1000 then whenever you use any of the variables in the structure, they will mean the location relative to 1000 indicated by their position in the structure.

```
I is at 1000,  J is at 1002, and K is at 1004
A is the real number at 1006, and
B is the real number at 1014.
```

If you then say MAP=2000 then

```
I is at 2000,  J is at 2002, and K is at 2004
A is the real number at 2006, and
B is the real number at 2014.
```

Structures are very powerful, because they let you treat data in memory in different ways. You might store data on a disk as an array of integers to speed up I/O, but work with it in your program with a structure. And you can determine different types of records with one structure, and then start referencing the data by a structure that is appropriate to that record type.

The format of a structure statement is:

```
STRUCTURE ( <pointer address expression> )
     <structure term> [ , <structure term> ] ;
```

where a structure term is:

```
[ INTEGER ¦ DOUBLE ¦ BYTE ¦ REAL ¦ STRING ]
 <identifier>  <integer literal>
 [ , <identifier> [ ( <integer literal> ) ] ] ...
```

If you use an indirect reference with a structure variable, then the data in the structure will be treated as a pointer to the actual data. As an example:

```
POINTER MAP;
STRUCTURE (MAP) I 0, J 2, BPTR 3;

BPTR=adr(A);
I=@BPTR;
```

I will contain the data that was stored in A.

Use of structure elements as pointers should not be mixed with disk I/O, as the data that would be stored on the disk is the address of that data at the time the file was written, and not the data itself. It is acceptable to use pointers to assign data to structure elements:

```
POINTER MAP, BPTR;
STRUCTURE (MAP) I 0, J 2, K 4;
INTEGER A;

BPTR=adr(A);
I=@BPTR;
```

Note the difference: BPTR is not within the structure. This restriction only applies when using structures to manipulate disk data records.

An analogous situation occurs with strings. Since string variables actually store a pointer address of a string data buffer, strings within a structure must be handled differently than normal if the structure refers to a disk data record:

```
POINTER DBFR;
STRUCTURE (DBFR) I 0, J 2, BYTE NAME 4;
```

To store a string within the structure starting at NAME, use the form of PUT that allows address specification:

```
PUT TO (NAME,NAMESIZE)"string data";
```

This situation will be cleared up in a future release of SPL.

17.2. Buffer Alocate and Release

SPL allows you to dynamically control memory usage. By using pointer variables and structures, you can allocate a buffer of memory space, and release it for other use when it is no longer needed. The format of the ALLOCATE statement is:

```
ALLOCATE <pointer address expr> , <integer byte count expr>
         [ \ <executable address expression> ] ;
```

The release statement is:

```
RELEASE <pointer address expression> ;
```

As an example, the following program will allocate a buffer and use it for saving a line of input text:

```
PROGRAM buffers;
POINTER bptr;
buffers:
ALLOCATE bptr,80;
GET @bptr;
PUT @bptr;
RELEASE bptr;
END.
```

Use the pointer variable as a normal integer for allocation and release of buffers, and as an indirect address to access the data in the buffer.

Normally, if there is no memory to allocate a buffer, an error message is generated. However, you may specify a place in your program to handle this condition. An example is:

```
ALLOCATE bptr,80\nomemory;
.....
nomemory: % come here if no memory to allocate %
```

17.3. Queues

A QUEUE is a linked list of buffers that are usually set up using ALLOCATE and RELEASE. There is a set of pointers to the first and last buffer, called a queue head, that is declared like a type of variable, using the QUEUE statement:

```
QUEUE  <queue name> [ , <queue name> ] ... ;
```

Once defined, the QUEUE variable may be used with the statements QINSERT, QPUSH, and QREMOVE.

The QINSERT statement places a buffer at the end of the linked list. This is appropriate for using a queue as a FIFO buffer list. Its form is:

```
QINSERT <pointer variable expression> IN <queue variable> ;
```

The QPUSH statement places a buffer at the start of the linked list. This is appropriate for using a queue as a FILO buffer list, or a stack of buffers. Its form is:

```
QPUSH <pointer variable expression> TO <queue variable> ;
```

The QREMOVE statement removes the first buffer from the queue and places its address into the designated pointer variable. The format of the remove statement is:

```
QREMOVE <pointer var expr> FROM <queue variable> \ <empty adrs> ;
```

If there are no buffers in the queue, the program will GO TO the specified <empty address>.

The first two words of any buffer used with a queue must be reserved for linked list pointers. The first word points to the address of the previous buffer in the list, and the second word points to the address of the next buffer in the list.

