SPL/M
Reference Manual
# TABLE OF CONTENTS

I. INTRODUCTION ............................................... 1

II. PRIMITIVES .................................................. 2
   Identifiers .................................................. 2

III. DATA REPRESENTATIONS ........................................ 3
   Constants ................................................... 3
   Variables .................................................. 4

IV. EXPRESSIONS AND ASSIGNMENT STATEMENTS .................... 5
   Operator Precedence ........................................ 6
   Assignment Statements ...................................... 7
   Implicit Type Conversions .................................. 8

V. DECLARATIONS .................................................. 9
   Variable Declarations ...................................... 9
   Constant Data Declarations ................................ 10
   Symbolic Constant Declarations ............................ 11

VI. FLOW OF CONTROL & GROUPING ................................ 12
   IF Statement ............................................... 12
   DO-FND Groups .............................................. 13
   DO-WHILE Statement ......................................... 13

VII. PROCEDURES .................................................. 15
   CALL Statement ............................................. 15
   RETURN Statement .......................................... 17

VIII. MISCELLANEOUS FACILITIES .................................. 13
   Direct References to Memory ................................ 13
   Explicit Type Conversions .................................. 19
   GENERATE Statement ........................................ 19

IX. PROGRAM ORGANIZATION AND SCOPE ............................ 21
   Block Structure and Scope .................................. 21
   Program Origins ............................................ 22

X. COMPIL3 AND CONFIGURATION OPTIONS .......................... 24
   System Considerations ...................................... 24
   Compiler Disk .............................................. 24
   Running the Compiler ...................................... 24
   Include Files ............................................... 26
   Printer Considerations ..................................... 27
   Memory Usage ............................................... 28

XI. ERROR HANDLING .............................................. 29
XII. APPENDICES

A. SPL/M Compiler Interface Routines........A.1
B. SPL/M DOS Library Routines..................B.1
C. "Size" Program (SPL/M Source)..............C.1
D. SPL/M Reserved Words..........................D.1
E. Grammar for SPL/M.............................E.1
I. INTRODUCTION

SPL/M (Small Programming Language for Microprocessors) is based on the language PL/M, initially developed by the Intel Corporation.

SPL/M is a block-structured language which features arbitrary length identifiers and structured programming constructs. It is suitable for systems programming on small computers, since the compiler requires only 20K of memory to run. Either two cassette decks or a disk are also required.

The language can be compiled in only one pass, which means that the source code has to be read only once.

Unlike most high-level language translators available for microprocessors, SPL/M is a true compiler: it generates absolute 68000 object code which requires no run-time interpreter. Due to extensive intra-statement optimization, the generated code is almost as efficient as the equivalent assembly language.

The compiler has a number of compile-time options, including a printout that contains the interlisted object code. Syntactical error messages use position indicators to indicate exactly where an error occurs.

This manual has been organized to be usable as both a tutorial and a reference guide. In addition to the many examples in the text, a complete SPL/M program is presented in Appendix C.

As an example of the type of application SPL/M is suited for, this entire manual was formatted using a text processing system written in 800 lines of SPL/M.

II. PRIMITIVES

An SPL/M program consists of primitives (reserved words, identifiers, and constants), along with special characters (operators).

One or more blanks (spaces) are required between any two primitives on the same line, to tell them apart. Blanks are allowed anywhere else, except in the middle of a primitive or a two character operator (such as >=). A carriage return is treated the same as a blank; therefore statements can spill over onto as many lines as necessary.

Comments may be embedded in an SPL/M program anywhere a blank is legal. Comments are delimited by a /* ... */ pair:

/* COMMENTS MAY GO OVER MORE THAN ONE LINE */

Identifiers

An identifier is a programmer assigned name for a variable, procedure, or symbolic constant. Identifier names may be up to 31 characters long.

The first character must be alphabetic (A-Z), while the remaining characters may be either alphanumeric (A-Z, 0-9) or the separation character ($). The latter is completely ignored by the compiler: an identifier with embedded $'s is equivalent to the same identifier with the $'s omitted.

Examples of valid identifiers:

ACIANO      ACIA$NO     (same variable)
BUFFER1     A$RATHER$LONG$PROCEDURE$NAME

Identifier names must not conflict with the reserved words of SPL/M, such as DECLARE, PROCEDURE, etc. A complete list of reserved words for both Versions 1 and 2 of SPL/M is provided in Appendix D.

All identifiers must be declared before they are referenced. Variables and symbolic constants are defined via the DECLARE statement (Section V); procedures are defined via the PROCEDURE statement (Section VII).
III. DATA REPRESENTATIONS

Constants

Constants can be either a number or a character string. As their name implies, their value remains constant during program execution.

A numeric constant, or number, is a string of digits representing an unsigned integer in the range 0-65535. A number is assumed to be decimal unless it is terminated by the letter H, indicating hexadecimal. The first character of a hexadecimal constant must always be numeric (a leading zero is always sufficient).

Examples of numeric constants:

```
0   32   65535
10  20H  0FFFFH
0AH
```

A character constant, or string, consists of one or more ASCII characters enclosed in apostrophes. A null string (i.e. "") is not permitted. Imbedded apostrophes are represented by two consecutive apostrophes (e.g. DON'T).

Constants of one or two characters are equivalent to the numeric constant representing the ASCII code for the character(s). In a two character constant, the left-most character is placed in the most significant byte.

Character constants of more than two characters may only appear in a DATA declaration (Section V).

Examples of character constants:

```
'A'  = 41H
'.'  = 20H
'12' = 3132H
'''' = 27H (one ')
'THIS IS A LONG STRING'
```
Variables

Variables are memory locations set aside by the programmer to hold data that changes during the execution of a program. Variables can be declared as either type BYTE (8 bit data) or type ADDRESS (16 bit data). BYTE variables should be used whenever possible to avoid the overhead associated with double precision arithmetic on the 6800.

Variables are defined using the DECLARE statement (Section V), e.g.

DECLARE CTR BYTE;
DECLARE BUF$PTR ADDRESS;

Vectors (one dimensional arrays) can also be declared, e.g.

DECLARE LIST.(10) BYTE;

which sets aside 10 bytes of storage. A vector has n elements, referenced as

V(0), V(1), ..., V(n-1)

The value in parentheses is the subscript, which can be any SPL/M expression (Section IV). The subscript is added to the base address for BYTE vectors to generate the correct memory reference. For ADDRESS variables, twice the subscript is added to the base to generate the correct memory reference.

For example, if the BYTE vector LIST declared above was located at memory address 400, then LIST(4) would refer to memory address 404. However if LIST was an ADDRESS vector, then LIST(4) would refer to memory addresses 408 and 409.

Subscripted variables can be used anywhere a variable is allowed in SPL/M, except as the operand of the dot operator (Section IV).

The first element of a vector may also be referenced without the subscript; i.e. V and V(0) are the same.
IV. EXPRESSIONS AND ASSIGNMENT STATEMENTS

An expression is simply a way of computing a value. Expressions are formed by combining operators (such as + or *) with either operands (variables or constants) or other expressions enclosed in parentheses.

An arithmetic expression consists of one or more operands which are combined using the following arithmetic operators:

+    addition
-    subtraction (unary minus also allowed)
*    unsigned multiplication
/    unsigned integer division
MOD  modulo (remainder from a division)
.    dot operator (see below)

Examples:

X
ALPHA - BETA
10 MOD 3  (result =1)
-1
X*(Y+Z)/2
.BUF1

The unary dot operator (.) generates a numeric constant equal to the memory address of a variable. The variable cannot have a subscript.

A relational expression consists of two arithmetic expressions combined with one of the following relational operators:

<    less than
<=   less than or equal to
=    equal to
<>   not equal to
>=   greater than or equal to
>    greater than

Comparisons are always performed assuming the operands are unsigned integers. If the specified relation holds, a value of OFFH (true) is returned; otherwise the result is 0 (false).
Examples:

\[ A > 1 \]
\[ \text{CNTR} \leq \text{LIMIT} + \text{OVER} \]
\[ \text{LOOP} < 0 \]

A logical expression consists of either arithmetic or relational expressions combined with one or more of the following logical operators:

- OR
- bitwise OR
- XOR
- bitwise exclusive OR
- AND
- bitwise AND
- NOT
- 1's complement (unary operator)

Examples:

- LADIES AND GENTLEMEN
- NOT FLAGS (same as FLAGS XOR -1)
- \[ X > 1 \text{ OR } Y < 2 \]

The following table summarizes the effect of each logical operator:

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>X OR Y</th>
<th>X XOR Y</th>
<th>X AND Y</th>
<th>NOT X</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Logical expressions are used in assignment statements to perform bit manipulation, and in IF and DO-WHILE statements (Section VI) to specify a series of conditional tests.

Operator Precedence

The order of evaluation of operators in an expression is primarily determined by operator precedence.

Operands are associated with the adjacent operator of highest precedence. Operands adjacent to two operators of equal precedence may be associated with either one. Operators with the highest precedence are evaluated first. Two operators of the same precedence may be evaluated in either order.
The following list summarizes the operator precedence for SPL/M:

highest: ( ) .
unary -
* / MOD
+ -
= < > <= >=
NOT
AND
lowest: OR XOR

Since parentheses have the highest precedence, they can be used to override the implicit order of evaluation. The following fully parenthesized expression

IF (A=3) OR (B > (10*(I+1))) THEN

can also be written:

IF A=3 OR B>10*(I+1) THEN

The parentheses around the I+1, to force the addition to be done first, are the only ones required in this case.

Assignment Statements

Assignment statements perform the real work of a program. They are used to assign the result of an expression to a variable location. The format is:

variable = expression;

The value of the variable on the left-hand side of the equal sign is replaced by the value of the expression on the right-hand side.

Examples:

CTR = CTR + 1;
LIST(I) = 0;
Implicit Type Conversions

Mixed mode is a situation which arises when BYTE and ADDRESS variables or constant are combined in the same expression or assignment statement. To avoid generating unexpected results, SPL/M attempts to use double-precision arithmetic throughout mixed mode expressions.

As soon as an ADDRESS variable or constant is encountered (scanning from left to right), then the remainder of the statement or expression is evaluated in double-precision mode. For example, if X is an ADDRESS variable, then

\[ X = -1; \]

will set \( X = \text{OFFFFH} \) since the unary subtraction will be carried out in double precision.

When operating in double-precision mode, the high-order eight bits of any BYTE variables or constants in an expression are assumed to be 0. In an assignment statement, if the variable on the left-hand side is type BYTE, whereas the expression on the right-hand side is type ADDRESS, then the high-order eight bits of the expression will be lost.

In a complex relational expression involving ADDRESS variables on one side and BYTE variables on the other, the ADDRESS variables should appear first to force the entire expression to be evaluated in double-precision.

Note: the rules used by SPL/M for evaluating mixed-mode expressions are not the same as PL/M.

Functions for performing explicit type conversions are also available in SPL/M; see Section VIII.
V. DECLARATIONS

Variables, constant data arrays, and symbolic constants are defined using the DECLARE statement. (DCL is an allowed abbreviation for DECLARE). All programmer-defined identifiers must be declared before they are referenced in the program. Declarations are subject to "scope", which is explained under program organization (Section IX).

Variable Declarations

The general form of the declare statement is:

```
DECLARE identifier [(bounds)] type;
```

where "(bounds)" is optional and is used only for vector declarations (see below). The "type" may be either BYTE, denoting 8-bit data, or ADDRESS (abbreviated ADDR), denoting 16-bit data.

Examples:

```
DECLARE CTR BYTE;
DCL BUF$PTR ADDRESS;
```

Vectors (one-dimensional arrays) are defined by specifying the number of elements following the variable name; e.g.

```
DCL LIST (10) BYTE;
```

which sets aside 10 bytes of storage, and

```
DCL A$LIST (10) ADDR;
```

which allocates 20 bytes (two for each address element). Vectors are referenced using subscripts as explained in Section III.

The number of elements in a vector declaration may be zero, in which case no storage is reserved. The variable will refer to the same memory location as the next data declaration. For example,

```
DCL BIG$CTR (0) ADDR,
HIGH$CTR BYTE,
LOW$CTR BYTE;
```
HIGH$CTR and LOW$CTR overlay the high and low bytes of BIG$CTR. This example also shows how several variables can be declared in the same statement. Each declaration is separated by a comma.

Sometimes it is desirable to declare a variable at a particular memory location. This is done by preceding the DECLARE statement with an origin, which will cause the next BYTE or ADDRESS variable to be allocated at the given address. Origins consist of a number followed by ':' For example,

36H: DCL ACIA$NO ADDR, NOPRNT BYTE;
3CH: DCL BUF$BEG ADDR;
36H: DCL BUF$END ADDR;

will cause the following allocations to take place:

36H-37H    ACIANO
3AH          NOPRNT
3CH-3DH     BUF$BEG
3FH-3FH    BUF$END

If a declaration is not preceded by an origin, variables are allocated storage immediately following the last declaration. Unless overridden by an explicit origin, the first variable declaration starts at 10H. Declare origins have no effect on DCL DATA and DCL LIT statements (discussed below); however an origin on either will affect the next variable allocation.

Constant Data Declarations

It is often necessary to define constant data, such as character strings or a table. This is done via a DECLARE DATA statement, which has the general form:

DECLARE identifier DATA (constant list);

where "constant list" is a list of numeric or character constants, separated by commas.

It is assumed that data declared in this way will not change during execution of the program. The data is located within the program object code.
The identifier defined in a DCL DATA statement is always of type byte, and is referenced using subscripts the same as any vector.

Examples:

DECLARE REVERSE$DIGITS DATA (9,8,7,6,5,4,3,2,1,0);
DCL MSG DATA ('A MESSAGE STRING',4);

Symbolic Constant Declaration

The DECLARE LITERALLY statement provides a compile-time symbolic constant substitution mechanism similar to the "equate" facility in assemblers. The general form is:

DECLARE identifier LITERALLY 'number';

LITERALLY may be abbreviated as LIT. Whenever the identifier is encountered in the program, it will be replaced by the number.

Examples:

DECLARE CASS1 LITERALLY '0f050h';
DCL TRUE LIT 'OFFH', FALSE LIT '0';
.
.
IF DECK <> CASS1 THEN
DEFAULT = FALSE;
VI. FLOW OF CONTROL & GROUPING

Varicous SPL/M statement types are used to alter the path of program execution. SPL/M does not have the GOTO statement available in BASIC and FORTRAN. However the structured programming constructs (IF-THEN-ELSE, DO-END, and DO-WHILE) can be used to express any program more clearly than if GOTO's were used.

**IF Statement**

The IF statement selects alternate execution paths, based on a conditional test. IF statements have two forms:

a) IF expression THEN statement-1;

b) IF expression
   THEN statement-1;
   ELSE statement-2;

Execution of an IF statement begins by evaluating the expression following the IF. If the right-most (least significant) bit of the result is a 1, then statement-1 is executed. If the bit is a 0, no action is taken for the first form (a), and statement-2 is executed for the second form (b).

Since the result of a relational expression is either OFFH (true) or 0 (false), the construction "IF relational-expr THEN" has the expected result.

In the second form of the IF statement above (b), statement-1 may not be an IF statement. This avoids any ambiguity in the following construction:

IF expression
   THEN IF expression
      THEN statement-1;
      ELSE statement-2;

The rule in this case is that the ELSE belongs to the second (innermost) IF statement. If needed, a DO-END group (defined below) can be used to associate the ELSE with the first IF statement.
IF expression
  THEN DO;
    IF expression THEN statement-1;
  END;
ELSE statement-2;

The ELSE now clearly belongs to the first IF. The following are examples of IF statements:

IF CFLAG THEN CTR = CTR+1;
IF A > 0 AND B > 0
  THEN A=B;
IF X>0 THEN Y=1; ELSE Y=2;

DO-END Groups

The DO-END statement is used to group together a sequence of SPL/M statements, such that they are treated as a single executable statement in the flow of control. For example,

IF SWITCH
  THEN DO;
    TEMP=A;
    A=E;
    B=TEMP;
  END;

All three statements in the DO-END group will be executed if the variable SWITCH is true. Note that indentation is usually used with IF and DO statements to make the logic of the program stand out.

Simple DO-END groups are also used (less frequently) to create a block in which local variables are declared, as described in Section IX.

DO-WHILE Statement

The DO-WHILE statement causes a group of statements to be repeatedly executed as long as a condition is satisfied. The general form is:
DO WHILE expression;
  statement-1;
    
    
  END;

The statements within the DO-WHILE are executed as long as the result of the expression has its right-most bit equal to 1. The expression is evaluated at the beginning of each execution cycle.

This version of SPL/M does not have the PL/M iterative-type DO (like the FOR statement in BASIC). However the more general DO-WHILE can be used in an identical manner:

I = 0;
DO WHILE I < 10;
  CHAR = I+'0';
  CALL PUTCCHR; /* DISPLAY 0-9 */
  I = I+1;
END;

It is sometimes desirable to terminate the execution of a DO-WHILE abnormally (i.e. for some condition other than the expression following the DO). This is facilitated by the BREAK statement, which causes a transfer of control to the first statement following the END which terminates the innermost DO-WHILE.

Example:

I = 0; FOUND = 0;
DO WHILE NOT FOUND;
  IF LIST(I) = KEY
    THEN FOUND = 1; /* EXIT NEXT CYCLE */
    ELSE DO;
      I = I+1;
      IF I >= 100 THEN BREAK; /* ABNORMAL EXIT */
    END;
END;

If the key is found in the list, the DO-WHILE will exit normally with FOUND=1 and I equal to the list index. Otherwise the BREAK will terminate abnormally with FOUND=0.

Note: the BREAK statement is an SPL/M extension and is not in PL/M.
VII. PROCEDURES

Well designed programs make frequent use of subroutines, each of which is related to a particular function. In SPL/M, subroutines are called procedures, and are defined as follows:

```
label: PROCEDURE;
    statement-1;
    :
    :
    statement-n;
END;
```

The "label" is the procedure name, which is required later when the procedure is called. PROCEDURE may be abbreviated PROC.

In this version of SPL/M, all procedures must be defined at the beginning of the program (see Section IX) and nesting of procedure definitions is not allowed.

Since a procedure is a block (also discussed in Section IX), all variables declared within it are "local" and cannot be referenced outside of the procedure. All storage declared in SPL/M is static. Automatic stacking of local variables is not done on entry to a procedure.

