SL5 System Reference Manual

Information contained in this manual is disclosed in confidence and may not be duplicated in full or in part by any person without prior written approval of The Stackworks. Its sole purpose is to provide the user with adequately detailed documentation so as to efficiently install, operate, and maintain the system supplied. The use of this document for all other purposes is specifically prohibited.

COPYRIGHT 1980
By The Stackworks
321 E Kirkwood Avenue
P.O. Box 1596
Bloomington IN 47402
(812) 336-1600

References are made throughout this manual to the CP/M operating system, to the Zilog Z-80 microprocessor, and to the Intel 8080 microprocessor. CP/M is a registered trademark of Digital Research of Pacific Grove CA. Z-80 is a registered trademark of Zilog Inc. 8080 is a registered trademark of Intel Corp.

Statement of Warranty

SuperSoft disclaims all warranties with regard to the software contained on diskette, tape, or printed form, including all warranties of merchantability and fitness; and any stated express warranties are in lieu of all obligations or liability on the part of SuperSoft for damages, including but not limited to special, indirect or consequential damages arising out of or in connection with the use or performance of the software licensed.

Transferability

SuperSoft software and manuals are sold on an individual CPU basis and NO rights for duplication are granted.

Title and ownership of the software and manual shall at all time remain with SuperSoft.

It is understood that acceptance of this software product implies agreement with the above policies.

Preface

SL5 is more than a language; it is a complete approach to small systems programming. The distributed CP/M-compatible diskette contains every line of source code for the system.

We believe SL5 will greatly enhance the development of software for the micro-processor/small system of the 1980s. It is self-contained, and can be completely regenerated at any time. Subsets of the development system can be created using a minimum of 2K bytes of storage. Thus, a complete and user-oriented package can be developed at a high-level, debugged, and then implemented in EPROMs or very small memory systems.

This feature should really impact OEM's who market micro-based products and are now using assembly code. SL5 is not likely to increase memory usage, and, on large programs, is likely to need less space, due to its threaded list structure. In any case, programming-debugging-modification time will be dramatically reduced.

Since SL5 is written in SL5, it will be easy to change to a new CPU in the future. Controllers now using a Zilog Z-80 CPU could be switched to a MC6809 without huge software changes. In addition, the system is easily extensible, and can be enhanced to meet your needs at any time. Want a simpler I/O structure than CP/M? All the hooks are in the SL5 source to make the change.

All of the SL5 system can be moved to a new CPU by redoing the kernel for that CPU and its operating system I/O. As part of this manual, we have provided documentation on how the kernel works, and all of its source code is included with the system.

Where does the system go next? In large measure, that depends on where you want it to go. We are committed to both implementing the system on more CPUs/operating systems, and also adding more powerful features and enhancements. We are already working on an AMD9511-compatible Floating-Point package, and a character-string handling package. We have a MC6809 version nearly ready, and we will consider doing other standard CPUs as well. We are definitely interested in having a version for the MC68000 in the coming months.

We want to establish a broad base of users/developers who can expand the system even further. The simplicity of structure inherent in the SL5 syntax, coupled with its compact production code, fast speed, high-level coding, and in-line assembly code capability, make this system among the best available. In arriving at the final vocabulary choices, we have varied from the 1977 Forth standard only where we felt that its structure was not powerful enough, or not readable enough.

We look forward to an exciting decade for small systems, and we are confident that you have purchased a product that will allow you to keep up with the fast pace of the 1980s. We think that, as you continue to work with this powerful system, you will share this confidence.

We are available as an additional resource for programming problems, bugs, special problems, etc. Feel free to contact us at any time.

Go with it...

Mike Brothers
Larry Mongin
Dave DeLauter

Table of Contents

Getting Started

SL5 is distributed as a set of CP/M text and COMmand files on a single diskette. It is a good idea to make one or more backup copies of the distribution disk before using the system. PIP the files over to a disk which has been SYSGENed with a copy of your CP/M system. The text file SL5.DOC contains a description of each file on the diskette. If you are familiar with Forth-type languages, browse through the Reference Section list of the SL5.DOC file, and go to it. New users should read the Tutorial Section, and work through the examples first.

To bring up SL5:

1. Enter Control-S or reset the system (to bring up CP/M)
2. Enter SL5 <cr>

The prompt > should appear on your screen. If not try again, then give us a call.

Using SL5

SL5 can be used in a variety of ways, depending on your needs. The development system can be used interactively to test out simple procedures, debug a new piece of hardware, or write a simple test driver for a device. New words (procedures) can be defined, but will disappear on a reset. New words are said to be "compiled", which actually means that a symbol table entry is made, and a threaded list of pointers to the words in the definition is created for later execution. Thus, the compiled code segment is extremely compact.

Execution of SL5 words is accomplished by an inner interpreter, which fetches word addresses from the threaded list and executes each word in turn. The overhead for each word is about equivalent to a subroutine call and return. Most coding is done using the stack for parameter storage for both input and output to a word. There are also arrays, variables, and constants defined in the language. For most programs, a 256-byte stack is more than sufficient, and careful structure of code allows for a ROM-based system implementation, with very small RAM and ROM requirements.