17.4. Lists

SPL provides statements for processing lists of dissimilar items by storing the address of each item, and a cursor that selects which item shall be processed next. Lists are considered an organization of pointer variables.

The statements used with lists are:

POINTER	-	to specify the list
LIST	-	to load the entries into the list
APPEND	-	to add more entries to a list
NEXT	-	to scan the items in a list
START	-	to restart the scan at the beginning of a list

17.4.1. Specifying a List

To declare a list, declare a pointer variable table containing sufficient room for all the items in the list plus three words of overhead. An example is:

```
POINTER names(33) % 30 items + 3 overhead % ;
```

17.4.2. Loading Data into a List

A list is set up with the LIST statement, which is executable. This statement assigns the values of address expressions to list entries. Its format is:

```
LIST <listname> = <adr expr>  [ , <adr expr> ] ... ;
```

An example, using the above declared list NAMES, is:

```
LIST names=a,b,@c,@(d+20),@(0C000);
```

You may append addresses to the end of a list to build a list dynamically, as when buffers are allocated and then stored in a list, with the append statement:

```
APPEND <address expr> TO <list variable adrs expr> ;
```

An example of the use of append with the above list is:

```
ALLOCATE ptr,1024;
APPEND ptr TO names;
```

17.4.3. Scanning a List

Once a list has been created, the NEXT statement is used to sequentially access elements from the list:

```
NEXT <pointer adrs expr> = <list adrs expr>
     [  \   <end of list branch address>  ]   ;
```

An example is:

```
NEXT ptr=names\alldone;
```

If the end of list branch address is ommitted, the NEXT statement will automatically start the list scan over upon reaching the end of the list. If the end of list exit is taken, the next list access using NEXT will access the first entry in the list.

Sometimes you will wish to start processing a list from the start, even though you have not finished the list. You can do this with the START statement:

```
START <list adrs expr> ;
```

As an example of using a list, the following program will read a line from the console, and search a list for a string that matches, and return the subroutine address that relates to the matched name:

```
program listdemo;
pointer ptr, names(11);
string n1 "line", n2 "circle", n3 "box", n4 "help";
string line;

listdemo:  % first set up the list %
     list names=n1,subr1,n2,subr2,n3,subr3,n4,subr4;
doit: % now get a "command" to look up %
     get line;
     start list;
trynext:
     next ptr=names\notfound;
     if line=@$ptr then found
     next ptr=names  % skip associated subr entry % ;
     goto trynext;
notfound:
     put "What?";  goto doit;
found: next ptr=names;  % get associated subroutine %
     call @ptr;  % perform the subroutine %
     goto doit;  % go get next command %
end.
```

In the above example, subr1, subr2, subr3, and subr4, are assumed to be subroutines that process the indicated commands, and must be defined in a different program module.

17.5. Stacks

SPL allows you to have First In / First Out data stacks in a high-level language. You may declare as many named stacks as memory will hold. By using the PUSH and POP statements to enter and remove data from the stacks, recursion, re-entrant code, and multi-level co-routine programing becomes possible.

Stacks are specified with the non-executable statement STACK:

```
STACK  <identifier> ( <integer byte count> )
     [ , <identifier> ( <integer byte count> ) ] ...  ;
```

Specify the total number of bytes of data that may be stored on the stack. Integers take 2 bytes, pointers take 2 bytes in the present release, but will require 4 bytes in future releases. Real values require 8 bytes each.

Data may be pushed to stacks with:

```
PUSH (<stack identifier>) <variable address list>;
```

and removed from a stack with:

```
POP (<stack identifier>) <variable address list>;
```

As an example, a recursive subroutine needs a data workspace for each nested level of call. The workspace may be referenced from a structure, and the buffer address pushed to a stack each time the routine is called.

```
Subroutine recurs;
pointer workarea;
stack wastack (100);
structure (workarea) .............  ;
integer wasize 256 ; % needed workarea size in bytes  %

recurs: push (wastack) workarea;
allocate workarea,wasize;
....
....
release workarea;
pop (wastack) workarea;
end.
```

18. SUBROUTINES

If every part of a program had to exist in one text file and be compiled at one time, programs would often grow unmanageably large. Also, variable names might often be used in different places in different and incompatable ways.

Subroutines solve this problem. You may write your program in smaller, easy to edit pieces, and test each part seperately. You can build a library of common subroutines that only need be LINKed into your program for use.