All values passed to and from procedures must be done via global variables since procedures cannot have parameters in this version of SPL/M.

**CALL Statement**

Procedures are invoked by the CALL statement:

```
CALL procedure-name;
```

where the procedure must have been previously defined as described above.
Example:

DCL MAX$LINE LITERALLY '30';
DCL LINE (MAX$LINE) BYTE; /* GLOBAL */
...
.
.
.
.

CLEAR$LINE: PROCEDURE;
DCL I BYTE; /* LOCAL */
I=0;
DO WHILE I < MAX$LINE;
  LINE(I) = '';
  I = I+1;
END;
END;
.
.
.
.

CALL CLEAR$LINE;

It is also possible to call a procedure by its address. This makes it easier to link to assembly language subroutines in an operating system. For example,

CALL OFC37H; /* HOME CURSOR */
CALL OFC3DH; /* CLEAR SCREEN */

Note: the construction "CALL number" is an SPL/M extension and is not in PL/M.

The "declare literally" facility (Section V) can be used to define the address as a symbolic constant to keep the reference symbolic:

DCL HOME LIT 'OFC37H';
...
.
.

CALL HOME;
RETURN Statement

When a procedure is called, it starts execution at the beginning of the procedure and normally does not return until the END matching the PROCEDURE statement is reached. However it is possible to force an earlier return by using the RETURN statement, e.g.

IF ERROR THEN RETURN;

Whether a RETURN statement is used or not, a procedure returns to the statement following the original CALL.


VIII. MISCELLANEOUS FACILITIES

Direct References to Memory

It is sometimes desirable to refer to the memory address space of the 6800 directly. (In fact this is the only way I/O can be performed directly in SPL/M, since the language does not have explicit input/output statements. But I/O is usually done via calls on existing operating systems routines.)

When required, direct reference to memory can be done using the MEM and MEMA vectors, which are predeclared to start at address 0. MEM is type byte, while MEMA is type address. The normal doubling of subscripts is not done for MEMA; for example

MEMA(38H) = 0F050H;

sets memory locations 38H and 39H to the hexadecimal value 0F050H.

Note: MEM and MEMA are SPL/M extensions and are not in PL/M.

When used on the left-hand side of an assignment statement, MEM is like the POKE function in some BASIC's. On the right-hand side, MEM is like the PEEK function.

The subscript can be any arithmetic expression, but usually is just an address variable. In the following byte move subroutine, global variables BUF1 and BUF2 contain the start addresses of two buffers, and BSIZE is the number of bytes to move:

BYTE$MOVE: PROC;
   DO WHILE BSIZE <> 0;
      MEM(BUF2) = MEM(BUF1);
      BUF1 = BUF1+1;  BUF2 = BUF2+1;
      BSIZE = BSIZE-1;
   END;
END;
Explicit Type Conversion

Section V discussed implicit (automatic) type conversions in mixed mode expressions. SPL/M also provides two explicit type conversions in the form of built-in functions, which take address expressions as arguments. The functions may appear anywhere an expression is legal.

LOW(expr) returns the least-significant byte of its argument.

HIGH(expr) returns the most-significant byte of its argument.

GENERATE Statement

It is occasionally necessary to link to operating system subroutines which pass values in registers. The GENERATE statement can be used to produce machine code "patches" to accomplish this. It generates code in-line wherever it appears in an SPL/M program. Because of the low-level nature of this statement, and the possibility of making errors, it should be used only where absolutely necessary.

The GENERATE statement has the form:

GENERATE (constant list);

where "constant list" is a list of numeric, character, or symbolic constants, including address (dot) references. GENERATE may be abbreviated GEN.

Note: the GENERATE statement is an SPL/M extension and is not in PL/M.

The following example stores the contents of the accumulator at location 42H after calling a subroutine to input a character:

CALL OFC4AH;
GEN(97H, 42H);

However using only hexadecimal constants makes the code nearly impossible to read. This can be improved by using DCL LIT's and declaring a variable at address 42H:
42H: DCL CHAR BYTE;
DCL GET$CHAR LIT 'OF4CH',
    STAA LIT '97H';
.
.
CALL GET$CHAR;
GEN (STAA, .CHAR);

For additional examples, refer to the SPL/M library routines presented in Appendix B.
IX. PROGRAM ORGANIZATION AND SCOPE

In general, an SPL/M program consists of a set of global declarations, followed by any procedure declarations, followed by the "main" portion of the program. The last line of the program must contain the characters EOF (end of file) which generates an RTS instruction to return to the caller of the main program.

DECLARE statements may appear anywhere in SPL/M, but their location may have different effects due to the "scoping" rules discussed below. In all cases, all names, whether they are variables, procedures, or symbolic constants, must be defined before they are referenced in the program.

Block Structure and Scope

The largest syntactic unit in an SPL/M program is the outermost program block, which consists of the global declarations, procedure definitions, and the "main" program.

Global declarations will be known, or available, to all procedures and the main program. Each procedure may also contain its own declarations, which are local; i.e. known only within that procedure.

Procedures and/or the main program may also have DO-END groups (Section VI) containing additional declarations, which are local to each group.

Example:

DCL A BYTE, B BYTE; /* GLOBAL*/

[XYZ: PROC;
 DCL B ADDR, C ADDR;
 DO;
   DCL A BYTE; A
 END;
 END;

XYZ
/* MAIN */

DCL C BYTE;

... 

EOF

B
A

B
C

A

B

C
The brackets indicate the "scope" of each variable.

Variables, once defined, can be redefined only within a nested block (procedure or DO-END group), which will result in additional static storage being allocated. The new definition is known only within the nested block(s); when the end of the nested block is reached the original definition is in effect again.

Variables, unless redefined, are known within the block in which they are declared and in all blocks nested within it.

Program Origins

Origins, which are simply a number followed by ':', have already been discussed in the context of declare statements (Section V).

A program origin is any origin not preceding a DECLARE statement. Program origins affect the generation of the next byte of object code, including DCL DATA constants (which are located within the program object module).

In this version of SPL/M, program origins are restricted to the following locations:

1) First statement of a program (defines starting address).

2) Beginning of each procedure definition (the origin must be placed just ahead of the procedure name).

3) First statement of "main" (allowed only if the program contains procedure definitions).

In all the cases above, origins are optional. In the absence of any origin the first byte object code will start at location 100H. If the main program or a procedure lacks an origin, the associated code will follow the code immediately preceding.

If provided, the initial (start) origin must be immediately followed by a "null statement" (e.g. 0A100H:;) to distinguish it from a declare origin.

When an origin is specified, the user is responsible for insuring that the resulting code does not overlap code that has already been generated.
The following example summarizes the SPL/M program organization. Everything in brackets [ ] is optional; and any addresses are for example only. Note that declares can go anywhere; however for clarity it is best to restrict them to the beginning of the program, the beginning of each procedure, and the beginning of "main".

```
[ 200H; ] /* OPT. START ADDRESS */
[[ 42H: ] DCL's ] /* GLOBAL DECLARES */
[[ 300H: ] XYZ: PROC; ] /* OPT. PROCEDURE DEFINITIONS */
    
    END;

[ 400H: ] /* OPT. ORIGIN FOR MAIN */
    
    /* main */
    
EOF
```

A jump from the beginning of the program (e.g. 200H) to the beginning of the code for main (e.g. 400H) is automatically generated if there are procedure definitions and if there is either an explicit start address provided or there are any global DCL DATA's.

Refer also to Appendix C for an example of a complete SPL/M program that contains many of the elements described above.
X. COMPILe AND CONFIGURATION OPTIONS
(FLEX Version 1.2)

System Considerations

This version of the compiler is designed to run on a
6800-based system, such as the SWTPc, running under the FLEX
Operating System. In particular, it assumes the existence of:

FLEX 1.0 or 2.0 (not miniFLEX)
20K of user RAM starting at location 0000
SWTBUG monitor ROM or equivalent

Compiler Disk

The disk supplied with the compiler contains the following
files:

SPLM.CMD - SPL/M compiler
FLX102.TXT - Assembler source for compiler interfaces
SPLM.LIB - SPL/M library (general DOS interfaces)
SPLMREAD.LIB - SPL/M library (reading sequential files)
SPLMWRT.LIB - SPL/M library (writing sequential files)
SIZE.TXT - SPL/M source for sample program (SIZE)

The SIZE.TXT source file is intended to be used as a test of
the compiler. It also brings in two of the library files using
the #INCLUDE facility discussed below.

Running the Compiler

The compiler has several compile-time options which control
the generation of listings and binary files.

The general syntax for the SPLM command is:

SPLM[,<source>[,<binary>][,+<option list>]]

The '()' enclose a field defined below and are not actually
typed. The '[]' surround optional fields.

All parameters are optional. If none are provided, then the
compiler runs interactively with the source input coming directly
from the keyboard. This is useful for experimenting, to see what
kind of code the compiler generates for a particular input. In
this mode a full code listing is always output to the terminal. A binary object file is not produced.

The normal mode however is for a <source> file name to be specified to be compiled. In this case the compiler reads the named file from disk until an EOF statement is encountered in the source. The defaults for the <source> file specification are a .TXT extension and the working drive number.

If the optional <binary> file name is also specified, it is used as the name of the object file written to disk. If <binary> is not included in the command, the binary file will have the same 'name' as the source file, but with a .BIN extension.

The option list is prefixed with a plus sign ('+'), with each option represented by a single letter. The letters may be in any order. The following options are available:

B (No binary). Do not create a binary file on disk, even if a <binary> file name is specified.

Y (Yes, delete). Delete an old binary file of the same name as the one about to be produced. If this option is not specified, the compiler will prompt if the binary file already exists. Respond with 'Y' to delete it.

E (Display errors only). The compiler normally produces a line-numbered source listing. If this option is selected only error lines (if any) will be displayed.

C (Display code). Output a full listing, including both the source and the interlisted object code.

G (Display globals symbols). Output a symbol table containing only globally-declared symbols (which includes all procedure entry points).

A (Display all symbols). Output a symbol table with both global and local symbols. Each symbol table block will be displayed as the block is exited.

If a binary file is being produced, it will have a transfer address only if an initial origin (e.g. 0A100H:) is specified as described in Section IX.

If the code option (C) is selected, the object code for each statement is output as it is generated. Since this is a one-pass compiler, occasionally lines like:
are output when the compiler knows that a forward jump is required (for example in an IF or DO-WHILE statement) but doesn’t know the address yet. In such cases an additional entry is output further down in the listing, when the address is resolved. Parentheses are used to indicate that this entry is a "fixup" to a previous unresolved jump:

(155C: 7E 15 90)

A symbol table is output only if one of the options A or G is selected. The symbols are alphabetized on the first character only. Along with each symbol is listed the type (BYTE, ADDR, PROC, or LIT), and its value. Appendix C was printed with the G option.

When the compiler has finished executing, it will display the number of errors, followed by the highest memory address used by the symbol table. If the compiler returns to the monitor without displaying these last two items, a fatal error has occurred (see Section XI).

Examples:

SPLM - Interactive input from keyboard
SPLM,SIZE - Source = SIZE.TXT, binary = SIZE.BIN
SPLM,SIZE,+CY - Source = SIZE.TXT, binary = SIZE.BIN,
display globals, delete old binary
SPLM,SIZE,0.SIZE.CMD,+E - Source = SIZE.TXT, binary = 0.SIZE.CMD,
display errors only

Include Files

The compiler has a built-in include processor, which allows source library files to be brought in during a compile. The syntax is:

#include <source>

where the <source> file name defaults to a .TXT extension and the working drive. The #INCLUDE must start in column 1. The include statement is replaced by the file it includes. When the end of the include file is reached, the compiler switches back to the original file. Included files should not be terminated by an EOF statement, and must not themselves contain #INCLUDE statements (i.e., includes can not be nested).
The source from an included file is normally output to the listing in place of the #INCLUDE statement. However this can be inhibited by the #NOLIST statement:

#NOLIST

source text

#LIST

None of the source text between the #NOLIST and the #LIST will be listed, except for any lines in error. Both statements must start in column 1, and neither are output to the listing.

The library files listed in Appendix B are intended to be included at the beginning of an SPL/M program, as needed. All the files have a #NOLIST statement at the beginning, and a #LIST statement at the end, so they won’t be listed during every compile.

Printer Considerations

To have the listing output to a printer, precede the SPLM command with a P (see the P command in the FLEX User’s Manual). For example,

P,SPLM,SIZE

would cause the line-numbered source listing for SIZE.TXT (along with any error messages) to be output to the printer.

Each page of the listing starts with a form-feed (OCH) character, which is followed by the top margin, title and finally the source/object listing. The title includes the source file name (without extension), date, and page number and is followed by two blank lines. This title is generated in FLX102.TXT and thus can be changed by the user if desired.

The byte at location 3A2H specifies the top margin, i.e. the number of blank lines from the top of the page to the title. This number can be 0, which will cause the title to be printed on the top line.

The byte at location 3A1H specifies the number of lines to be printed on each page before the formfeed is issued. This count includes the top margin (see above), plus three for the title.
To accommodate narrow-width printers, if the byte at location 039DH = 1 the title and source/object listing is limited to 40 columns (assuming the input source is kept less than 32 characters wide).

Note: printer spooling should not be performed during a compile, since the compiler reroutes SWI's back to the ROM monitor to handle fatal errors (see Section XI). The SWI vector is restored when the compiler returns to the DOS.

Memory Usage

The main part of the compiler uses RAM from 0380H to 3FFFH. The symbol table starts at location 400CH and can go up to 47FFH. The highest address actually used by the symbol table is displayed at the end of each compile.

The interface routines which link the compiler with the DOS are assembled to reside at 4800H–4FFFH, but they can be easily moved by changing one ORG statement in FLX102.TXT if more room is needed for the symbol table.

The compiler also uses low memory up to location 0EFH. The top of the stack is set to 1FFH on entry but is restored on exit.
XI. ERROR HANDLING

(SSB/FLEX Version 1.2)

When an error is detected, the source line is printed followed by a line containing one or more single-character flags indicating the error(s). The error codes are:

D - Duplicate declaration of the same identifier
O - Origin error (see Section IX for rules)
P - Procedure definition error (Section VII)
S - Syntax error; statement has an illegal construction
U - Undefined identifier

The flags are positioned under the primitive or operator where the error was discovered. For example, in the printout below,

```
0210  TBL(I) = CTR1 ++ CTR2;
      **** U S U
```

TBL and CTR2 are undefined, and there is a syntax error because of the second ‘+’. When a syntax error is discovered, the remainder of the statement is ignored (up to the next ‘;’), except that undefined identifiers will continue to be flagged. Also, when undefined identifiers are encountered code is still generated (assuming an address of O) to allow patching.

The above errors are the only ones which should occur for most users. They are all non-fatal; that is the compile is allowed to proceed.

In addition there are a number of fatal errors which result in the compiler aborting. They are implemented via software interrupts, and result in the ROM monitor (e.g. SWTEUG) being entered.

If the compiler quits and a register dump is displayed, then a fatal error has occurred. The next to the last field of the dump gives the address of the software interrupt, which should be listed on the next page:
OE73 - expression too complex (operator stack overflow)
OE7F - expression too complex (operand stack overflow)
OE89 - expression too complex (expr type stack overflow)
15AB - program too complex (symbol table nesting >64)
1B94 - input line too long (>80 characters)
26A9 - program too complex (fixup jump for IF or DO-WHILE is longer than 512 bytes)
2712 - bad source format (input doesn't end with ODH)
29EF - program too complex (IF chain nest >60)
29FA - identifier too long (>31 characters)
2F83 - out of symbol table memory (as defined by location 0386H)

If any of the above errors occur, return to the DOS via the warm start address, correct the problem and recompile.

If a fatal error occurs that is not listed above, an internal "impossible" compiler error has occurred. Please send the error code plus a listing of the program causing the error to Programma Consultants, using the attached SER (Suspected Error Report) form.
APPENDIX A

SPL/M Compiler Interface Routines
SPL/M COMPILER - FLEX LINKAGES
(C) COPYRIGHT 1979 BY THOMAS W. CROSLEY

FLEX 1.0/2.0 COMPILER VERSION 1.2

THIS CODE CONTAINS THE DOS-SPECIFIC ROUTINES
NECESSARY TO INTERFACE THE SPL/M COMPILER
WITH A PARTICULAR OPERATING SYSTEM.