Complex programming is usually done by writing the SL5 colon or code definitions onto a file using an editor. This file is then compiled on top of the development system, by using the word FLOAD. Each defined word can be checked for proper execution, starting from the simple definitions, and working up to the more complex words.

When the file (program) is debugged, there are several options:

1. Do nothing... That is to say: load the program each time it is used, by loading SL5, and then FLOAD the program.
2. Create a memory image of the working version of the program, that can then be saved on disk. The COMMOD procedure does this.
3. Create a RAM or ROM image of the program (with all, or part, or the SL5 system included), starting from a minimal 1K system with simple user I/O, to a 10K development system plus the working program. The SYSMAKE routine is used to create these modules. The Object Modules Section describes how these systems are created in some detail.

The SL5 system is extensible, adaptable, collapseable, and well documented, making it a very powerful, self-contained programming tool.

Tutorial

1. Introduction

This section describes how to do simple SL5 programming, and includes examples of some of the predefined SL5 procedures, called words. SL5 programs are developed by defining new words using the predefined ones. For the most effective learning, try the examples as you read. Bring up the SL5 system now, as described in the Interface Section.

The prompt character ">" on the CRT indicates that SL5 is waiting for input. Keyboard entries are terminated by a Carriage Return which, in the early examples, is indicated by the symbol <cr>. In later examples, the <cr> is omitted, but a Carriage Return is assumed after each entry. For example, the predefined word DECIMAL can be entered to tell the SL5 system that numbers entered are in base 10. Make this entry now.

```
Type: DECIMAL <cr>
```

DECIMAL and many other predefined words are described in the Reference Section, and also in the SL5 Glossary.

2. Numbers

SL5 has two primary structures, words (procedures) and numbers. Numbers are stored as 16-bit integers in memory.

```
Enter a: 2 <cr>
```

Not much seems to have happened, but the system has recognized the number 2, and stored it in the primary storage area, the "Push-Down Stack". Much like a pile of plates, the last number entered on the stack will be the first to be removed. The stack concept is the heart of the SL5 system, and will become more obvious in the examples that follow.

```
Now enter a: 3
```

The 3 is stored on the Top Of the Stack (TOS), with the 2 being pushed Next On the Stack (NOS). A period is predefined to mean "remove and display the TOS".

```
Type: .
```

This removes the TOS, and displays it (the 3, in this case) on the CRT. This leaves the value 2 remaining on the TOS.

```
Type: .
```

Now, the 2 is displayed, leaving the stack empty.

```
Type: .
```

The error message "Stack Underflow Abort" indicates an empty stack. Try entering several numbers and then displaying them with the period.

3. Simple Stack and Arithmetic Operators

Many of the predefined words use the stack for their input data, and leave their results on the stack when they finish. DUP and SWAP are two of the most used words. DUP means create a copy of the TOS and put it on the TOS (the original is now NOS). SWAP means reverse the top number with the one under it.

```
      Try: 4  6  DUP   .  .  .
and also: 4  6  SWAP  .  .
```

This same use of the stack for input and output carries over to the arithmetic operators, as

well.

```
Try:  2  3  +  .
```

"+" removes and adds the top 2 stack entries, and puts the sum back on the stack. Try some more experiments with other operators, including:

```
8  2  3  *  -  .      (result=2)
```

This notation is slightly confusing, but more than offset by the ease of defining new words, the execution speed, and the simplicity of structure.

4. Displaying a Message

```
Type: T" This is a message "
```

Anything between the T" and the " will be displayed. Note that a " will not display using this technique. The space after the T" is mandatory. Another useful word in this context is CR.

```
Try: T" line one " CR T" line two " CR
```

Try some variations of the above example.

5. Colon Definitions -- Creating New Words

New words can be defined by using the two words : ("colon") and ; ("semi-colon").

```
Type: : MESSAGE T" This is a test " CR ;
```

The predefined word : creates a symbol table entry for the word MESSAGE.

```
Now, type: MESSAGE
```

```
Try: : SUM + T" THE SUM IS " .  CR ;
```

This word SUM expects two numbers to be on the stack when it is called. It adds them, displays a message, displays the TOS, and then does a Carriage Return.

```
Type: 2   3 SUM
Type: 20 -3 SUM
```

New word definitions consist of a :, a one-word name, a list of already defined words or numbers, and a terminating ;. Fairly complex high-level words are possible, with each word being defined in terms of lower-levels, until the bottom-level words are defined completely by the predefined words supplied with the system.

Word names can be any sequence of non-blank characters. Some examples are: + 1+ Q QQQ*T THIS-IS-A-LONG-WORD and so on. Some care is necessary, to avoid confusing names. Since most programmers work in base 16, the names FF ABCD 1BAD and so on would be valid, but ambiguous with valid hex numbers. The system first checks a CRT input, to see if it is a valid word, then tries to interpret it as a number.