A subroutine has the general organization:

```
Options statements, if any
The SUBROUTINE statement
........
SPL statements
........
END.
```

The SUBROUTINE statement has the following form:

```
SUBROUTINE <identifier> [ <parameter list> ] ;
```

where the parameter list is:

```
( <variable identifier> [ , <variable identifier> ] )
```

The identifier that follows the SUBROUTINE keyword is the name of the subroutine, and is also the statement label of the first statement to be executed. Such a statement label is called an ENTRY. A subroutine may have several entry labels, if you define any extra entry labels with the ENTRY statement:

```
ENTRY <identifier> [ <paramter list> ] ;
```

Parameters are "dummy" variables that are specified each time the subroutine is CALLed. They must be specified inside the subroutine with the appropriate type statement, but no data may be specified in the subroutine's type statements for those variables.

ENTRY statements are placed at the location where that ENTRY code starts. They are "skipped" by in-line code. If the parameters for an entry are the same as for the subroutine, they need not be listed.

A main program or another subroutine may CALL a subroutine to cause it to be executed. The format of the CALL statement is:

```
CALL <subroutine entry adrs expr> [ <call parameter list> ] ... ;
```

where the call parameter list is:

```
( <parameter> [ , <parameter> ] ... )
```

and each parameter may be:

an address expression or
a literal of the correct type

Also, subroutines may use variables that are defined externally using *. See the description of INTEGER, REAL, and STRING statements for more information.

As an example of a subroutine's usage:

```
SUBROUTINE message(query,answer);
STRING query,answer;
STRING default "none";
message:
  PUT: query;
  GET answer;
  IF LEN(answer)=0 THEN answer=default;
END.
```

The main program that might use this is:

```
PROGRAM main;
STRING name,adrs1,adrs2,phone;

main:
  CALL message("What is your name?",name);
  CALL message("What is your street address?",adrs1);
  CALL message("City, State, and Zip code?",adrs2);
  CALL message("What is your phone number?",phone);
  ........% use the information just entered %
  END.
```

Note that the subroutine returns to the CALLing program when the subroutine's END. statement is executed.

18.1. Assembly Language Subroutines

SPL can call subroutines written in assembly language if certain rules are followed. First, certain registers have specified uses in SPL, and either must not be changed, or they must have their values restored on return.

The CALL statement generates the code:

```
    mov     r8,*r9+      save old parameter pointer
    mov     r9,r7        save new parameter pointer

    <parameter addresses are pushed to the R9 stack>

    bl      subroutine   call the subroutine
```

At the subroutine entry address, the following code must exist:

```
            jmp     $+6         (if multiple entry point)
 subname    mov     r11,*r9+    save return address
            mov     r7,r8       R8 now references parameter address list

            <subroutine code. R8, R9 must not be changed>
            <R10 is a utility stack, and may be used if
             it is restored before the subroutine ends>

            b       ~~subrt$~~ sbrt$   exit the subroutine.
```

To use parameters, you should define equates for each parameter. The SUBROUTINE statement:

```
            SUBROUTINE alpha(a,b,c,d,e);
```

will generate the code:

```
            a       equ     0
            b       equ     2
            c       equ     4
            d       equ     6
            e       equ     8
```

since each parameter address takes two bytes. To reference the address of a parameter, use the EQU label as an indexed R8 reference:

```
                    mov     d(r8),r0    Get D address
```

To get the actual variable data, an indirect reference must be made:

```
mov     D(r8),r0
mov     *r0,r0      D data now in r0
```

A subroutine may use BL or BLWP instructions without concern for changing registers, as they are not in use during a CALLed subroutine's execution.

The following code sequence will tell you how may parameters have been supplied by the CALL statement:

```
mov  r9,r0
dect r0
s    r8,r0
srl  r0,1
```

The number of parameter addresses is now in r0.

Since SPL uses the same variable names in the generated code as in SPL, externally defined variables may be referenced using the link editor. Variables defined in the assembly code should have their label followed by an *:

```
xref*  data  01234
```

Any undefined label in assembly code automatically generates an external reference for the LINK editor to resolve.

18.2. CODE modules

It may be desireable to break your main program into smaller pieces that do not need the overhead of parameter passing structures and the call statement. If you wish to write sections of code that are reached by GOTO or DO statements only, then you may compile them in a CODE module. its form is:

```
CODE <module name>;
...
...
... spl statements
...
END.
```

You should declare any labels that you wish to goto or do with an *, as described in the appropriate section of this manual. All variables that are to be shared between the code module and the main program should also be defined using * for local and external definitions. Any variables defined in the code module and not specified with an * will be local to the code module and not available to external code or program modules.