******************************************************************************

* EQUATES FOR FLEX DOS
*  
0000  XFC     EQU       0  FUNCTION CODE
0001  XES     EQU       1  ERROR STATUS
0003  XUN     EQU       3  UNIT NUMBER
0004  XFN     EQU       4  FILE NAME
000C  XEX     EQU       12 EXTENSION
003B  XSS     EQU       59 SPACE COMP FLAG
0002  QSO4W   EQU       2  OPEN FOR WRITE
0001  QSO4R   EQU       1  OPEN FOR READ
0004  QSOCL   EQU       4  CLOSE
000C  QDEL    EQU       12 DELETE
0003  FFE     EQU       3  FILE EXISTS
0003  FEOF    EQU       8  END OF FILE
0001  TXTXT   EQU       1  TEXT EXTENSION
0000  BINTXT  EQU       0  BINARY EXTENSION
0160  TRNREC  EQU $16  TRANSFER RECORD
0002  BINREC  EQU       2  BINARY RECORD
0003  FNLEN   EQU       8  FILE NAME LEN
B406  FMS     EQU       $406
B403  PMSCLS  EQU       $403
AD2D  GETFIL  EQU       $AD2D
AD2F  RPTERR  EQU       $AD2F
AD03  WARMS  EQU       $AD03
A090  IB      EQU       $A090  IB Pointer
AC14  LINPTR  EQU       $AC14
AD1B  INBUF   EQU       $AD1B
AC13  CURCHR  EQU       $AC13
AD15  GTCHR   EQU       $AD15
AD13  PUTCHR  EQU       $AD13
AD12  OUTCH2  EQU       $AD12
AD27  NXCH    EQU       $AD27
AD33  STXTXT  EQU       $AD33
AD2A  RSTRO   EQU       $AD2A
AD24  PCLPF   EQU       $AD24
AD39  OUTDEC  EQU       $AD39
AC0E  MONT    EQU       $AC0E
AC0F  DAY     EQU       $AC0F
AC10  YFAR   EQU       $AC10
* EQUATES FOR SWTBUCC
E124
  SFF1 EQU $E124   NON-VECTORED SWI
AO12
  SWIJMP EQU $AO12

* EQUATES TO INTERFACE WITH REST OF COMPILER
0570
  INFOPT EQU $570   INPUT OPTION
0571
  PRTOPT EQU $571   PRINT OPTION
0572
  OUTOPT EQU $572   CORE GENERATION OPTION
0573
  SYMTOPT EQU $573   SYMBOL TABLE OPTION
3D80
  SBFFND EQU $3D80   END OF SOURCE BUF
0C00
  INTORG EQU $CO   INITIAL ORIGIN FLAG
005C
  BUFAFR EQU $3C   CURRENT BUF PTR
0C5E
  BUFFEND EQU $3E   END OF BUFFER PTR

OCCD
  CR EQU $D
0020
  SPACE EQU $20

* VECTOR TABLE FOR COMPILER:

0380
  ORG $380
0380 7E 2C 78
  COLD START ENTRY POINT
     JMP $2C78

0383 7E 48 00
* GETPARMS — JUMP TO USER SUB TO PARSE COMMAND LINE
     JMP GPARMS

0386 47 FF
* HIGH MEMORY — HIGHEST MEM LOC USABLE BY SYMBOL TABLE
     FDB GPARMS-1

0388 00 00
* LOADX — ADDRESS OF USER SUB TO TRANSFER BA TO X
     FDB 0   IF 0, COMPILER WILL GENERATE

038A 7E AD 24
* PCLRF — JUMP TO USER ROUTINE TO OUTPUT CRLF
     JMP PCLRF

038D 7F AD 18
* PUTCRR — JUMP TO USER OUTPUT ROUTINE
     JMP PUTCRR

0390 7F 49 7D
* CASS/DISK READ — JUMP TO USER ROUTINE TO READ SOURCE
     JMP DREAD

0393 7E 4A 65
* CASS/DISK WRITE — JUMP TO USER ROUTINE TO WRITE OBJECT
     JMP DWRITE

0396 00 00
* MULT — ADDRESS OF USER SUB TO MULTIPLY BA BY CONTENTS
     OF BYTES 0,1   RESULT IN BA
     FDB C   IF 0, COMPILER WILL GENERATE

0399 00 00
* DIV — ADDRESS OF USER SUB TO DIVIDE BA BY CONTENTS OF
     BYTES 0,1   QUOTIENT IN BA, REMAINDER IN 0,1
     FDB C   IF 0, COMPILER WILL GENERATE
SPL/IX COMPILER - FLEX LINKAGEFS

6-12-79 TSC ASSEMLER PAGE A.4

039A AO 20
   * LINBUF - ADDRESS OF LINE BUFFER USED BY INBUF
   LINBUF FDB 1B
   *
039C 00
   * FCB C NOT USED
   *
039D 00
   * NARROW - SET TO 1 IF PRINTER HAS 40 COLUMNS
   NARROW FCB 0
   *
039E 7E AD 15
   * GETCHR - JMP TO USER KEYBOARD CHARACTER INPUT ROUTINE
   JMP GETCHR
   *
03A1 39
   * PLEN - NUMBER OF LINES OUTPUT AFTER FORMFEED
   FCB 57
   *
03A2 02
   * TMAR - NUMBER OF BLANK LINES BETWEEN FORMFEED AND TITLE
   FCB 2
   *
03A3 00
   * FCB 0 NOT USED
   *
03A4 7E AD 1B
   * LINEIN - JMP TO USER KEYBOARD LINE INPUT ROUTINE
   JMP INBUF
   *
03A7 7E 4B 1F
   * PTITLE - JMP TO USFR SUB TO OUTPUT TITLE AT TOP
   *
   OF PAGE
   JMP PTITLE
   *
03AA 7F 48 44
   * WRAPUP - JMP TO WRAPUP ROUTINE
   JMP CLOSE
   *
   *
   * NOTE — THE FOLLOWING CODE IS VECTORED TO FROM LOCATIONS
   * 360-3AC, AND CAN BE REASSEMBLED ANYWHERE BY CHANGING THE
   * THE FOLLOWING ORIGIN:

4800 ORG $4900
   *
   *** NOTE: NEXT 2 INSTRUCTIONS FOR SWITHUG ONLY ***

4800 CE E1 24 CPARMS LDX #$5FE1 RESTORE NORMAL SWI'S
4803 FF A0 12 STX SWIJMP
   *
4806 7F 05 70 CLR INPOPT CLEAR OPTION FLAGS
4809 7F 05 71 CLR PTOPT
480C 7F 05 72 CLR OUTOPT
480F 7F 05 73 CLR SYMPT
4812 7F 4B F3 CLR DELOPT
   *
   * PARSE THE COMMAND LINE

4815 B6 AC 18 LDA A CURCHR
4818 B1 0D CMP A #$CR
481A 26 09 BNE CP10
481C BD AD 2A JSR RST310 INTERACTIVE KEYBOARD OPTION
481F BD 4B 9F JSR TTITLE OUTPUT TITLE
4822 7F 48 F4 JMP CP70
SPL/M COMPILER - FLEX LINKAGES 6-12-79 TSC ASSEMBLER PAGE A.5

* SET DEFAULTS FOR DISK INPUT
4825 BE 02  CP10  LDA A #2
4827 B7 05 70  STA A INPOPT  INPUT FROM DISK
482A BE 05 71  STA A PRTOPT  SOURCE PRINTOUT
482D 7C 05 72  INC OUTOPT  PRODUCE BINARY

* CLR INCLP  INCLUDE NEST=0
4830 7F 4B FF  CLR REOF  READ FCF=FALSE
4833 7F 4B FF  CLR PAGENO  PAGE NUMBER=0

* PARSE SOURCE FILE NAME
4839 CE 4C 03  LDX #RFCB
483C BD AD 2D  JSR GETFIL
483F 24 09  BCC CP30  BRANCH IF OK
4841 BD AD 3F  ERROR JSR RPTERR
4844 BD B4 03  CLOSE JSR FMSCLIS  CLOSE ALL FILES
4847 7E 00  JMP WARMIS

* OPEN SOURCE FILE
484A 86 01  LDX #TXTEXT
484C BD AD 33  JSR SETEXT  DEFAULT EXT IS .TXT
484F 86 01  LDA #QSO4R
4851 A7 00  STA A XFC,X
4853 BD B4 06  JSR FMS
4856 26 E9  BNE ERROR

* COPY SOURCE FILE NAME TO BINARY
4858 CF 4C 03  LDX #RFCB
485B FF 4B F4  STX XTMP
485E CE 4D 43  LDX #WFCB
4861 FF 4B F6  STX XTMP2
4864 BD 49 49  JSR COPYFN
4867 CE 4D 43  LDX #WFCB
486A 6F 0C  CLR XEX,X  CLEAR EXTENSION
486C 6F 0D  CLR XEX+1,X
486F 6F 0E  CLR XEX+2,X

* JSR NXTCH
4870 BD AD 27  CMP A #CR
4873 81 0D  BEQ GP70  USE DEFAULTS
4875 27 7D  CMP A #+
4877 81 2B  BEQ OPTLP  GET OPTIONS
4879 27 16

* LDX LINPTR
487B FF AC 14  DEX
487E 09  STX LINPTR  RESET FOR GETFIL

* PARSE BINARY FILE NAME
4882 CE 4D 43  LDX #WFCB
4885 BD AD 2D  JSR GETFIL
4888 25 B7  BCS ERROR
488A BD AD 27  JSR NXTCH
488D 81 2B  CMP A #+

*
* GET OPTIONS (+BYECAC)

BNE CP70 USE DEFAULTS

* ALL DONE

BNE OPT10 DON'T PRODUCE BINARY

BRA OPTLP

DELETE OLD BINARY

BRA OPTLP

PRINT ERRORS ONLY

BRA OPTLP

FULL PRINTOUT WITH CODE

BRA OPTLP

PRINT ALL SYMBOLS

BRA OPTLP

PRINT ONLY GLOBAAL SYMBOLS

BRA OPT45

ILLEGAL OPTION

JMP CLOSEF

ILLOPT

FDB $0DOA

'ILLEGAL OPTION SPECIFIED'

FCB '4'

TST OUTOPT

NO BINARY

LDX #WPCB

LDA #BINEXT

DEFaulT EXT IS .BIN

STA XFC,X

FMS

GP80

LDA #3FF

NO SPACE COMPRESSION

LDA XSC,X
4907 39 RTS ALL DONE WITH COMMAND LINE

4910 A6 01 GP3C LDA A XEF.S.X GET ERROR
4912 B1 03 CMP A #EFEE EXISTS ALREADY?
4914 26 30 BNE ERROR0 SOME OTHER ERROR
4916 7D 4B F3 TST DELOP
4918 26 10 BNE GP90 DELETE OLD BINARY
491A CE 49 61 LDX #$DELMSG
491E BD 4B 6C JSR OUTST2
4921 BD AD 15 JSR GETCHR
4924 1A 59 CMP A #Y
4926 27 03 BEQ GP90
4928 7E 48 44 JMP CLOSEF ADOPT

* DELETE OLD BINARY FILE

492B CF 4D 43 GP3C LDX #$VFCE
492F FF 4B F4 STX XTMP
4931 CE 4E 33 LDX #$FCE
4934 FF 4B F6 STX XTMP2
4937 BD 49 42 JSR COPIFN USE INCL FCB AS TEMP
493A CF 4E 03 LDX #$FCB
493D 86 0C LDA A #$QDEL DELETE DESTROYS FCB
493F A7 00 STA A XFC,X
4941 BD B4 06 JSR FMS
4944 27 B4 BEQ GP75 NOW GO OPEN IT
4946 7E 43 41 ERROR0 JMP ERROR

* COPY FILENAME IN FCB(XTMP) TO (XTMP2)

4949 C6 0C COPIFN LDA B #12
494D FF 4B F4 CPLP LDX XTMP
494F A6 03 LDA A XUN,X
4950 08 INX
49 1 4B F4 STX XTMP
4914 FF 4B F6 LDX XTMP2
4917 A7 03 CPLP1 STA A XUN,X
4919 08 INX
49 A FF 4B F6 STX XTMP2
491D 5A DEE E
49 F 26 FB BNE CPLP
4960 39 RTS

* DFLMSG FDB $ODCA
4963 44 FCC *DELETE OLD BINARY (Y-N)?
497C 04 FCB 4

* READ SOURCE FROM DISK

497D 7D 4B FF DREAD TST EOF
4980 27 05 BEQ DREAD1
4982 CF 4C 03 LDX #$FCE
4985 2C 63 ERA ERROR1 TRYING TO READ PAST EOF

* DREAD1 FSR BEPD READ FIRST BYTE OF SOURCE LINE
4987 8D 26 TST BEQ EOF END OF FILE?
4989 7D 4B FF TST BEQ EOF YES
SPL/A COMPILER – FLEX LINKAGES 6-12-76 TSC ASSEMBLER PAGE A.8

4967 B1 23  CMP A "";
4968 C7 5E   BFFQ INCL  CHECK FOR "#INCLUDE";
4969 D6 0F   BSR RDLINE  READ READER ID OR LINE
496A C6 3D   LDA B #SEFEND/256 CHECK FOR BUFFER OVERFLOW
496B 36 30   LDA A #SEFEND
496C 26 3F   SUB A SEFEND+1
496D D2 3F   SBC B SEFEND
496E 26 01   BNE EN
496F 4D   TST A
4970 22 E6   BH BHI DREAD1 READ ENOUGH FOR ROW
4971 39   RDREF RTS
4972 DE 0F   RDLINE LDX SEFEND
4973 A7 00   STA A 0,X ASSUMES ONE READ BEFORE CALL
4974 B0   INX
4975 BE 0F   STX SEFEND
4976 81 0D   CMP A "CR"
4977 27 04   BEQ RL10
4978 5D 03   BSR RBD5
4979 20 F3   BRA RLO5
497A 3F   RL10 RTS

* READ BYTE FROM DISK

49B2 FF 4B F4   RBFD STX XTMP DEFAULT IS READ FCE
49B3 C0 4C 03   RBFD0 LDX #RFCE
49B4 7D 4B FF   TST INCLP
49B5 27 03   BEQ RBFD1
49B6 CF 4E 33   LDX #IFCE
49B7 04 06   RBFD1 JSR FMS SWITCH TO INCLUDE FCB
49B8 27 1E   BEQ ROK
49B9 A6 01   LDA A XES,X
49BA 31 00   CMP A #EOF
49BB 26 1F   BNE ERROR1
49BC 7D 4B FE   TST INCLP
49BD 27 0F   BEQ EOF
49BE 7D 4B FF   CLR INCLP
49BF 26 04   LDA A #QSC1
49C0 A7 00   STA A XEC,X
49C1 26 0E   JSR FMS CLOSE INCLUDE FILE
49C2 20 D7   BNE ERROR1
49C3 26 01   ERA RBFD0
49C4 BE 4E 06   JSR FMS
49C5 26 0E   BNE ERROR1
49C6 20 D7   ERA RBFD0
49C7 26 01   JSR XEOF
49C8 B7 4B FF   STA A REOF
49C9 4D   ROK TST A
49CA 27 DA   BEQ RBFD1
49CB FF 4B F4   LDX XTMP
49CC 20 53   RTS
49CD 7E 4B 41 ERROR1 JMP ERROR

* IGNORE NULL CHAR

49DE CD C3   INCL PRR RBFD
49E0 C1 49   CMP A ""
49E1 27 0B   BEQ INCLO5
49E2 DE 3F   LDX SEFEND
49E3 26 23   LDA B ""
40F7  87 00  STA  B  C,X
40F8  CC  INX
40F9  FF 37  STX  BUFEND
40FA  2C 94  BRA  DREAD2
40FB  7F 4B  FF  INCL05  TST  INCLP
40FC  8D AD  INCL10  BNE  INCE  ERROR - NESTED INCLUDE
40FD  81 0D  CMP  A  "/CR"
40FE  27 42  BEQ  INCE
4100  81 20  CMP  A  "/SPACE"  ERROR - NO FILENAME
4101  26 F6  BNE  INCL10
4102  8D A3  BSR  REFD
4103  81 0D  CMP  A  "/CR"
4104  27 32  BEQ  INCE
4105  FF 03 9A  LDX  LINDUF
4106  FF AC 14  STX  LINPTR
4107  A7 00  INCL20  STA  A  C,X  COPY FILE SPFC INTO INPUT LUFFTR
4108  08  INX
4109  81 0D  CMP  A  "/CR"
410A  27 04  BEQ  INCL30
410B  6D 90  BSR  REFD
410C  20 F5  BRA  INCL20
410D  CF 4E 33  INCL30  LDX  "/ICBF"
410E  BD AD 2D  JSR  GETFIL  PARSE INCLUDE FILE NAME
410F  25 14  EBCS  INCO
4110  86 01  LDA  A  "/TXTTEXT"
4111  BD AD 33  JSR  SETEXT  DEFAULT EXT IS .TXT
4112  86 01  LDA  A  "/QSO42"  OPEN INCLUDE FILE
4113  A7 00  STA  A  XFC,X
4114  BD B4 06  JSR  IMS
4115  26 06  BNE  INCO
4116  7C 4B FF  INC  INCLP
4117  7F 49 77  JMP  DRREAD1
4118  CE 4A 54  INCO  LDX  "/INCMSG"
4119  BD 4B 5C  JSR  OUTST2
411A  CE 4E 23  LDX  "/IFCB"
411B  20 9F  BRA  ERROR1
411C  CE 4A 54  INCF  LDX  "/INCMSG"
411D  BD 4B 6C  JSR  OUTST2
411E  7F 48 44  JMP  CLOSE
411F  OD 0A  INCMSG  FDB  SODA
4120  23  FCC  "/INCLUDE ERROR"  "WRITE OBJECT BUFFER TO DISK"
4121  04  FCB  4
4122  
4123  DE 3C  DWRITE  LDX  BUFADR  POINTS TO OBJ IUF
4124  A6 00  LDA  A  C,X  GET RECORD TYPE
4125  26 04  BNE  W03
4126  7F 4B FB  CLR  ISTRR  START RECORD INITIALIZATION
4127  35  W01  RTS
4128  81 FF  W03  CMP  A  W0FF
4129  26 15  BNE  W10
412A  9C  LDA  A  INTORG  END RECORD
412B  27 F7  BEQ  W01
LDA A STRNUMG
JMP WBTD

4A88 21 01  W10
CMP A #1

REGULAR OBJ RECORD (MAX 512 BYTES)

SAVE PTR TO RFC OF CODE

BA HAS LENGTH - 1

IF >128 BYTES, SPLIT UP

DUMMY JUMP ONLY?