19. Preliminary Information on Future Releases of SPL

SPL is a growing language. It has been planned out to a much larger extent than the current implementation. The following information is provided so that you may see in what direction future releases will be moving. Since it is preliminary, all information contained in this section is subject to change.

19.1. Features to be implemented in SPL 1.40

The following features are expected to be implemented in release 1.40 of SPL:

19.1.1. Virtual Disk File Read/Write/Seek

Disk file access specified by a byte position in the file, with any number of bytes per record.

19.1.2. TEXT Variables

SPL string variables are dynamically allocated text buffers. While this conserves memory, there are instances where it creates problems, as in disk buffer structures. TEXT variables will be non-dynamic strings with a specified maximum size, that will work correctly, (and quickly), within disk buffers and structures.

19.1.3. DO WHILE and DO UNTIL

The DO programing construct will be expanded to allow the structured forms WHILE and UNTIL.

19.1.4. AVMEM function

The AVMEM function will return the byte size of the largest block of memory currently available for allocation.

19.1.5. QUEUE operations

The QUEUE operations will be modified to allow scaning forward and backward through the queue and performing insertions, deletions, and re-arangements of buffers not at the start or end of the queue.

19.1.6. LIST operations

List operations will be expanded to include random access within a list, random repositioning of the list cursor, sub-lists, and type (INTEGER, POINTER, STRING, etc) testing of list elements.

19.2. Features to be implemented in SPL 2.0

19.2.1. Arrays and Subscripting

19.2.1.1. introduction

An array is a collection of related data items of the same type. Each array has a single name, and may have more than one variable in it. The size of an array is the total number of individual variables stored in it.

Picture an array of three integers, called A. It might look like this in memory:

```
234      -98       45
```

You may refer to a specific element (variable) in an array by using a subscript. This is an expression that specifies which element of the array you wish to use.

In the example above, A(1) is a variable that currently has the value 234 stored in it. A(2) holds -98, and A(3) contains 45.

Subscripts are written by entering the array name and then the character (followed by an expression indicating which element you wish to use, and then ending the subscript with the character).

For example, you may write A(NR) which means that the current value of the variable NR should be used to pick out which element of array A that you wish to use. If NR contains the value 2, then A(NR) refers to A(2).

19.2.1.2. Multi-dimensional Arrays

Arrays may also be organized into rows and columns. For example, the array B shown below has 3 rows of four columns each. The size of B is 12, since there are 12 elements in it.

```
  123     67     -8     32767
-1000    300    982       -17
    4      0      0       123
```

To specify which element of such an array you mean, you give the subscript for the desired column, and then the subscript for the desired row, seperated by a semicolon, such as B(X;Y). For example, B(2;3) has the value 0, B(3;2) has the value 982, and B(1;1) has the value 123.

An array such as A is called a single dimension array. B is an example of a two dimension array. In general, the number of subscript values that must be specified to pick out a single element is the number of dimensions an array has.

Arrays may have as many dimensions as you desire. A three dimensional array may be used, and you might wish to think of it as consisting of rows, columns, and pages.

Arrays with more than three dimensions are not often used, perhaps because it is harder for people to conceive of them. You might consider a four dimensional array to be made of rows, columns, pages, and books.

19.2.1.3. Specifications of several elements

In SPL, you may specify more than one element of an array at one time. There are several ways that this can be done. You might list the subscripts of each element you wish to use, specify a range of subscripts that you wish to use, or specify a starting subscript and how many elements that are to be used.

19.2.1.4. Specifying by list

To specify elements by list, you include the subscript of each element in order, seperated by commas, as in A(2,3,1) which indicates that first A(2) is to be used, then A(3), and lastly, A(1).