DON'T OUTPUT JUST 7F 0000

BINARY BLOCK

REMEMBER INITIAL STRT ADDR
WRITE STRT ADDR

NORMALIZE LENGTH
WRITE LENGTH
WRITE OUT CODE
4AFF 08  INX
4AFF 7A 4B FA  DEC  COUNT
4AFF 26 F6  BNE  4LOOP
4AFF 83 E9  STX  CODE  SAVE PTS TO NEXT BYTE
4AFF 39 WRTS  RTS

* 4AFF 86 7F  WSEC  LDA A  #37F  WRITE A SECTION (122 BYTES)
4AFF 8D BD  BSR  #WBLK
4AFF D5 3C  LDX  #BUFADR
4AFF E6 01  LDA B  1,X
4AFF E6 02  LDA A  2,X
4AFF 2B 30  ADD A  #30  ADD 122 TO START ADDR
4AFF C0 00  ADC B  #0
4AFF E7 01  STA B  1,X
4AFF E7 02  STA A  2,X
4AFF 2C 95  BRA  #15

* WRITE BYTE TO DISK
4BCD FF 4B F4  WBTB  STX  #XTMP
4B10 CF 4D 43  LDX  #WFGB
4B13 BD 4B 06  JSR  #FNS
4B16 26 04  BNE  ERROR2
4B13 FF 4B F4  LDX  #XTMP
4B1D 3C  RTS
4B1C 7F 43 41  ERROR2  JMP  ERROR

* OUTPUT TITLE AT TOP OF PAGE
4B1F CF 4C 03  PTITLE  LDX  #RFGB
4B22 CF 03  LDA B  #FNLN  LENGTH OF FILE NAME
4B24 AE 04  PTTL05  LDA A  #XFN,X  GET CHAR OF IN
4B26 26 02  BNE  PTTL10
4B23 86 20  LDA A  #SPACE  PAD
4B2A 8D AD 12  PTTL10  JSR  #PUTCRR
4B2D 02  INX
4B2F 5A  DEC B
4B2F 26 F3  BNE  PTTL05

* 4B31 CF 4B BB  LDX  #TITLE0
4B34 BD 4B 5F  JSR  #OUTSTR
4B37 E6 03 9D  LDA A  #NARROW  40 CHAR PRINTOUT?
4B3A 27 03  BEQ  PTTL12  NO
4B3C CF 4B C0  LDX  #TITLE2
4B3F BD 4E 5F  JSR  #OUTSTR
4B42 20 06  BRA  PTTL15
4B44 CF 4B C5  PTTL12  LDX  #TITLE3  OUTPUT COMPILER VERSION
4B47 BD 4B 5F  JSR  #OUTSTR
4B4A BE 4B 02  PTTL15  JSR  DATE  OUTPUT DATE
4B4D CF 4B 7A  LDX  #PAGE
4B57 BE 4B 5F  JSR  #OUTSTR
4B5A 7C 4C 00  INC  #PAGENO
4B5E 66 4C 00  LDA A  #PAGENO
4B5F BD 4B 73  JSR  #ONEDEC  OUTPUT PAGE NUMBER
4B C7F AD 24  JMP  #PCRLT
* SAME AS PSTRMC EXCEPT NO INITIAL CRLF

4B6F A6 00 OUTSTR  LDA A  C,X
4B61 31 04 CMP A  04
4B67 27 06 BEQ OSRTS
4B65 BE AD 10 JSR PUTCJR
4B63 0C INX
4B69 2C F4 BRA OUTSTR
4B6B 3C OSRTS RTS

* SAME AS OUTSTR EXCEPT USES OUTCH2

4B6C A6 00 OUTST2  LDA A  C,X
4B6F 21 04 CMP A  04
4B70 27 F9 BEQ OSRTS
4B72 BE AD 12 JSR OUTCH2
4B75 0C INX
4B76 2C F4 BRA OUTST2

* OUTPUT ONE BYTE IN DECIMAL

4B73 B7 4C 02 ONEDEC  STA A  DCT+1
4B7E CF 4C 01 LDX  0DOT
4B7F 5F CLR B
4B7F 7F AD 29 JMP OUTDEC

* OUTPUT DATE

4B82 BE AC 0E DATTR  LDA A  MONTH
4B85 BE 4B 78 JSR ONEDEC
4B87 BE 2D  LDA A  "-"
4B8A DD AD 13 JSR PUTCJR
4B8D BE AC 0F LDA A  DAY
4B8E BE 4B 78 JSR ONEDFC
4B92 BE 2D  LDA A  "-"
4B95 DD AD 13 JSR PUTCJR
4B98 BE AC 10 LDA A  YEAR
4B9F 7F 4B 78 JMP ONEDFC

* TITLE FOR INTERACTIVE USE

4B97 BE AD 24 ITITLE  JSR PCRLF
4BA1 BC 03 9D  LDA A  "NARROW"
4BA4 2C 0C  BNE ITITLE10
4BA6 CE 4B EE  LDX  #"TITLE0"
4BA9 BE 4B 5F JSR OUTSTR
4BAC CE 4B BC  LDX  #"TITLE1"
4BAF BE 4B 5F JSR OUTSTR
4BB2 CE 4B 05 ITITLE10  LDX  #"TITLE3"
4BB5 BE 4B 5F JSR OUTSTR
4BB8 7E AD 24 JMP PCRLF

* TITLE0  FCC  ""
4BCE 2C TITLE1  FCC  ""
4BCC 2C TITLE2  FCC  ""
4BC4 2C TITLE3  FCC  "SPL/X COMPILER VERSION 1.2"
4BFC 04 FCB  4
4BFE 2C TITLE3  FCC  "PAGE"
4BFA 2C PAGE  FCC  "PAGE"
<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>4BF2 C4</td>
<td></td>
<td>FCB</td>
<td>4</td>
</tr>
<tr>
<td>4BF3 CC</td>
<td></td>
<td>DFLOPT</td>
<td>FCB C</td>
</tr>
<tr>
<td>4BF4 CC CO</td>
<td></td>
<td>Xtmb</td>
<td>FDB C</td>
</tr>
<tr>
<td>4BF6 CC CO</td>
<td></td>
<td>Xtmb2</td>
<td>FDB C</td>
</tr>
<tr>
<td>4BF8 CC CO</td>
<td></td>
<td>C0DF</td>
<td>FDB O</td>
</tr>
<tr>
<td>4BFA CC</td>
<td></td>
<td>COUNT</td>
<td>FCB C</td>
</tr>
<tr>
<td>4BFB CC</td>
<td></td>
<td>ISTRT</td>
<td>FCB C</td>
</tr>
<tr>
<td>4BFC CC CO</td>
<td></td>
<td>STRT</td>
<td>FDB C</td>
</tr>
<tr>
<td>4BFF CC</td>
<td></td>
<td>INCLP</td>
<td>PCB O</td>
</tr>
<tr>
<td>4BF0 CC</td>
<td></td>
<td>REOF</td>
<td>PCB C</td>
</tr>
<tr>
<td>4C00 CC</td>
<td></td>
<td>PAGENO</td>
<td>PCB C</td>
</tr>
<tr>
<td>4C01 CC CO</td>
<td></td>
<td>D0T</td>
<td>FDB O</td>
</tr>
<tr>
<td>4C03</td>
<td></td>
<td>RFCB</td>
<td>RMB 220</td>
</tr>
<tr>
<td>4D43</td>
<td></td>
<td>VFCB</td>
<td>RMB 220</td>
</tr>
<tr>
<td>4E83</td>
<td></td>
<td>IFCE</td>
<td>RMB 220</td>
</tr>
<tr>
<td>4FC3</td>
<td></td>
<td>PCFHd</td>
<td>FQU</td>
</tr>
</tbody>
</table>

No error(s) detected.
SYMBOL TABLE:

```
BH  409F  BEXT  COCO  BEXTBC  CO02  BUFADR  CO3C  BUFDND  CO3E
CLOSE  4844  CCDE  4BF3  COPYFN  4549  CCOUNT  4BEA  CCLP  424B
CPLP1  4957  CCR  COOD  CURCHR  AC13  DATE  4EB2  DAY  AC0F
DELMS  4961  DELOPT  4BF3  DCT  4C01  DRFAD  457D  DREAD1  4937
DRTAD2  4992  DMTRFT  4A65  DFRG  CO00  DFE  CO03  DRRC  4341
ERROR0  4946  ERROR1  49FA  ERROR2  4B1C  EFS  D406  ESECIS  E403
EXEC  CO08  EFSTR  AD15  EFSTRIL  AD2D  EPC10  4325  EP3C  434A
CP70  43F4  CP75  42FA  CP20  4316  GP90  482E  GP84S  4300
IB  A030  IYCE  4B23  ILOOPT  43B9  INBUFF  AD1B  INCH  4A4B
INCL  49ED  INCL05  49FF  INCL10  4A03  INCL20  4A19  INCL30  4A24
INCLP  49E9  INCMSC  4A54  INCO  4A4C  INPOPT  0570  INCOR  CO00
ISTR  48F4  ITTLFL  4B9E  ITTLF1C  4B2E  LBEUFL  039A  LIMITR  AC14
ISO  4A0F  ISNARROW  032D  ISNCH  AD27  LIMITQ  457B  OPT10  49A1
OPT20  43AA  OPT25  43E0  OPT20  4B5B  OPT40  43ED  OPT45  45C3
OPT30  43C3  OPT30  4810  OPT30  4B9D  OPT30  4B6D  OUTCH2  AD12
OUTDEC  AD32  OUTDECT  0572  OUTSTR2  4B6C  OUTSTR  4B2F  PAGE  4BEA
PAGENO  4C00  PAGLFL  AD24  PGEND  4FC3  PRTOPT  0571  PTITLE  4B1F
PTITC0  4B24  PTITC10  4B2A  PTITLE2  4B44  PTITLE15  4B4A  PTITCH  AD1C
QDRL  400C  QSCL  CO04  QSO4R  CO01  QSO4W  CO02  REPED  49B2
RBBBB  49B5  RBED1  49C0  RDLINE  4A22  REON  4A1A  REON  4BEF
RCB  4C03  RLO5  49A4  RL10  4B10  RCK  4B25  RTPWR  AD3F
RSTRT  AD2A  SRSTID  5D60  SRTF  49DF  RSTEXT  4B33  SFE1  F124
SPAC  CO20  STRT  4BFC  SWSJMP  AD0C  SYNPR  0573  TITL30  4EBB
TITL1  4BBC  TITL2  4BCC  TITL13  4BCC  TNRRC  CO16  TXTEXT  CO01
W01  446C  WC3  4A6F  W10  4A82  W15  4482  W20  4ACD
W50  4AD9  WAMS  ADC3  WBLK  4ABA  WEVD  4BCD  WEVD  4BD3
WLOCP  4AEB  WRLS  4F83  WSLC  4AF9  XFS  CO01  XEX  CO0C
XFC  CO03  XFR  CO04  XSC  CO3B  XTHIP  4FB4  XTHIP2  4BF6
XUN  CO03  XYPAR  AC10
```
APPENDIX B

SPL/M DOS Library Routines
/* NOLIST
   */
/*
   SPIN LIBRARY 'SPIN.LIB' ----
   DOS INTERFACE Routines

   FLEX VERSION 1.0 6-2-79 */
/

/*
   THESE ROUTINES CAN BE USED BY AN
   SPIN PROGRAM TO INTERFACE WITH
   THE DOS. PARAMETERS NORMALLY
   PASSED IN REGISTERS ARE PLACED
   IN GLOBAL VARIABLES INSTEAD.

   SEE THE FLEX 2.0 "ADVANCED PRO-
   GRAMMERS GUIDE" FOR A DETAILED
   DESCRIPTION OF EACH OF THE
   ROUTINES.

   THE VERSION NUMBER OF THE PROGRAM
   MUST BE DECLARED AS A SYMBOLIC
   CONSTANT BEFORE INCLUDING THIS
   FILE. THE STARTING ADDRESS AND ANY
   GLOBAL VARIABLES NOT ON PAGE 0 (SUCH
   AS ARRAYS) SHOULD ALSO BE DECLARED
   BEFORE THE LIBRARY INCLUDES, E.G.

   OA1COH:
   DCL VERSION LIT '1';

   OA24CH: DCL RFCE (320) BYTF;
   #INCLUDE SPIN.LIB
   #INCLUDE SPINREAD.LIB

   VARIABLES DECLARED AFTER THE INCLUDES
   WILL BE PLACED ON PAGE 0 UNLESS
   PRECEDED BY AN ORIGIN. */
/

/* GENERATE VERSION NUMBER */
GEN('*/BRA 1*/2CC1H/VERSION;');
/
/* OVERLAY FOR PART OF DOS MEMORY MAP */
   ACOOH: DCL LINFU (128) BYTF;
OA0CH: DCL FOLCHR BYTE;
OA0EH: DCL SMONTH BYTE, SDAY BYTE, SYEAR BYTE;
OA11H: DCL LASTTFRN BYTE;
OA14H: DCL LINFTR ADDR;
OA1CH: DCL CURCHR BYTE, PREVCHR BYTE;

DCL TRUE LIT 'OFF';
DCL FALSE LIT '0';
DCL CRFL LIT 'CDOAH';
/
/* SYMBO LIC CONSTANTS FOR DISK IO */
DCL XFC LIT 'O'; /* FCB OVERLAY */
DCL XFS LIT '1';
DCL XUN LIT '3'
DCL XFN LIT '4'
DCL XFX LIT '12'
DCL XFS LIT '15'
DCL XHC LIT '59'
DCL QSRW LIT '0' /* FUNCTION DEFS */
DCL QSO4R LIT '1'
DCL QSO4W LIT '2'
DCL QSO4U LIT '3'
DCL QSCLS LIT '4'
DCL QSRW LIT '5'
DCL ERRW LIT '3' /* ERROR STATUS */
DCL DXRN LIT '0' /* DEFAULT EXTENSIONS */
DCL DXTXT LIT '1'
DCL DXCLD LIT '2'
DCL DXSYS LIT '4'
DCL DEXAK LIT '5'
DCL DXOUT LIT '11'

WARMS:PROC;
 GEN(/**JMP*/7EH, OAD03H);
END;

1OH:DCL CHAR BYTE;
 /* READ ONE BYTE INTO CHAR */
GETCHR:PROC;
 CALL /**GETCHR*/OAD15H;
 GEN(/**STA*/097H, CHAR);
END;
 /* WRITE ONE BYTE FROM CHAR */
PUTCHR:PROC;
 GEN(/**LDA*/096H, CHAR);
 CALL /**PUTCHR*/OAD16H;
END;
 /* OUTPUT A SPACE */
SPACE:PROC;
 GEN(/**LDA*/036H, ' ');
 CALL /**PUTCHR*/OAD16H;
END;

DCL INBUFF LIT 'OAD1EH';
DCL MSGA ADDR;
 /* OUTPUT STRING WHOSE ADDRESS IS IN MSGA */
PSTRNC:PROC;
 GEN(/**LDX*/CDEH, MSGA);
 CALL /**PSTRNC*/OAD1FH;
END;

DCL ERROR BYTE;
 /* CLASSIFY CHAR; ERROR = TRUE IF NOT ALPHANUMERIC */
CLASS:PROC;
 ERROR = OFFH;
GEN(/*LDA*/_/96H, CHAR);
CALL /*CLASS*/_/OAD21H;
GEN(/*BCC*/_/24H, 1); RETURN;
ERROR = 0;
END;

DCL PCRLF LIT 'OAD24H'; /* GET NEXT BUFFER CHARACTER INTO CHAR */
NXTCH:PROC;
CALL /*NXTCH*/_/OAD27H;
GEN(/*STAA*/_/97H, CHAR);
END;

DCL RSTPIO LIT 'OAD2AH';

DCL FCBA ADDR;
/* GET TILE SPEC INTO FCB WHOSE ADDRESS IS IN FCBA. NORMALLY
ONLY CALLED BY LIBRARY ROUTINES RDOPEN AND WTOPEN */
GETFILE:PROC;
ERROR = OFH;
GEN(/*LDX*/_/ODEH, FCBA);
CALL /*GETFILE*/_/OAD2DH;
GEN(/*BCC*/_/24H, 1); RETURN;
ERROR = 0;
END;

DCL LOAD LIT 'OAD30H';
DCL DEFFXT BYTE;
/* SET DEFAULT EXTENSION CONTAINED IN DEFFXT */
STEXT:PROC;
GEN(/*LDA*/_/96H, DEFFXT);
GEN(/*LDX*/_/CDEH, FCBA);
CALL /*STEXT*/_/OAD33H;
END;

DCL DGETA ADDR, LDSPC BYTE;
/* OUTPUT DECIMAL NUMBER WHOSE ADDRESS IS IN DGETA. LEADING
SPACES WILL BE PRINTED IF LDSPC = TRUE */
OUTDEC:PROC;
GEN(/*LDAB*/_/06H, LDSPC);
GEN(/*LDX*/_/CDEH, DGETA);
CALL /*OUTDEC*/_/OAD39H;
END;
/* OUTPUT HEX BYTE WHOSE ADDRESS IS IN DGETA */
OUTHFX:PROC;
GEN(/*LDX*/_/CDEH, DGETA);
CALL /*OUTHFX*/_/OAD3CH;
END;

/* REPORT DCS ERRORS. NORMALLY
ONLY CALLED FROM DISK I/O
LIBRARY ROUTINES */
RPTERR:PROC;
  GEN(/*LDX*/ODEH,.FCBA);
  CALL /*RPTERR*/OAD3FH;
END;

DCL NUH ADDR, ANYDCTS BYTE;
/* GET HEX NUMBER INTO NUH.
   ERROR SET TRUE IF NOT HEX.
   DCTS SET <> C IF ANY DIGITS
   FOUND. */
GETHEX:PROC;
  NUH=0; ERROR=05H; ANYDCTS=0;
  CALL /*CSTHEX*/OAD42H;
  GEN(/*BCC*/24H,1); RETURN;
  ERROR=0;
  GEN(/*STX*/ODEH,.NUH);
  GEN(/*STAB*/OD7H,.ANYDCTS);
END;
/* OUTPUT 2 HEX BYTES WHOSE
   ADDRESS IS IN DCQA */
OUTADR:PROC;
  GEN(/*LDX*/ODEH,.DCQA);
  CALL /*OUTADR*/OAD45H;
END;
/* INPUT DECIMAL NUMBER INTO NUH.
   ERROR SET IF INVALID NUMBER.
   DCTS SET <> C IF ANY DIGITS
   FOUND. */
INDEC:PROC;
  NUH=0; ERROR=0FFH; ANYDCTS=0;
  CALL /*INDEC*/OAD45H;
  GEN(/*BCC*/24H,1); RETURN;
  ERROR=0;
  GEN(/*STX*/ODEH,.NUH);
  GEN(/*STAB*/OD7H,.ANYDCTS);
END;

DOCMD:PROC;
  CALL /*DOCMD*/OAD4FH;
  GEN(/*STAB*/OD7H,.ERROR);
END;

FMS:PROC;
/* SET ERROR = CFFH WITHOUT
   DESTROYING CHAR IN ACCA */
  ERROR = 0; ERROR = ERROR-1;
  GEN(/*LDX*/ODEH,.FCBA);
  CALL /*FMS*/OBA6H;
  GEN(/*BEC*/27H,1); RETURN;
  ERROR = 0; /* ACCA STILL HAS CHAR */
END;

DCL FMSCLS LIT 'OB4C3H';
"LIST
/* NOLIST */
/* SPLX LIBRARY 'SPLXREAD.LIB' — 
   READ ROUTINES
   
   FLEX VERSION 1.0 6-2-79 */
/* THESE ROUTINES CAN BE USED BY ANY 
   SPLX PROGRAM TO READ A SEQUENTIAL 
   FILE. A FILE CONTROL BLOCK NAMED 
   'RFCB' MUST BE DECLARED BEFORE 
   THE LIBRARY INCLUDE, E.G.: 
   
   OAS4CH: DCL RFCB (320) BYTF; 
   #INCLUDE SPLX.LIB 
   #INCLUDE SPLXREAD.LIB */
/* RDCLOSE — CLOSE A FILE PREVIOUSLY 
   OPENED FOR READING */

RDCLOSE:PROC;
   RFCB(XFC) = QSCLS;
   FCBA = .RFCB;
   CALL FKS;
   IF ERROR THEN DO;
      CALL RPTERR;
      CALL WARKS;
   END;
END;

/* RDERR — HANDLE FATAL READ ERRORS */

RDERR:PROC;
   FCBA = .RFCB;
   CALL RPTERR;
   CALL RDCLOSE;
   CALL WARKS;
END;

/* RDOPEN — OPEN A FILE FOR READING. 
   ON ENTRY, (GLOBAL) DEFEXT MUST 
   CONTAIN THE DEFAULT EXTENSION 
   TYPE - SEE 'SPLX.LIB' FOR 
   SYMBOIC CONSTANTS TO USE. 
   SPACE COMPRESSION IS ALWAYS 
   INHIBITED BY DEFAULT */

RDOPEN:PROC;
   FCBA = .RFCB;
   CALL GETFIL;
   IF ERROR THEN DO;
      CALL RPTERR;
      CALL WARKS;
   END;
RFCE(XFC) = QSO4P;
CALL SETTXT; /* DEFFXT MUST BE SET UP */
CALL ENS;
IF ERROR THEN DO;
   CALL RPTERR;
   CALL WARRS;
END;
/* INHIBIT SPACE COMP */
RFCE(XNC) = TRUE;
END;

/* RBFT - READ ONE BYTE FROM DISK INTO (GLOBAL) CHAR.
ON EXIT, REOF = TRUE IF END OF FILE, ELSE REOF = FALSE */

DCL REOF BYTE;

RBFD:PROC;
REOF = TRUE;
RFCE(XFC) = QSRV;
FCBA = .RFCE;
CALL ENS;
GEN(/*STAA*/97H..CHAR);
IF ERROR THEN DO;
   IF RFCE(XES) = REOF THEN RETURN;
   ELSE CALL RPTERR;
END;
REOF = FALSE;
END;

/* RBFD - READ ONE BYTE FROM DISK INTO (GLOBAL) CHAR. END OF FILE HANDLE AS FATAL ERROR */

RBFD2:PROC;
   CALL RBFD;
   IF REOF THEN CALL RPTERR;
END;
/*LIST
"NOLIST
/* SPLM LIBRARY 'SPLMWRT.LIB' —
   WRITF ROUTINES

   FLEX VERSION 1.0 6-9-79 */

/* THESE ROUTINES CAN BE USED BY AN
   SPLM PROGRAM TO WRITE A SEQUENTIAL
   FILE. A FILE CONTROL BLOCK NAMED
   'WFCB' MUST BE DECLARED BEFORE
   THE LIBRARY INCLUDES, E.G.:

   10CH: DCL RCFC (320) BYTF;
        DCL WFCB (320) BYTF;
    #INCLUDF SPLM.LIB
    #INCLUDF SPLMREAD.LIB
    #INCLUDF SPLMWRT.LIB */

/* WTCLOSE — CLOSE A FILE PREVIOUSLY
   OPENED FOR WRITING */

WTCLOSE:PROC;
   WFCB(XFC) = QSCLS;
   FCBA = .WFCB;
   CALL FSC;
   IF ERROR THEN DO;
     CALL RPTERR;
     CALL WARMS;
   END;
END;

/* WTRR — HANDLE FATAL READ ERRORS */

WTRR:PROC;
   FCBA = .WFCB;
   CALL RPTERR;
   CALL WTCLOSE;
   CALL WARMS;
END;

/* WTOPEN — OPEN A FILE FOR WRITING.
   ON ENTRY, (GLOBAL) DEFEXT MUST
   CONTAIN THE DEFAULT EXTENSION
   TYPE — SET 'SPLM.LIB' FOR
   SYMBOLIC CONSTANTS TO USF.
   SPACE COMPRESSION IS ALWAYS
   INHIBITED BY DEFAULT */

WTOPEN:PROC;
   FCBA = .WFCB;
   CALL CTFFIL;
   IF ERROR THEN DO;
     CALL RPTERR;
CALL WARNS;
END;
WFCE(XFC) = QSO4W;
CALL SETEXT; /* DEFFEXT MUST BE SET UP */
CALL FVS;
IF ERROR THEN DO;
  CALL RRETERR;
  CALL WARNS;
END;
/* INHIBIT SPACE COMP */
WFCE(XNC) = TRUE;

END;

/* WBTD - WRITE ONE BYTE FROM (GLOBAL) CHAR TO DISK. */
WBTD:PROC;
  WFCE(XFC) = QSRW;
  FCBA = .WFCB;
  GFN(*LDA4*/96H,.CHAR);
  CALL FVS;
  IF ERROR THEN CALL WTER;
END;
#LIST
APPENDIX C

"Size" Program (SPL/¥ Source)
SIZE

/* SIZE — DISPLAYS SECTOR COUNT, */
/* LENGTH IN DECIMAL AND HEX, */
/* NUMBER OF LINES (CR’S), PLUS */
/* CHKCSUM AND CREATION DATE OF */
/* A FILE. */

FLEX VERSION 1.0
6-11-79

0:10 OA100H:
0:11 DCL VERSION LIT '1';
0:12
0:13 OA340H: DCL RFCB (320) BYTE;
0:14
0:15 /* "INCLUDE SPLM.LIB  — LIBRARIES INCLUDED HERE */
0:16 /* "INCLUDE SPLMPFAD.LIB */
0:322 DATE:PROC; /* OUTPUT DATE AS MM-DD-YY */
0:327 DCL MONTH LIT '25', DAY LIT '26', YEAR LIT '27';
0:328 DCL DOT ADDR;
0:329 DDSFC = FALSE;
0:330 IF RFCB(MONTH) < 10 THEN CALL SPACE;
0:331 DCTA = .DOT;
0:332 DCT = RFCB(MONTH); CALL OUTDEC;
0:333 CHAR = ['-']; CALL PUTCHR;
0:334 DCT = RFCB(DAY); CALL OUTDEC;
0:335 CHAR = ['-']; CALL PUTCHR;
0:336 DCT = RFCB(YEAR); CALL OUTDEC;
0:337 IF RFCB(DAY) < 10 THEN CALL SPACE;
0:338 CALL SPACE;
0:339 EWD;
0:337
0:338 ASIZE:PROC; /* OUTPUT SIZE AND CHECKSUM INFO FOR A FILE */
0:339 DCL BYTESCTR ADDR, LINESCTR ADDR, CHKSUM BYTE;
0:340 DCL TYESCTR ADDR, FLAG BYTE;
0:341 DCL XSIZE LIT '21'; /* LOC OF SECTOR SIZE IN FCB */
0:342 DCL CR LIT '0EH';
0:43
0:343 BYTESCTR = 0; LINESCTR = 0; FLAG = FALSE; CHKSUM = 0;
0:345 CALL RBFD;
0:346 DO WHILE NOT REOF;
0:347 IF FLAG AND (CHAR <> 0) THEN FLAG = FALSE;
0:348 IF NOT FLAG AND (CHAR = 0) THEN DO;
0:349 FLAG = TRUE;
0:350 /* MARK LAST NON-ZERO BYTE */
0:351 TYESCTR = BYTESCTR;
0:352 END;
0:353 CHKSUM = CHKSUM + CHAR;
0:354 BYTESCTR = BYTESCTR + 1;
0:355 IF CHAR = CR THEN LINESCTR = LINESCTR + 1;
0:356 CALL RBFD;
0:357 END;
0356 IF FLAG THEN /* STRING OF NULLS AT END */
0358 BYTCTR = TBYTCTR;
0360
0361 LISPCL = TRUE;
0362 DCTA = .RECB+XSIZ; CALL OUTDEC; /* SECTOR SIZE */
0363 CALL SPACE;
0365
0366 DCTA = .BYTECTR; CALL OUTDEC; /* BYTE COUNT */
0367 CALL SPACE; CALL SPACE;
0368
0369 CALL OUTADR; /* IN HEX */
0369
0370 CALL SPACE;
0371 DCTA = .LINESCTR; CALL OUTDEC; /* LINE COUNT */
0372 CALL SPACE; CALL SPACE;
0373
0374 DCTA = .CHKSUN; CALL OUTFX; /* CHECKSUM */
0375 END;
0376
0377 /* MAIN */
0378 DCL HEADER DATA (' DATE NS DEC HEX LINES CS',
0379 CRLF,CRLF,4);
0380
0381 DPDFXT = DXTXT;
0382 CALL RDOFPN;
0383
0384 MSGA = .HFADR; CALL PSTRNG;
0385 CALL DATE;
0386 CALL ASIZE;
0387
0388 CALL RDCLOSE;
0389 CALL WRAMS;
0390
0391 LVL CC
0010 AYDCTS BYTEF
A2A8 ASIZE PROC
AC18 CURCHR BYTEF
"DOA" CRLF LIT
0C10 CHAR BYTEF
A12I CIASS PROC
0000 DXEIN LIT
0001 DXTXT LIT
0002 DXCMD LIT
0004 DXSYS LIT
0005 DXBAK LIT
0006 DXOUT LIT
0016 DPDFXT BYTEF
0017 DCTA ADDR
A19E DCCMNT PROC
A253 DATE PROC
AC02  FIOCHR BYTE
0008  FFOF LIT
0013  ERROR BYTE
0000  FALSE LIT
0014  FCBA ADDR
A1A4  FNS PROC
B403  FNSCLIS LIT
A10A  GETCHR PROC
A138  GETFIL PROC
A164  GETHEX PROC
A366  HEADER BYTE
AD1B  INBUFF LIT
A184  INDEC PROC
A080  LINBUF BYTE
AC11  LASTTERM BYTE
AC14  LINPTR ADDR
AD30  LOAD LIT
0019  LDSCPC BYTE
0011  MSGA ADDR
A132  NXTCH PROC
001A  NUM ADDR
A150  OUTDEC PROC
A158  OUTHEX PROC
A17E  OUTADR PROC
AC19  PREVCHR BYTE
A110  PUTCHR PROC
A11C  PSTRING PROC
AD24  PCRLF LIT
C000  QSRW LIT
0001  QS04R LIT
0002  QS04W LIT
0003  QS04Y LIT
0004  QSCIS LIT
0005  QSCREW LIT
A840  RFCH BYTE
AD2A  RSTINO LIT
A15E  RPTERR PROC
A1B6  RDCLOSE PROC
A1DC  RDER PROC
A1E1  RDOPEN PROC
001D  RFOF BYTE
A216  RFBD PROC
A244  RFDE PROC
AC0E  SMOUTH BYTE
AC0F  SDAY BYTE
AC10  SYEAR BYTE
A116  SPACE PROC
A148  SETEXT PROC
00FF  TRUE LIT
0001  VERSION LIT
A106  WARMSPROC
C000  XFC LIT
0001  XFS LIT
0003  /XUN LIT
0004  XFN LIT
000C  XFX LIT
000F  XFS LIT
003B  XNC LIT

0391  EOF

***  NO ERRORS

HIGH ADDR USED: 44D6
APPENDIX D

SPL/M Reserved Words
SPL/M Reserved Words

ADDR
ADDRESS
AND
** BASED
BREAK
** BY
BYTE
CALL
DATA
DCL
DECLARE
DO
ELSE
END
EOF
GEN
GENERATE
** HIGH
IF
LIT
LITERALLY
* LOW
* MEM
* MEMA
** MINUS
MOD
** MONITOR
NOT
OR
** PLUS
PROC
PROCEDURE
RETURN
THEN
** TO
WHILE
XOR

* - Reserved word in Version 1 only

** - Reserved word in future versions; illegal in Version 1
APPENDIX E

Grammar For SPL/N
Grammar for SPL/h V1.1

<program> ::= <init> <main> EOF

<init> ::= <stmt list> | <origin> ; <stmt list>

<stmt list> ::= <stmt> ; <stmt list> <stmt> | NIL

<stmt> ::= <decl stmt> ; | <proc def> ; | <gen stmt> ;

<origin> ::= <number>:

<proc def> ::= <proc head> <stmt list> END

<proc head> ::= <identifier>: PROCEDURE ;
 | <identifier>: PROC ;
 | <origin> <proc head>

<main> ::= <stmt list> | <origin> <stmt list>

<stmt list> ::= <stmt> ; <stmt list> <stmt> | NIL

<stmt> ::= <basic stmt> ; <if stmt>

<basic stmt> ::= <assignment> ;
 | <group> ;
 | <call stmt> ;
 | RETURN ;
 | BREAK ;
 | <decl stmt> ;
 | <gen stmt> ;

<if stmt> ::= <if clause> <stmt>
 | <if clause> <basic stmt> ELSE <stmt>

<if clause> ::= IF <expr> THEN

<group> ::= <group head> <stmt list> END

<group head> ::= DO ;
 | DO WHILE <expr> ;

<call stmt> ::= CALL <identifier> ; CALL <number>
<decl stmt> ::= DECLARE <decl element>
    | DCL <decl element>
    | <decl stmt> , <decl element>
    | <origin> <decl stmt>

<decl element> ::= <identifier> <type>
    | <identifier> ( <number> ) <type>
    | <identifier> DATA <data list>
    | <identifier> LITERALLY "<number>"
    | <identifier> LIT "<number>"

<type> ::= BYTE | ADDRESS | ADDR
<data list> ::= <data head> <constant> )
<data head> ::= ( | <data head> <constant> ,
<gen stmt> ::= GENERATE <data list>
    | GEN <data list>
<assignment> ::= <variable> = <expr>
<expr> ::= <logical factor>
    | <expr> OR <logical factor>
    | <expr> XOR <logical factor>

<logical factor> ::= <logical secondary>
    | <logical factor> AND <logical secondary>

<logical secondary> ::= <logical primary>
    | NOT <logical primary>

<logical primary> ::= <arith expr>
    | <arith expr> <relation> <arith expr>

<relation> ::= = | < | > | <> | <= | >=
<arith expr> ::= <term>
    | <arith expr> + <term>
    | <arith expr> - <term>
<term> ::= <secondary>
    | <term> * <secondary>
    | <term> / <secondary>
    | <term> MOD <secondary>
<secondary> ::= <primary>
   | - <primary>  

<primary> ::= <constant>
   | <variable>
   | ( <expr> )
   | HIGH ( <expr> )
   | LOW ( <expr> )  

<variable> ::= <identifier>
   | <identifier> ( <expr> )
   | MEM ( <expr> )
   | MEMA ( <expr> )  

<constant> ::= <number> | "<string>" | .<identifier>  

<identifier> ::= <letter>
   | <identifier> <dec digit>
   | <identifier> <letter>
   | <identifier> $  

<letter> ::= A | B | C ... | Z  

<number> ::= <dec number> | <hex number> H  

<dec number> ::= <dec digit>
   | <dec num> <dec digit>
   | <dec num> $  

<hex number> ::= <dec digit>
   | <hex num> <hex digit>
   | <hex num> $  

<dec digit> ::= 0 | 1 | 2 ... | 9  

<hex digit> ::= <dec digit> | A | B | C | D | E | F  

<string> ::= <str element> | <string> <str element>  

<str element> ::= <ASCII char> | ""
This is to document version 1.3 of SPL/M, a Systems Programming Language for Microcomputers. These pages are in addition to the SPL/M Reference Manual for version 1.2.

SPL/M has proven itself a useful and appropriate language for systems and utility programming for the 6800 microcomputer. Faster than an assembler, SPL/M generates code at the rate of 1000 lines of source per minute. Code is easily block structured and simply documented for clean code generation. And I/O libraries make interfacing with various computers just a matter of substituting the appropriate libraries.

Now SPL/M is being enhanced from v.1.2 to v.1.3. There are currently four compilers running under development:

SPLM00, the enhanced 6800 compiler;
SPLM09, a 6809 compiler which runs on the 6809;
SPLM09X, a 6809 cross-compiler which runs on the 6800; and
SPLM00X, a 6800 cross-compiler which runs on the 6809.

Currently being developed are cross-compilers to generate 8088 and 6502 code.

If the enclosed disk is for generating 6809 code on a 6809 FLEX system, it contains:

SPLM09.CMD
FLX09.TXT, source for the I/O portion of SPLM09.CMD, and its LIB files, FLXA-C09, FLXB, FLXC-T68, FLXD-C09, FLXE, and FLXF.
SPLM.LIB, SPLMREAD.LIB, and SPLMWRT.LIB for FLEX09.

SPLM's transfer address remains 380H.

The I/O section (the files starting with "FLX") is located at $7000—you may relocate it elsewhere if you wish by changing it in FLXD-C00.TXT or FLXD-C09.TXT (whichever is on your disk). We have put it at $7000 to allow us larger symbol tables and thus larger programs.

Version 1.3 of SPL/M is still under development, but here are the changes from version 1.2 so far:

1) Lower case is now fully supported: within the code being compiled; in response to prompts; in naming filenames in includes; and in listing options on the command line—that is, everywhere. For identifiers and reserved words, upper and lower case are treated identically.

2) The dot-operator can be used with procedures, i.e., `proc` generates a numeric constant equal to the memory address of a procedure.

3) Jumps around data declarations: When the primitive 'DCL' is used only once with more than one set of 'DATA' declarations (each set separated by commas), for example,
DCL GOFLAG DATA (0),
      TEST DATA (1),
      RUNFL DATA (0);

only one jump is generated around all of the data code
(subject to the fix up jump limitation of 512 bytes); in
v. 1.2, a jump was generated around each `DATA` declaration; to maintain compatibility, v. 1.3 will
generate a jump around each `DATA` declaration when a
`DCL` is put in front of each one and a semicolon is used
to separate them.

4) The maximum line length is changed from 80 characters to
   132.