An example of specifying elements by list for a two dimensional array is: B(2,4,5;7,5) which is the same as:

```
B(2;7)
B(4;7)
B(5;7)
B(2;5)
B(4;5)
B(5;5)
```

19.2.1.5. Specifying by Range

Specifying elements by list is quite a time saver when the elements are to be used in an unusual order. When you wish to use several elements of an array in a row, it is easier to specify them by range. This means that you specify the starting and ending subscripts, and all of the elements in between are used:

C(3:6) means C(3), C(4), C(5), and C(6)

The generalized form of expressing a range is FROM:TO where FROM is the first subscript to be used, and THRU is the last subscript to be used. The range may be specified in reverse:

C(6:3) means C(6), C(5), C(4), and then C(3)

You may also specify the increment. Normally, the value 1 is used. If you specify the BY increment in the form FROM:THRU\BY then the value of BY is added to the value of FROM to get the 2nd and following values. As an example,

C(1:9\2) means C(1), C(3), C(5), C(7), and C(9)

19.2.1.6. Specifying by Count

You may also specify a group of elements by count, that is by using the form FROM#COUNT or FROM#COUNT\BY. This means to use COUNT different elements, starting with element FROM, and incrementing by BY (or by 1 if BY is not specified).

C(5#3) means C(5), C(6), and C(7)

C(5#3\2) means C(5), C(7), and C(9)

19.2.1.7. Specifying All

If all possible values of a subscript are to be used, this is indicated by the character *. If an entire array is to be used, simply use the array name without any subscripts:

If array B has 3 rows and 4 columns, then B(2;*) means
B(2;1), B(2;2), and B(2;3).

B without subscripts means B(1;1), B(2;1), B(3;1), B(4;1),
B(1;2), B(2;2), B(3;2), B(4;2),
B(1;3), B(2;3), B(3;3), B(4;3).

19.2.1.8. Mixed specifications

SPL also allows you to mix these forms, by allowing specification by range or count as an element in specifying by a list. An example would be:

C(3:5,1,7#4,6) which specifies:

C(3), C(4), C(5) then
C(1) followed by
C(7), C(8), C(9), and C(10)
ending with C(6).

SPL allows so many forms of array specification because many program errors are made writing "loops" to express these different groupings of data. Also, by including these forms of grouping in the compiler, the generated code can be optimized, reducing the overhead normally associated with random subscripting.

19.2.2. Automatic Statement Looping

There are two ways that multiple elements of an array may be used. First, each time a statement is executed, then one element in the array list is used. This is the normal form. As an example,

```
Put A(1,4,3)
```

Will print out A(1) the first time the statement is executed, A(4), the 2nd time, and A(3) the 3rd time the statement is executed.

You may also specify an "automatic loop" which will print out all 3 values each time the statement is executed, by using brackets instead of parenthesis:

```
Put A[1,4,3] will print A(1), A(4), and A(3) each time.
```

You can even mix the two formats:

```
A[3#3\2]=B(3,4)+C[2:4]
```

which means that A(3)=B(3)+C(2),
A(5)=B(3)+C(3), and
A(7)=B(3)+C(4)

the first time the statement is executed, but the second time the statement is executed, the meaning is:

A(3)=B(4)+C(2),
A(5)=B(4)+C(3), and
A(7)=B(4)+C(4)

Any statement that has one or more subscript references that use brackets is considered to use automatic looping. An automatic loop continues until some subscript reference finishes. If there are other loops, they will continue on the next execution of a statement. For example:

```
A[3:4]=B[5#4]
```

will perform A(3)=B(5) and A(4)=B(6) on the first execution, which ends the statement because [3:4] completes its list.

The next time the statement is executed, it will perform A(3)=B(7) and A(4)=B(8).

19.2.3. LISTs and Automatic Looping

Automatic looping takes on a slightly different meaning when used in a LIST structure, such as an Input / Output statement. Each loop is completed before going on to the next entry in a list. Any format specification applied to the "loop specification" is applied to every entry in the loop. As an example:

```
Put A[3:6],B,A[7,2]\3
```

means to put out A(3),A(4),A(5),A(6), and B using as much space as needed, but then to put out A(7) and A(2) using only 3 digit places each.

19.2.4. Declaring an Array

To create and use an array in SPL, it must be declared in a type statement, such as REAL, INTEGER, STRING, etc. You must list the array name and the largest value of each subscript. For example,

```
REAL  A(5),B(3;4),C
```

specifies that A is an array having 5 elements, B is a two dimensional array having 3 columns and 4 rows, and that C is also a real variable.

Normally, each array subscript begins with 1 and ends with the value specified. It is possible to specify other starting dimensions, by using range notation:

```
INTEGER A(0:4)
```

specifies an array having 5 elements, numbered 0,1,2,3, and 4. Note that increments are not allowed when specifying an array dimension in a type statement. However, count notation is allowed:

```
STRING  S(3#4)
```

specifies an array of four elements, numbered 3,4,5, and 6.

CORTEX USERS GROUP

S.P.L.

USER GUIDE -2