5) Indirect CALL's can now be made. This can be done two
   ways, both involving use of an ADDR variable:

   a) There are times when a specific address has been set
      aside to hold the address to which you want to jump.
      For example, in the Color Computer, $A002 holds
      the address of the CHROUT routine—to call it in 6809
      assembly language means writing JSR [$A002]. Doing
      the same indirect call in 6800 assembly language
      means writing several lines of code, loading X with
      the variable's address and jumping indexed (and
      indirect) through it. To do the same indirect call
      in SPLM, first declare the specific address as a
      variable,

      $$\text{0a002h:dcl jump addr;}

      \text{Then just}

      \text{CALL JUMP;}

   b) On the other hand, you may have set up a data table
      of addresses, possibly using the new .proc function,
      in your SPLM code. Your code has figured out which
      of the addresses to call. So, having declared AAA an
      ADDR variable, write:

      $$\text{AAA=mem(a\text{data});}

      \text{(or AAA=.proc or whatever) and}

      \text{CALL AAA;}

      CALLing variables was illegal in v.1.2. Now only
      calling BYTE variables is illegal—a variable byte
      wide obviously can't be holding the address of the
      procedure to be called indirectly. If you call a
      variable that has been declared as a BYTE variable, a
new error, "T" for Type Error, will be put in the
code as it's compiled, below the variable name you've
tried to call.

6) Fatal errors send messages to the screen, then return to
FLEX (WARMS). Supposed "impossible" errors send the
address at which the program failed to the screen along
with a message, then return to WARMS (if you get the
error message "IMPOSSIBLE ERROR", please send an error
report to SOFTWARE, 465 S. Mathilda Ave., Suite 104,
Sunnyvale CA 94086). No longer do fatal errors of either
type cause a register dump, then bomb to the monitor.

7) While the manual (p. 30) documents 64 levels of symbol
     table nesting before the program is too complex, it was
     wrong. The old level was 8. The new level is 30.

8) The default address at which variables are put, always
     10H until now, has been changed to 0 and put in a data
     table so the user can change it. It's called IDATA and
     is declared in the I/O section in FLX-C-T68.TXT.

9) The default address at which the program is put remains
     100H, but is now in a data table so the user can change
     it. It's called IPC and is declared in the I/O section
     in FLX-C-T68.TXT.

10) SPLM now checks numbers as it reads them and puts a "T"
     for Type Error on those hex numbers greater than 0ffffh
     and those decimal numbers greater than 65535. So now
     users get notified when they try constructions like

     DCL JUMP DATA (7E3F0OH);
     T

     which should be written

     DCL JUMP DATA (7EH,3F00H);

11) The multiply and divide routines no longer use memory
     address space: v.1.2 put variables at locations 0 and 1;
     v.1.3 uses no memory-only the registers and the stack.

12) #PAGE is the first of a series of new #directives.

     #Directives, directives to the compiler itself, were
     limited in v.1.2 to: #INCLUDE, #LIST, and #NOLIST.
     Unlike program source statements, #directives need not be
     ended with a semicolon, but must appear on a single line,
     with their first character, the "#", in column 1 of the
     line. Comments (/*comments*/) must never be put on the
     same line with a #directive.
Directives which are printed out (only #LIST, #Nolist
and #PAGE are not printed out) are not preaced by line
numbers, since they are messages to the compiler and not
source statements.

#PAGE is a page formatting command which calls for a
formfeed to be output. #PAGE does nothing, however, when
found inside a nolist area (delimited by #NOLIST and
#LIST), so that when source is not being listed,
formfeeds are obviously not required either.

#PAGE causes a change, but is never printed on the
listing itself, just as #NOLIST and #LIST are not printed
on listings.

13) #INCLUDE lines are now printed on listings to tell you
from which file the source you're reading came.

14) #SPLMVERSION is the first of two several portability
directives. Any program with lower case, for example,
or longer-than-80-column lines or use of dot-proc
requires at least version 1.3 of the compiler to compile
it. So the programmer would want to write "#SPLMVERSION:
1.3" at the beginning of the program. The SPLM compiler
spots the statement and compares the number with its own
version number, located in an internal data statement, to
be sure it can compile the program. If not, it outputs a
polite message and calls WARMS. This will become
important as future versions of SPL/M provide further
enhancements, which previous versions cannot support, and
particularly as SPL/M programmers trade, sell or give
away source code.

15) #PROCESSOR is another portability command. If a
programmer writes a GEN statement for, say, a
6809-machine-language LDY instruction, then the program
is clearly 6809-bound. He or she would want to indicate
that by inserting in the program: "#PROCESSOR: 6809".
If, on the other hand, he or she puts in a GEN statement
for a jump, the code for which is the same for 6800 and
6809 machines, the statement to include would be
"#PROCESSOR: 6809, 6800" (in either order). The
compiler, when it encounters the statement, checks to be
sure one of the named processors (separated by commas) is
the same as the processor it compiles code for. If not,
it outputs a polite message and calls WARMS. This will
become increasingly important as we do SPL/M compilers
for the 6805, the 6502 and the 8088.

Until the compiler encounters either statement
(#PROCESSOR or #SPLMVERSION), it will assume that any
version and any processor will do. Attempting to compile
a program which includes either of these two commands
using the v.1.2 compiler will result in a syntax error flag.

16) Files, either main files or INCLUDE files, can be chained together with the new #CHAIN directive. In other words, when the compiler encounters

#CHAIN NXTFIL

it closes the file it has been reading source from and opens the file NXTFIL for continued reading. Nesting INCLUDE files is still not allowed, but a file called as a #INCLUDE file could be chained to another file with #CHAIN and both would be read before the compiler returned to the main file.

#CHAIN and #INCLUDE errors, however, are fatal (both the erroneous line and an error message are put before the return to WARMS).

17) Conditional compilation is now allowed using the new #IF and #ENDIF directives. Now you can write just one program which will compile different ways (one source listing which will compile four sets of object, each with a different terminal driver, for example; or one set of source which will compile two ways, one for 6800 and one for 6809), depending on the values of a few initial LITERALS.

For example, you could set up a file PROGRAMO:

/*PROGRAM: PROGRAM FOR THE 6800*/
DCL TARGET LIT '6800';
#SPLMVERSION: 6800
#CHAIN PROGRAM

And another file PROGRAM9:

/*PROGRAM9: PROGRAM FOR THE 6809*/
DCL TARGET LIT '6809';
#SPLMVERSION: 6809
#CHAIN PROGRAM

Now PROGRAM will be written to contain the source for both 6800 and 6809 versions with #IF to differentiate:

/*PROGRAM*/
#if TARGET=6800
0A100H:;
#endif

#if TARGET=6809
0C100H:;

SPL/M A SYSTEMS PROGRAMMING LANGUAGE FOR MICROCOMPUTERS
#ENDIF

DCL VERSION LIT '1';

#if TARGET=6800
OA840H:DCL RFCB(320) BYTE;
#include SPLM00.LIB
#include SPLMRD00.LIB
#if

#if TARGET=6809
OC840H:DCL RFCB(320) BYTE;
#include SPLM09.LIB
#include SPLMRD09.LIB
#endif

/*REST OF PROGRAM*/

The compiler will compile only #IF segments which are true. So working on the 6800 computer, you can type SPLM00 PROGRAM0 and get 6800 code or SPLM09 PROGRAM9 and get 6809 code. The #SPLMVERSION protects you from doing an SPLM00 PROGRAM9 or a SPLM09 PROGRAM0: both will issue you a message noting the incompatibility and return you to WARMS.

The syntax of #IF is limited to two forms, both requiring a previously declared LITERAL:

#IF <literal-name>
#IF <literal-name> <relational-operator> <constant>

For example, #IF TARGET would evaluate TARGET just as it would be evaluated in the source line IF TARGET THEN DO; -- that is, based on whether the rightmost bit of TARGET's value is a '1' (in which case it evaluates true) or a '0' (in which case it evaluates false).

Examples of the second #IF statement, using relational operators, include the #IF TARGET=6800 above, #IF TARGET>=6800, #IF GIMIX=OFFH, #IF GIMIX=FALSE (with FALSE defined as a LITERAL earlier as well as GIMIX defined as a LITERAL earlier), and #IF TARGET<8088.

If a #IF #directive is found to be true, every statement which follows is compiled as though the #IF is not there, except that a matching #ENDIF must be encountered before the EOF ending the program.

If, on the other hand, a #IF #directive is evaluated false, then all source is ignored to the matching #ENDIF: No object is generated; the ignored source is printed out, but without line numbers; and only a subset
of the #directives are executed:

    #INCLUDE
    #CHAIN
    #PAGE
    #LIST
    #NOLIST

The portability commands #PROCESSOR and #SPLMVersions are not evaluated inside invalid-#IF segments.

#IF #directives may be nested up to 8 deep (deeper nesting causes a fatal error).

If a #IF is encountered inside a #IF segment already found invalid, the new #IF is automatically evaluated false. Now two #ENDIF #directives must be found to match both #IF's before object code generation will continue.

The #ENDIF to match a #IF should always appear in the same file. That is, if you use a #IF before calling a #INCLUDE file, do not put the matching #ENDIF in the #INCLUDE file; the matching #ENDIF must be found in the calling file following the #INCLUDE.

18) A command line option, +I, has been added. If used, the source inside invalid-#IF segments will not be printed on listings (and the #PAGE command found inside an invalid-#IF segment is not honored).

Using the +I option, you could print out separate listings for each of the sets of object a single program compiles.

19) A new "#" error flag has been created to put beneath non-fatal erroneous #directive lines. This error flag would be put for example, for incorrectly written #SPLMVersions and #PROCESSOR lines, or beneath the EOF when a #IF has not been matched with a #ENDIF at the point the EOF is reached (note: if the EOF is inside an unmatched-but-invalid-#IF segment, it won't even be seen and you'll get FLEX's "Read Past End of File" error message).

20) Symbol tables now include both the line number and the address at which a procedure, literal, or variable is declared (previously, line numbers were not included in the symbol table). This makes it simple and straightforward to use the symbol tables to reference into source-only listings (in which no object code is listed).
As has always been the case, SPL/M-generated code is interrupt-compatible. Stack space below the stack pointer is never used without first decrementing the stack pointer (thus, in case of interrupt, no data can be written over when the registers are stacked).

If this is a 6809 version of the compiler, here are two 6809 compiler design assumptions:

The compiler does not use the U register at all—we left it free for OS-9's use. An OS-9 version of SPL/M is under development.

SPLM09 does not support any direct page other than 0, at this time, so SPLM09 automatically sets the direct page to 0 in the first few bytes of every program it compiles.

Code generated by the current level of SPLM09 is not relocatable. A relocatable 6809 code generator is under development, and of necessity will be a part of the OS-9 version of SPL/M.
SPL/M LIBRARIES

The purpose of the SPL/M libraries is to create an operating system interface and I/O support functions in a portable manner. Owners of SPL/M may use the libraries in any programs they write, including programs for commercial distribution, free of any charges beyond the original purchase price of SPL/M.

The SPL/M libraries are not necessary for writing a program in SPL/M. SPL/M is often used, for example, for writing instrument controllers, an application for which a library designed to interface with a standard microcomputer operating system and computer has no use. On the other hand, some companies have found it useful to create their own libraries of routines (perhaps to put characters and strings on the display, even though it's an LCD display) which match the library routines, allowing some testing to be done with standard libraries on an IBM or SWTP before the code is recompiled with the special libraries and moved into the instrument.

Each set of SPL/M libraries creates an I/O interface to a particular operating system and/or computer. The libraries are designed to make writing to or reading from a terminal, printer, communications line, or disk files easy.

They are also designed to create an I/O interface which is completely portable between the many computers and operating systems which the different sets of libraries support: Each routine in the libraries is called in the same way and sent the identical parameters regardless of the target computer or chip.

For example, to output a message to the terminal requires setting a library parameter called MSGA equal to the address of the message (which is terminated by a 0) before calling a library routine called PUTTERMSTR, which prints it on the screen. Using the library routine allows you to ignore the incompatibilities between the FLEX operating system, which has a routine to print strings terminated by a 4, and the IBM DOS operating system, which has a routine to print strings terminated by a ']', and other operating systems which require yet other terminators for their print-string routines. The SPL/M library routine PUTTERMSTR for FLEX prints strings terminated by a 0, the SPL/M library routine PUTTERMSTR for IBM DOS prints strings terminated by a 0, and the SPL/M library routine PUTTERMSTR for all other operating systems prints strings terminated by a 0.

A full set of portable library interfaces to each DOS creates considerable code, so routines are divided into three libraries:

SPLM___.LIB (the underlines are for characters which change - SPLM000FS.LIB for 6800 FLEX running with the SWTBUG monitor, SPLM009F.LIB for 6809 FLEX, and SPLM88MI.LIB for 8088 MSDOS running on the IBM PC) is made up of routines: to output to the screen, printer, and communications line (plus a redirectable set); to clear the screen; to ring the terminal's bell; to output

Copyright (c) 1983 SOFTWEST
numbers in decimal or hex; to input (a character or a line from the terminal keyboard, a character from the communications line, a character from a redirectable source, and hex or decimal numbers); to set and get date and time; to move strings; to classify characters; and to initialize all these library routines. This library also sets initial locations for all variables in the libraries and for the program. This library may be used exclusive of the other two libraries.

SCRN__.LIB is written for specific terminals; it may or may not be portable to yours. It contains routines which get the cursor position or position the cursor, home it, clear to end of line, clear to end of screen, (all of which requires a terminal with go-to-x-y addressing) and to put underline, boldface, and reverse characters on the screen, for terminals so capable. Routines in this library call routines located in SPLM__.LIB, so that library must be included before this one is.

RDWT__.LIB is made up of routines for accomplishing disk operations: Getting and setting the working drive; getting freespace on a disk; doing a disk directory; deleting a file; renaming a file; doing a binary load; reading from two simultaneously open files (open file, read byte, and close file); and writing to two files simultaneously (open file, write byte, and close file). Routines in this library call routines located in SPLM__.LIB, so that library must be included before this one is.

The libraries are brought into a program by using the #INCLUDE statement. Because SPLM__.LIB sets the initial variable location, this library must be included prior to declaring any other compiler-located variables in your program. Of course (since SPL/M is a one-pass compiler), libraries must be included before any of their routines are called or their variables used.

Both SPLM__.LIB and RDWT__.LIB are sprinkled with conditional compilation statements to shorten the amount of code the libraries generate; you'll need to declare literals prior to including the libraries to get a number of sections to compile code. For example, to compile code from the printer routines in SPLM__.LIB, you'll have to put the following statement into your code prior to including the library:

DCL NEEDPRT LIT 'TRUE';

So just as the literal NEEDPRT controls compilation of printing routines, NEEDCOM controls com-line routines, NEEDNUMS controls numeric input and output routines, NEEDDISKUTILS controls disk utility routines (directory, freespace, rename, delete, etc.), NEEDRFCBS controls disk-read routines, and NEEDWFCBS controls disk-write routines. All are initialized to be false, so that code within will not be generated. To turn them on: declare NEEDPRT, NEEDCOM, NEEDNUMS, or NEEDDISKUTILS literally '1' or
'Z' depending on if you need one or two read or write files open at a time.

You may also trim both the size of the source file and the size of code generated by editing down the library files to just the routines and variables you need for a specific program.

There are limits to portability:

The SCRN___LIB library has the least portability. Each SCRN___LIB library supports a single terminal. Terminals must have go-to-x-y addressing to be able to implement any of the cursor functions in the library. A program which uses these functions is not portable to computers with terminals which cannot go-to-x-y; the results are unpredictable. On the other hand, programs which call for characters to be displayed in reverse, boldface, or underline are portable to terminals without such character attributes: Characters are displayed normally on such systems.

Routines, variables, and other identifiers which are not guaranteed to be portable from one machine/operating system/chip to another have been given labels which begin with "ZZ", such as "ZZLOAD", which loads a binary file into memory, but not portably. Be warned that using any library label beginning with "ZZ" in your program source puts your program's portability at serious risk.
SPLM____.LIB

SPLM____.LIB (the underlines are for characters which change
- on 6800 FLEX with the SWTBUG monitor, it's called SPLM00FS.LIB,
on 6809 FLEX SPLM09F.LIB, and on 8088 MSDOS for the IBM PC
SPLM88MI.LIB) is made up of:

- constants,
- variables,
- a routine to initialize the libraries,
- general routines,
- terminal routines (input from the keyboard; output to the screen),
- redirectable routines (input from anywhere; output to anywhere),
- comline routines (communications line via modem or local network),
- printer routines,
- time and date routines,
- move routines, and
- number input and output routines (both hex and decimal).

This library also sets an initial variable location for all
variables in the libraries and the program. Use of this library
does not require use of either of the other two SPL/M libraries.

The libraries are brought into a program by using the
#INCLUDE statement. Because SPLM____.LIB sets the initial
variable location, this library must be included prior to
declaring any other variables in your program.

SPLM____.LIB is sprinkled with conditional compilation
statements to shorten the amount of code the libraries generate;
you'll need to declare literals prior to including the libraries
to get a number of sections to compile code. This is noted in
each section to which it applies (printer, communications line,
and numbers).
Constants

SPLM.....LIB provides a set of constants to describe the environment which the library is designed for. Some constants are declared as literals because we believe there would be no purpose in patching them. Others are declared as data to allow them to be patched should different hardware present differing requirements. All are available for use by your programs.

TARGET

TARGET is a literal which specifies the target microchip for use in your source later (e.g., #IF TARGET=6800).

BS

BS equals the ASCII value which the backspace key on the keyboard returns.

ADDLFT - add line feed to terminal
ADDLFC - add line feed to communications line
ADDLFP - add line feed to printer
ADDLFD - add line feed to disk

These constants are used to determine if the library must, after sending a cr to a particular hardware device, follow the cr with a line feed (the constant is set equal to 1), or if the hardware takes care of the function or no line feed is required to be put at all (it's set equal to 0).

PRTWIDTH - number of columns your printer will print
SCRNWIDTH - number of columns on your screen
SCRNDEPTH - number of lines on your screen
Variables

SPLM_.LIB initializes a starting origin for variables and then dynamically allocates space for all the variables in both the libraries and your program. Only variables which are specifically assigned locations by your program (as opposed to those for which space must be dynamically allocated) may be declared prior to including this library.

It is permissible to remove the variable origin from the library and place it on the first variable in the program, provided that that variable really is the first variable to be dynamically allocated space in the program and provided that all variables which are listed in the library source as "page 0" variables remain so (the type of addressing used in library GEN statements requires them to be "page 0" type variables).

Most of the library variables are intended to serve solely for passing parameters to and from certain routines. A routine may use and/or change both its own parameters and any other library variable.

Except that there are certain library variables which, by design, can be guaranteed to at all times hold certain information (set either by the library itself, by your program, or by either):

LINPTR

LINPTR, an ADDR variable, is designed to point into the line buffer. It is initially set by LIBINIT to point to the first character of the first argument on the command line (following the program name which invoked this program itself). If no arguments exist on the command line, it points to the cr terminating the command line. LINPTR is automatically reset by the INBUFF routine and advanced by the NEXTCHAR routine. LINPTR must be set to point to a filename before calling many of the disk routines.

HOURS, MINUTES, SECONDS, HSECONDS

These BYTE variables must be set before calling SETTIME. They hold their values – after being set or after a call to GETTIME.

YEAR, MONTH, DAY

These variables must be set before calling SETDATE. They hold their values – after being set or after a call to SETDATE.

LASTTERM

This type BYTE variable holds the last terminator - the most recent non-alphanumeric character encountered by CLASS (and thus by NEXTCHAR, OUTDEC, OUTHEX, and OUTADDR).

Copyright (c) 1983 SOFTWEST
CURCHAR

This BYTE variable holds the most recent character parsed by NEXTCHAR.

PREVCHAR

This BYTE variable holds the character previous to the most recent character parsed by NEXTCHAR.

BUFFER

LIBINIT sets BUFFER to the address of the first byte available for a user-program data buffer.

MEMEND

LIBINIT sets MEMEND to the address of the last byte available for a user-program data buffer.

PRTON, COMON

These BYTE flags, initialized FALSE by LIBINIT, indicate whether the printer and communications line respectively have been initialized.
Library Initialization

LIBINIT

This routine initializes the libraries and sets up the line buffer, a number of variables, and the file control blocks necessary for reading or writing to disk.

When a program reaches main, the first code put is a call to LIBINIT. This is done automatically, provided you've previously included LIBINIT in your file (either SPLM____.LIB's LIBINIT or your own). This guarantees that whole sets of parameters on which other library routines depend will be initialized. If you haven't included SPLM____.LIB, or if LIBINIT has been removed from the library or its name changed, then no automatic call is generated.

LIBINIT sets up:

BUFFER, an ADDR variable which holds the address of the first byte of buffer space available to your program.

MEMEND, an ADDR variable which holds the address of the highest memory location available to your program. You may design a text-processing program, for example, to read in as much text as possible, filling memory from the location in BUFFER to the location in MEMEND.

A line buffer, which holds the command line, and LINPTR, an ADDR variable, which points into the line buffer. Initially, LINPTR points to a cr (0DH, a carriage return) if the program name was the only word typed on the command line which invoked the program. Otherwise, LINPTR points to the first non-delimiter character following the program name. (Warning: Calling INBUFF changes the contents of the line buffer and resets LINPTR to point to the beginning of the new contents.)

File control blocks: If you have literally declared NEEDRFCBS to be 1 or 2, then LIBINIT creates 1 or 2 read file control blocks, respectively. If you have literally declared NEEDWFCBS to be 1 or 2, then LIBINIT creates 1 or 2 write file control blocks.

Initial I/O vectors:

PUTTERM is vectored to output normal screen characters (as opposed to reverse, boldface, etc.).
PUTCHAR is vectored to PUTTERM, to put characters to the screen.
GETCHAR is vectored to GETTERMINVIS, to get characters from the keyboard.

Flags PRTON and COMON: set false to indicate that neither printer nor communications line has been initialized.

Copyright (c) 1983 SOFTWEST
MSDOS: Interrupts are enabled (making the keyboard live even when the program is elsewhere).

FLEX: The screen pausing flag and the screen width are saved for restoration in DOSRET.
General Routines

DOSRET

This routine terminates a program, restores any previously saved parameters, and returns to DOS.

The last code put in a program is a call to DOSRET; this is done automatically when the EOF end-of-file operator is parsed, provided you've previously included DOSRET in your file.

UPPER

This routine converts lower to upper case: If the ASCII value in the BYTE variable CHAR represents a lower case letter, it is converted to upper case.

CLASS

This routine classifies the value in the BYTE variable CHAR: Upon exit, if the value in CHAR is not a letter or a number (not alphanumeric), the BYTE variable ERROR is set TRUE and the value in CHAR is automatically stored in the BYTE variable LASTTERM; on the other hand, if CHAR is alphanumeric, ERROR is set FALSE.

CLASSALPH

This routine also classifies the value in the BYTE variable CHAR: Upon exit, if the value in CHAR is not a letter (not alphabetic), the BYTE variable ERROR is set TRUE; on the other hand, if CHAR is alphabetic, ERROR is set FALSE.

CLASSNUM

This routine also classifies the value in the BYTE variable CHAR: Upon exit, if the value in CHAR is not a number (not numeric), the BYTE variable ERROR is set TRUE; on the other hand, if CHAR is numeric, ERROR is set FALSE.
Terminal Routines

CLRTERM

A call to CLRTERM clears the terminal screen.
MSDOS: CLRTERM calls the IBM BIOS INT 10H.
FLEX: CLRTERM clears the screen by sending the character in ZZCLR (normally the formfeed character, 0CH) to PUTTERM.

PUTTERM

Output the character in CHAR to the terminal. If the character is a carriage return, then if ADDLFT is other than zero, then a line feed is also output. If the character is a backspace, and the terminal can backspace, then PUTTERM does the backspace, writes a space at this position, and remains there.

PUTTERM is revectorable. LIBINIT initializes PUTTERM to a standard teletype kind of output to the screen (one character at a time at the cursor, with the cursor position moving right and down). Calling the BEGSPECIALSCRN routine in the SCRNN_.LIB library revector PUTTERM to the screen output routine in that library, which allows cursor positioning and bold, reversed, and underlined characters. Calling ENDSPECIALSCRN resets PUTTERM to teletype screen output.

PUTTERM is intended primarily for guaranteeing message output to the screen regardless of where the main output through PUTCHAR is vectored.
FLEX: If PUTCHAR is outputting to the printer, PUTTERM will ignore the TTYSET parameters like width and pausing.

PUTTERMSPC

Send one space to the screen.

PUTTERMNUMSPC

Send NUM number of spaces to the screen (set NUM equal to the number of spaces you want before calling PUTTERMNUMSPC).

PUTTERMCR

Send one carriage return (and, if ADDLFT is not zero, a matching line feed) to the screen.

PUTTERMNUMCR

Send to the screen NUM number of carriage returns (and, if ADDLFT is not zero, matching line feeds). Set NUM equal to the number of CRLFs you want before calling PUTTERMNUMCR.

PUTTERMSTR

Copyright (c) 1983 SDFTWEST
Output to the screen a string which is terminated by a zero (\(0\)) (the zero indicates the end of the string; it is not output). Set MSGA to the location of the first byte in the string before calling PUTTERMSTR. For example:

DCL MSG1 DATA (CR,'This is a message.\r\0);  
MSGA=\&MSG1; /*Set MSGA to point to MSG1*/  
CALL PUTTERMSTR; /*Output MSG1 to the screen*/

**PUTBELL**

Ring the terminal's bell.

**GETTERM**

Get one character from the keyboard and echo it to the screen. This and the other get-character routines will halt a program until a character is typed on the keyboard.

MSDOS: None of the routines which get a character from the keyboard will return extended ASCII (a \(0\) followed by a code), except that a \(0\) followed by a 3, which represents the CTRL-\(0\), is returned as its accepted ASCII value of \(0\). Other extended ASCII characters are ignored and the routine continues to await a valid character.

**GETTERMINVIS**

Get one character from the keyboard and do not echo it to the screen.

FLEX: The FLEX operating system does not provide an echo-less getchr routine. So the library routine goes directly to the SWTBUG monitor to turn off echo before calling FLEX's GETCHR. Other monitors may require revisions to this routine.

**KBDSTAT**

Check the keyboard. If a key has been pressed, CHAR is set TRUE (to read the depressed key, follow with a call to GETTERM or GETTERMINVIS). If no key has been pressed, CHAR is set FALSE. (To actually read a pressed key, call KBDSTAT; if it returns TRUE, then call GETTERM or GETTERMINVIS.)

68000 FLEX: KBDSTAT is dependent on Z2KBDTYP being set to \(0\) for serial keyboard or \(1\) for parallel keyboard, and on Z2KBDLOC, initially set for the keyboard to be connected to Port 1 (location \(3004H\)).

**INBUFF**

Input a line (terminated by the user pressing ENTER or RETURN) from the keyboard into the line buffer. A cr is placed in the buffer at the end of the line. On exit, the ADDR variable LINPCTR points to the first character in the line buffer. Note: The line buffer is used on entry to a program to hold the remainder of the command line; since calls to INBUFF would replace that command line with the line from the keyboard, any parsing of the command line must

Copyright (c) 1983 SOFTWARE
be done prior to calling INBUFF.

NEXTCHAR

Set the character pointed to by LINPTR and both return it in CHAR and save it in CURCHAR (after first saving CURCHAR's contents to PREVCHAR). NEXTCHAR calls CLASS before returning: if CHAR is alphanumeric, ERROR is set FALSE; otherwise, ERROR is set TRUE and CHAR is also stored in LASTTERM.

If CHAR is a carriage return (or in FLEX: if it's either a cr or the TTYSET End-of-Line character). then LINPTR is not advanced, and subsequent calls to NEXTCHAR return the same character.

Otherwise, LINPTR is advanced to point to the next character in the line buffer. If CHAR is a space, then NEXTCHAR advances LINPTR to point to the first non-space character (so multiple spaces are skipped and a single space is returned).
Redirectable Routines

PUTCHAR

Output the character in CHAR. LIBINIT initializes PUTCHAR to output to the screen. PUTCHAR is revectorable to the printer (CALL PICKPUTPRT), to the communications line (PICKPUTCOM), or to either disk file that’s been opened for writing (PICKWFCE1 and PICKWFCE2), as well as restorable to the screen (RSTRPUTTERM). See PUTTERM, PUTPRT, PUTCOM, WBDI1, and WBDI2 for details on how characters are output to each device. In the case of output to the screen, revectoring PUTTERM to special screen capabilities (bold and reverse characters and cursor positioning) See SCRNLINUX.LIB) revectors PUTCHAR’s screen output to those capabilities, too.

RSTRPUTTERM

Calling RSTRPUTTERM revectors PUTCHAR to the screen. If it’s already vectored to the screen, there’s no effect.

FLEX: Calling RSTRPUTTERM after printing restores FLEX’s screen parameters (pausing, width), in addition to revectoring PUTCHAR to the screen.

PUTSPC

Send one space out through PUTCHAR.

PUTNUMSPC

Send NUM number of spaces out through PUTCHAR (set NUM equal to the number of spaces to be output before calling PUTNUMSPC).

PUTCRLF

Send one carriage return (and line feed if the appropriate ADDLF... add-line-feed flag is not zero) out through PUTCHAR.

PUTSTR

Output through PUTCHAR a string which is terminated by a zero (\0); the zero terminator is not output. Set MSGA equal to the address of the first byte in the string before calling PUTSTR.

GETCHARINVIS

Get one character: do not echo it to the screen. GETCHARINVIS is redirectable. Initialized by LIBINIT to get the character from the keyboard, GETCHARINVIS may be redirected to get it from the communications line (PICKGETCOMINVIS) or from either read file (PICKRFBD1 and PICKRFBD2). RSTRGETTERMINVIS restores GETCHARINVIS to get its characters from the keyboard again.

There is no redirectable GETCHAR routine in the library.
(get one character and echo it to the screen): If you don't need redirection but you want echo, then call GETTERM; if you really do need both redirection and echo, then make two calls, the first to GETCHARINVIS, the second to PUTTERM.

RSTRGETTERMINVIS

Restores the GETTERMINVIS keyboard input routine as the source of characters for the redirectable GETCHARINVIS routine.
Comline Routines

Comline routines are designed to put characters out through an RS232 port to a communications line, or to get characters from that communications line.

Comline routines are not normally compiled: They are conditionally compiled by the compiler directive #IF NEEDCOM, which defaults to FALSE. To compile the comline routines, type DCL NEEDCOM LIT 'TRUE'; in your program before the #INCLUDE SPLM____.LIB.

COMINIT

Initialize the communications line. This routine is called automatically upon the first call to either GETCOM or PUTCOM, if it hasn't been already initialized by a direct call. (It knows because of the BYTE flag COMON.)

FLEX and MSDOS: A nonportable BYTE DATA item, ZZCOMDEFS, is set to initialize the communications line for no parity, 1 stop bit, and 8-bit word length.

MSDOS: ZZCOMDEFS also sets the IBM's software-controlled default baud rate to 2400 baud. Comline routines assume the first RS232 card. The COMINIT routine uses the IBM BIOS INT 14H.

FLEX: The hardware controls the baud rate. The nonportable ADDR DATA item ZZCOMPORT locates the communication line ACIA in Port 0 (location 8000H).

PUTCOM

Output a character in the BYTE variable CHAR to the communications line. If necessary (if COMON is FALSE), first call COMINIT to initialize the comline. If the character is a carriage return and ADDLFC is not zero, then PUTCOM puts a line feed to the comline following the cr.

PICKPUTCOM

Revector PUTCHAR’s output to PUTCOM.

PUTCOMSTR

Output the string, terminated by 0 and pointed to by MSSA, to the communications line.

GETCOMINVIS

Get a character from the communications line (no echo to screen). If necessary (if COMON is FALSE), first call COMINIT to initialize the comline.

GETCOM

Get a character from the communications line (by calling GETCOMINVIS), then echo the character to the screen.

Copyright (c) 1983 SOFTWEST
PICKGETCOMINVIS

Revector GETCHAR to get its characters from GETCOMINVIS.

COMSTAT

Check the status of the communications line. CHAR is set TRUE if a byte is ready to be received (receiver data register is full). SENDFLAG is set TRUE if communications line is free to send another byte (transmitter data register is empty).
Printer Routines

Printer routines are designed to output characters to a printer.

Printer routines, like comline routines, are not normally compiled: They are within a #IF NEEDEDRT conditional compiler directive, and NEEDEDRT is by default FALSE. To compile the printer routines, type DCL NEEDEDRT LIT 'TRUE'; in your program before the #INCLUDE SPLM____.LIB.

PRTINIT

Initialize the printer. This routine is called automatically upon the first call to PUTPRT, if it hasn't already been called directly (it knows because the BYTE flag PRTON remains FALSE until PRTINIT is called). Suggestion: Because FLEX can return from PRTINIT uninitialized (because it can't find PRINT.SYS, or because the printer is already busy spooling), you will be safest to call PRTINIT directly, then test for PRTON being true (successful initialization).

FLEX: PRTINIT loads PRINT.SYS if necessary. It also turns pausing off and sets TTYSET width to 0.

PUTPRT

Output a character in the BYTE variable CHAR to the printer. If necessary (if PRTON is FALSE), first call PRTINIT to initialize the printer. If the character is a carriage return and ADDLF is not zero, then PUTPRT puts a line feed to the printer following the return.

FICKPUTPRT

Revector PUTCHAR's output to PUTPRT.

FLEX: Turns off pausing and sets the TTYSET width to 0. (Previous width and pausing status are saved; they are restored by calls to RSTRPUTTERM or DOSRET.)

PUTPRTSTR

Output the string, which is terminated by 0 and pointed to by MSGTA, to the printer.
Time/Date Routines

SETDATE

Set the month, day and year. Before calling, set BYTE variable MONTH equal to 1 to 12, BYTE variable DAY equal to 1 to 31, and ADDR variable YEAR equal to 1980 to 2079. On return, ERROR is FALSE if the set operation was successful.

SETTIME

Set the time. Before calling, set BYTE variables HOURS to 0 to 23, MINUTES to 0 to 59, SECONDS to 0 to 59, and HSECONDS (hundreds of a second) to 0 to 99. On return, ERROR is FALSE if the set operation was successful.

FLEX: If you have a clock card, you'll have to rewrite this routine to set it; as written, it returns with ERROR set TRUE.

GETDATE

Get the date. On return, MONTH equals 1 to 12, DAY equals 1 to 31, and YEAR equals 1980 to 2079.

GETTIME

Get the time. On return, BYTE variables HOURS should return 0 to 23, MINUTES 0 to 59, SECONDS 0 to 59, and HSECONDS (hundreds of a second) 0 to 99. If time is not available, all will be set to 0FFH.

FLEX: If you have a clock card, you'll have to rewrite this routine to get it; as written, it returns with all four variables set to 0FFH.
Move Routines

Move routines are designed for moving an array of bytes from one location to another. Note: These routines should not be used if the source and destination arrays overlap.

MOVECR

Move a line of any length ended by a cr from SOURCE to DEST. Set SOURCE and DEST, pointers to the beginning byte of the source and the destination arrays, before calling.

MOVENUM

Move NUM number of bytes from SOURCE to DEST. Set SOURCE and DEST, pointers to the beginning bytes of the source and the destination arrays, and NUM before calling.

MOVECRNUM

Move a line ended by a cr - but a maximum of NUM bytes - from SOURCE to DEST. Set NUM, SOURCE and DEST before calling. If a cr is not found by the NUMth byte, the NUMth byte at the destination is set to a cr.
Number Routines

Number output routines are designed to output (redirectably), in either hex or decimal form, numbers which are held in a variable. Number input routines are designed to take a string of hex or decimal digits, convert them into a number in binary form, and return it in the ADDR variable NUM.

Number routines are not normally compiled: They are conditionally compiled based on NEEDNUMS, and NEEDNUMS defaults to FALSE. To compile the number routines, type DCL NEEDNUMS LIT 'TRUE'; in your program before the #INCLUDE SPLM___.LIB.

FLEX: The number output routines are redirectable both for portability and for useability. If you need solely to send numbers to the screen, you may use FLEX’s number output routines, which are much shorter:

Replace the innards of PUTDEC with:

```
GEN(0D6H,.LEADSPC); /*LDAB LEADSPC*/
GEN(0DEH,.DGT);    /*LDX DGT*/
CALL 0AD39H;       /*CALL FLEX’S OUTDEC ROUTINE*/
```

Replace the innards of PUTHEX with:

```
GEN(0DEH,.DGT);    /*LDX DGT*/
CALL 0AD3CH;       /*CALL FLEX’S OUTHEX ROUTINE*/
```

Replace the innards of PUTADDR with:

```
GEN(0DEH,.DGT);    /*LDX DGT*/
CALL 0AD45H;       /*CALL FLEX’S OUTADDR ROUTINE*/
```

PUTDEC

Output (redirectable) in decimal an unsigned 16-bit number, the address of which is in DGT. Before calling, if the number is held in a BYTE variable, then reassign it to an ADDR variable; set DGT to point to the address of the ADDR variable which holds the number. Set the BYTE variable LEADSPC equal to TRUE to right-justify the number in a five-character field (that is to say, to print a space for each leading zero); set LEADSPC to FALSE to left-justify the number (to output only digits starting with the first non-zero one).

PUTHEX

Output (redirectable) as two hex digits an unsigned 8-bit number, the address of which is in DGT. Before calling, set DGT to point to the address of the BYTE variable which holds the number.

PUTADDR

Output (redirectable) as four hex digits an unsigned 16-bit number, the address of which is in DGT. Before calling, if
the number is held in a BYTE variable, then reassign it to an ADDR variable — or call PUTHEX instead; set DGTA to point to the address of the ADDR variable which holds the number.

GETHEX

Get unsigned hex digits and convert them into a 16-bit binary number. If the hex digits are already in memory, set LINPTR to point to the address of the first digit. Or to get the hex number from the user, CALL INBUFF, then CALL GETHEX.

On return: ERROR is TRUE if LINPTR points to an invalid number or FALSE if LINPTR points to a valid number or to a separator character; use ANYDIGITS if ERROR is FALSE — then if ANYDIGITS is other than zero then LINPTR is pointing to a valid number, but if ANYDIGITS is zero then LINPTR points to a separator character. If a valid number is found, it’s returned in NUM (truncated to 16 bits); NUM returns a zero if LINPTR points to a separator character; LINPTR is left pointing to the character following the separator character, unless the separator is a cr (the same rule as for NEXTCHAR).

GETDEC

Get an unsigned decimal number (a series of ASCII decimal digits) and convert it into a 16-bit binary number. If the number is already in memory (as digits in a string), set LINPTR to point to the address of the first digit. Or to get the decimal number from the user, CALL INBUFF, then CALL GETDEC.

On return: ERROR is TRUE if LINPTR points to an invalid number or FALSE if LINPTR points to a valid number or to a separator character; use ANYDIGITS if ERROR is FALSE — then if ANYDIGITS is other than zero then LINPTR is pointing to a valid number, but if ANYDIGITS is zero then LINPTR points to a separator character. If a valid number is found, it’s returned in NUM (truncated to 16 bits); NUM returns a zero if LINPTR points to a separator character; LINPTR is left pointing to the character following the separator character, unless the separator is a cr (the same rule as for NEXTCHAR).
SCRN___.LIB

SCRN___.LIB is made up of:

cursor positioning routines, and
special screen character routines.

Routines in this library call routines located in
SPLM___.LIB, so that library must be included before this one
is.

SCRN___.LIB is written for specific terminals; it may or may
not be portable to yours. The SCRN___.LIB library has the least
portability of the libraries. Each SCRN___.LIB library supports
a single terminal. Terminals must have go-to-x-y addressing to
be able to implement any of the cursor functions in the library;
a program which uses these functions is not portable to computers
with terminals which cannot go-to-x-y. On the other hand,
programs which call for characters to be displayed in reverse,
boldface, or underline are portable to terminals without such
character attributes, but without the specially displayed
characters; in this case, the SCRN___.LIB routines would be dummy
routines - they would consist only of

name:PROC;
END:
Cursor Positioning

SCRN____.LIB provides a set of routines which set and get the
cursor position, and which clear a line or lines starting from
the cursor position. Terminals must have go-to-x-y addressing to
be able to implement any of the cursor functions in the library;
since each terminal is different, each terminal needs a
SCRN____.LIB custom-designed for it.

GETCURSPOSN

Get the current cursor position into the BYTE variables ROW
and COLUMN. The upper left position is (0,0).

POSNCURS

Move the cursor to the position specified by the BYTE
variables ROW and COLUMN. The upper left position is
(0,0).

HOMECURS

Move the cursor to the home position (the upper left
corner), which is row 0, column 0.

CURSDOWN

Move the cursor down one row, but maintain the same column
position. If the cursor is already on the bottom row, do
not change its position.

CURSUP

Move the cursor up one row, but maintain the same column
position. If the cursor is already on the top row, do not change its position.

CURSFORWARD

Move the cursor forward one column, on the same row. If the
cursor is already in the last column, do not change its
position.

CURSBACK

Move the cursor back one column, on the same row. If the
cursor is already in the first column, do not change its
position.

CLREAL

Clear from the cursor to the end of the line.

CLREOS

Clear from the cursor to the end of the screen.
Special Screen Characters

SCRN..LIB provides a set of routines for sending characters to the screen with special attributes - bold, underline, and reverse. If the terminal to which a particular SCRN..LIB is directed does not support one or more of these features, a CALL to those routines does nothing.

BEGSPECIALSCRN

Redirect the output of PUTTERM (and, when going to the screen, of PUTCHAR - that is, redirect the output of all screen output routines) to a screen driver which allows output of characters with special attributes. This routine does not turn on any of the special attributes - that's done using BEGULCHARS, BEGBFCHARS, and BEGREVCHARS.

The routine also takes care of any initialization required to prepare for output of special characters. For example, SCRNN0FG.LIB for the 6800 FLEX GIMIX video card, as written, initializes the card to allow reverse characters to be output.

If the terminal has lowlight/hilight capabilities, then BEGSPECIALSCRN puts it into lowlight mode. On the IBM, this causes no change, with normal characters output as before, and boldface characters in the IBM's double-intensity mode. On many terminals, however, lowlight is half-intensity; on these terminals, BEGSPECIALSCRN initializes the terminal so that normal characters are now output as half-intensity, with boldface characters output at the normal intensity.

ENDSPECIALSCRN

Return screen output to normal channels; do not allow characters to be output with special attributes.

BEGULCHARS

Begin underlining: Underline every character which follows which is sent to the screen.

BEGBFCHARS

Begin boldfacing: Boldface every character which follows which is sent to the screen.

BEGGREVCHARS

Begin reversing: Reverse every character which follows which is sent to the screen.

ENDULCHARS

End underlining of characters to the screen.

ENDBFCHARS
End boldfacing of characters to the screen.

ENDREYCHARS

End reversing of characters to the screen.

RSTRNORMCHARS

End any special character attributes being sent to the screen, and restore output of normal characters (but don’t revector the screen output routines from the special character screen driver – that’s a job for ENDSPECIALCHARS).
RDWT____.LIB creates a portable set of routines for reading from and writing to disk.

Text files pose a portability problem: Some systems, like MSDOS, terminate lines stored on disk with two bytes, a cr/lf pair; others, like FLEX, use a single byte, a cr, as a terminator. For portability, lines are returned by the SPL/M library read routines terminated by a single cr, regardless of system. Thus, in the MSDOS operating system, in which lines in standard text files on disk are terminated by carriage return-linefeed pairs, the SPL/M text-file write-byte-to-disk routines automatically write a linefeed character to disk after writing each carriage return character to disk. Similarly, the MSDOS library routines to read bytes from disk automatically strip off a linefeed which immediately follows a carriage return in a standard DOS file. In FLEX text files, on the other hand, linefeeds are not added or removed, since lines in standard FLEX text files on disk are terminated only by carriage returns.

RDWT____.LIB is made up of:

constants,
disk utility routines,
read file routines, and
write file routines.

Routines in this library call routines located in SPLM____.LIB, so that library must be included before this one is.

All successful calls to disk routines return the BYTE variable ERROR set FALSE; if there was any problem, however, ERROR is returned set TRUE. The BYTE variable ZERRNO may also be set to one of the error literals to indicate which type of error occurred; but all start with the 'ZZ' non-portability indicator because, unfortunately, the types of errors which may be returned from disk routines vary enormously from one system to another.

Three #IF statements control generation of code within RDWT____.LIB: NEEDDISKUTILS controls disk utility routines (directory, freespace, rename, delete, etc.), NEEDRFCBS controls disk-read routines, and NEEDWFCBS controls disk-write routines. All are initialized to be false, so that source they surround will not generate code. To turn on code generation: declare NEEDDISKUTILS literally TRUE; declare NEEDRFCBS or NEEDWFCBS literally '1' or '2' depending on if you need one or two read or write files open at once.

There is one routine which is always compiled, regardless of conditional compilation.

CLOSEALLFILES
Close any disk files which are open, either for reading or for writing.
Constants

RDWT.__.LIB provides a set of constants for portability between different disk operating systems:

FIRSTDRIVE, SECONDDRIVE,...FOURTHDRIVE, WORKDRIVE, SYSDRIVE

A drive letter or number is specified to the directory routine (DIR) by sending it a literal: FIRSTDRIVE, SECONDDRIVE, THIRDDRIVE, and FOURTHDRIVE are fairly obvious; WORKDRIVE and SYSDRIVE specify, respectively, the working drive (location of text or data files) and system drive (location of commands) on systems which have such designations; on other systems which have only one such automatically selected drive, they both specify the "default drive."

DRIVEBIAS

DRIVEBIAS is a literal which, added to FIRSTDRIVE, converts it to the ASCII character used to specify the first drive ('A' in MSDOS, '0' in FLEX). A program which calls the directory routine might, for example, prompt the user for the drive letter of the directory desired.

DRIVESEP

This is the ASCII character which, in a filename specification, separates drive letter from filename, useful for parsing or building filenames.

EXTSEP

This is the ASCII character which is used to separate a filename from its extension, useful for parsing or building filenames.

MAXFILONLEN

MAXFILONLEN specifies the number of bytes needed to hold a full-length filename plus a terminator (such as a carriage return). Use this to specify the length of an array you intend to use for storing or building a filename. Included in MAXFILONLEN is room for the drive letter or number, the drive separator, the filename, the extension separator, the extension, and the terminator character (e.g., 1.FILENAME.TXT or A:FILENAME.TXT – plus a carriage return terminator).
Disk Utilities

RDWT---.LIB provides a set of disk utility routines. Declare NEEDDISKUTLIS literally TRUE before including the RDWT library into your program to get these routines to compile.

GETDRIVE

Return in the BYTE variable CHAR the ASCII letter or number of the working (default) drive. This value may be converted to one of the portable literals (FIRSTDRIVE, etc.) by subtracting the literal DRIVEBIAS.

CHANGEDRIVE

Change the working (default) drive to the one specified. Before calling CHANGEDRIVE, set CHAR equal to the ASCII drive letter or number (convert one of the portable drive literals, like FIRSTDRIVE, by adding the DRIVEBIAS literal). If the drive letter or number is invalid, then an error message 'INVALID DRIVE LETTER' is output to the screen and ERROR is set TRUE (and ZZERRNO is set equal to ZZEIDS).

FREESPACE

Return the number of free sectors available on the disk specified. Before calling, set CHAR to one of the drive number literals (FIRSTDRIVE, etc.). On return, the ADDR variable NUM contains the number of free sectors (unless ERROR has been set TRUE).

DIR

Output to the terminal a directory or catalog of the disk specified, including a one-line report on the free space left on the disk. Pauses at screenfuls (hit a character to continue). Before calling, set CHAR to one of the drive number literals (FIRSTDRIVE, etc.). To guarantee keeping the final screenful from scrolling off the screen, your calling program must put no more than one linefeed before pausing itself (for example, after the call to DIR it might output a prompt preceded by a single cr using PUTTERMSTR, then call GETTERM, which would pause to await a response). If there is an error in doing the directory, ERROR is returned TRUE.

FLEX: DIR uses the FLEX "DO-COMMAND" routine to call from disk FLEX's CAT (or any other you choose) command, the name of which is in the data statement, ZZDIRCMD. If you've changed "CAT" to another name, or if you wish to use a directory command other than "CAT", change the ZZDIRCMD data statement to the name of your catalog command.

DELETEFILE

Copyright (c) 1983 SOFTWEST
Delete a disk file. Before calling, set LINPTR to point to the first character of the filename, which should be terminated by a valid separator character (comma, space, or cr on FLEX, for example). On return, ERROR is set TRUE if no file was deleted; and LINPTR is updated to point to the first character following the separator or separators, except it will point to the separator itself if it’s a carriage return. The file being deleted must not be already open. FLEX: DELETEFILE defaults to the extension .TXT.

RENAMEFILE

Rename a disk file. Before calling, set DEST to point to the first character of what will be the new filename, which should be terminated by a valid separator character; set SOURCE to point to the first character of the filename to be renamed, which should be terminated by a valid separator character. On return, ERROR is set TRUE if no file was renamed. The file being renamed must not be already open. FLEX: The extension of the filename to be renamed defaults to .TXT if none is specified; the extension of what will be the new filename defaults to the extension of the original name if none is specified.

Z7LOAD, etc.

Routines are provided for loading a binary file into memory. These are totally non-portable: Each is different on different systems. See the particular library’s source code for parameters and details.
Read Files

SPLM....LIB provides a set of disk read routines. Declare NEEDRFCBS literally '1' to get routines to compile for opening, reading from, and closing one read file at a time. Declare NEEDRFCBS literally '2' to get routines to compile for opening, reading from, and closing two read files simultaneously.

RDOPEN1FORTEXT

Open a file (which we will generically call "readfile1") for reading text. Before calling, set LINPTR to point to the first character of the filename; the filename should be terminated by a valid separator character. On return, ERROR is set TRUE if the filename was invalid or if the file could not be found; ERROR is set FALSE and RIOpen is set TRUE if the file was successfully opened; and LINPTR is updated to point to the first character following the separator or separators, except it will stop and point to a carriage return if it encounters that character.

MSDOS: Sets up linefeed suppression in textfile cr/lf pairs; looks for CTRL-Z as end-of-file flag.

FLEX: Sets default extension of filename to be opened as .TXT; sets up space compression for reading text.

RDOPEN1FORBIN

Open readfile1 for reading, as above in RDOPEN1FORTEXT, except set it up for binary read.

MSDOS: Binary files find end-of-file by counting bytes and comparing to number of bytes listed as being in the file.

FLEX: Sets default extension of filename to be opened as .BIN; disables space compression for reading binary.

RBFD1

Read a byte from disk readfile1 into the BYTE variable CHAR. The file must have previously been successfully opened. The calling program need not check the value of ERROR: All read errors (other than finding end-of-file) are fatal (they result in a call to DOSRET).

At the end of the file: RBFD1 returns the last character in the file; then, the next call to RBFD1 returns READ (read end of file) set TRUE. ERROR may also be set TRUE on read-end-of-file - but use READF1 to check for no more bytes in the file left to be read. To read all the bytes in a file into memory, you might, for example, use the following code:

CALL RBFD1:
DO WHILE READF1=FALSE:
   MEM(MEMORYPOINTER)=CHAR;
   MEMORYPOINTER=MEMORYPOINTER+1;
   CALL RBFD1:
END:

Copyright (c) 1983 SOFTWEST
MSDOS: In files opened for reading text, carriage-return-line-feed pairs return only a carriage return to your program; and Ctrl-Z is considered end-of-file.

FLEX: In files opened for reading text, space compression is set up; in files opened for reading binary, space compression is disabled.

**RDCLOSE1**

Close readfile1. ERROR should be returned FALSE. R1OPEN is reset from TRUE to FALSE, indicating readfile1 is no longer open. Any file-closing operations needed are performed.

**PICKRBFD1**

Pick RBFD1 as the source for the redirectable input routine GETCHARINVIS. You'll still have to open and close readfile1, though, before and after reading from it.

**RDOPEN2FORTEXT, RDOPEN2FORBIN, RBFD2, RDCLOSE2, and PICKRBFD2**

These routines open, read from, and close a second read file; they are completely orthogonal with the set of routines just described (with the number "1" in them) except that these routines use a second file control block for reading from disk. NEEDEDFCBS must have been declared literally '2' or more for these routines to compile.
Write Files

SPLM....LIB provides a set of disk write routines. Declare NEEDWFCSBS literally '1' to get routines to compile for opening, writing to, and closing one write file at a time. Declare NEEDWFCSBS literally '2' to get routines to compile for writing to two write files simultaneously.

WTOPEN1FORTEXT

Open writefile1 for writing text. Before calling, set LINPTR to point to the filename, which should be terminated by a valid separator character. On return, ERROR is set TRUE if the filename was invalid or if the filename already exists as a file on the disk or if the disk is write-protected (in FLEX); ERROR is set FALSE and W1OPEN is set TRUE if the file was successfully opened; and LINPTR is updated to point to the first character following the separator or separators, except it will stop and point to a carriage return if it encounters that character.

MSDOS: Adds a final Ctrl-Z as textfile end-of-file,
when closing the file: automatically writes a linefeed following every carriage return to create standard MSDOS text files which can be read with the MSDOS TYPE command (can be disabled by setting ADDLFD equal to zero).

FLEX: Sets default extension of filename to be opened as .TXT; sets up space compression for writing text.

WTOPEN1FORBIN

Open writefile1 for writing, as above in WTOPEN1FORTEXT, except set it up for binary write.

FLEX: Sets default extension of filename to be opened as .BIN; disables space compression for reading binary.

WRTD1

Write one byte in CHAR to the disk writefile1. If the byte is a carriage return, and the file was opened to write text, and ADDLFD is other than zero, then a linefeed character is automatically and immediately written to disk after the carriage return. Disk-full errors return with ERROR set TRUE and the character unwritten to the disk.

WTCLOSE1

Close writefile1.

MSDOS: If the file was opened for text, output a final Ctrl-Z end-of-file marker before closing the file.

PICKWRTD1

Pick WRTD1 as the output vector for the redirectable output routine PUTCHAR. You'll still have to open and close writefile1, though, before and after writing to it.

Copyright (c) 1983 SOFTWEST
WTOPEN2FORTEXT, WTOPEN2FORBIN, WBTD2, WCLOSE2, PICKWBTD2

These routines open, write to, and close a second disk file; they are completely orthogonal with the set of routines just described (with the number "1" in them) except that these routines use a second write file control block for writing to disk. NEEDWFCS must have been declared literally '2' or more for these routines to compile.