A complete program is usually defined as a single word which, when executed, invoke many other words to accomplish a task. The words can be tested individually, by entering them in the same manner as SUM and MESSAGE were entered. Try defining some words using either the words mentioned in this section, or in the SL5 Glossary section.

6. Program Control Words

There are several ways for SL5 programs to loop and branch. The predefined words BEGIN END IF ELSE ENDIF DO and LOOP will be discussed in this section. For a more complete list, see the SL5 Glossary section and the Reference section. The most simple loop words are the combination of BEGIN and END. END is predefined to remove the TOS and, if it is true

(not zero), terminate the loop. If the TOS is false (zero), control transfers back to the BEGIN. For example, the sequence BEGIN 0 END never ends, while the sequence BEGIN 1 END executes only once.

The words IF ELSE and ENDIF are used for most common branching. If the TOS is true (non-zero), words following an IF will be executed. If the TOS is false (zero), words following ELSE (ELSE is optional, and may be omitted) will be executed. In either case, control will transfer to the words following ENDIF.

```
Try: : ATEST IF T" true " ELSE T" false " ENDIF MESSAGE CR ;
```

Note: The SL5 system allows use of some words, such as BEGIN END IF ELSE ENDIF and so on, only within a colon definition. Attempts to use them otherwise will leave the stack in an unknown state.

```
Try: 1 ATEST
and: 0 ATEST
```

Most languages have some built-in procedures to do common kinds of branching such as DO and CASE statements. SL5 has these procedures predefined, and a more complete description can be found in the Reference section. Here is an example of a simple DO statement to display the numbers 0 to 4.

```
Type: : DOTEST 5 0 DO I .  LOOP CR ;
```

The predefined word I puts the inside-most loop counter on the TOS.

```
Type: DOTEST
```

The word DO expects two values on the stack: the TOS is the start value for the loop, and the stop value less 1 is NOS. LOOP increments the count by 1, and continues the loop until the stop value is reached.

```
Now, type: : DOTEST 0 DO I .  LOOP CR ;
```

The system responds with a message "REDEF DOTEST", indicating that the word DOTEST has been redefined. The first definition is still stored in memory, but future references to DOTEST will use the new one. This second version requires a stop value to be on the TOS when it is invoked.

```
Type: 7 DOTEST
```

For a still more general DOTEST, put both DO parameters on the stack.

```
Type: : DOTEST DO I .  LOOP CR ;
```

Once again, DOTEST is redefined, and there are now three versions. The latest one will be executed.

```
Type: 8 1 DOTEST
```

Try some more experiments. There is a convenient predefined word, FORGET, to allow returning to an old definition. Caution: FORGET will throw away all word definitions until it reaches the word specified.

```
Type: FORGET DOTEST
```

The second version of DOTEST is now active. Test it...

```
Then type: FORGET ATEST
```

This causes the system to forget ATEST and all words defined since ATEST, in this case DOTEST versions two and one. Test it.

```
Type: DOTEST
```

The error message "DOTEST ?" indicates that DOTEST is unknown to the system. ATEST is

gone, too. FORGET is most useful during the debugging phase of program development. Each new load is preceded by a FORGET of the old version.

7. Constants, Variables

In complex programs, it is convenient to have access to storage areas other than the stack for commonly-used data. In SL5, there are four predefined words for this purpose: CONSTANT VARIABLE ARRAY and BARRAY (byte array).

The word CONSTANT defines a name which, when executed, will leave a 16-bit value on the stack. Usually, this value remains unchanged during execution, and it is considered as a ROMable memory area. By convention, values that might be changed are stored in variables. The word VARIABLE defines a name which leaves the 16-bit address of a 16-bit value on the stack.

The predefined word @ ("fetch") removes the TOS, assumes it to be an address, and leaves the data from that address on the TOS. The predefined word ! ("store") removes an address (TOS) and a data value (NOS) from the stack, and stores the data in the location specified by the address.

```
Type: 1  CONSTANT  ONE
and:  2  VARIABLE  VARTEMP
and:  VARTEMP  @  .
```

to display the current value of VARTEMP.

```
and:  ONE  .
```

to display the value of ONE.

```
Try: 6  VARTEMP  !
and: VARTEMP  @  .
and: VARTEMP  @  ONE  SUM
```

Try defining some variables, and experiment with them. Note that the use of @ and ! is not limited to variables. If you wish to read the contents of memory location zero, 0 @ will leave that value on the stack. For example,

```
: ZAP 100 0 DO 0 I B! LOOP ;
```

would set memory locations from 0000H to 0099H to zero upon execution of ZAP. The words B@ and B! fetch and store bytes. (Warning: Setting low-memory cells to zero is likely to crash most Operating Systems.)

8. Writing a Program -- A Simple Example

A very short sample program to do inventory control is outlined below. The top-level word ICP calls a word to initialize variables, tables, and so on. Then, it begins a loop which looks for a CRT entry, does the appropriate action, and continues to loop until a termination is requested. The top line enclosed by parenthesis is a comment.

Once the top-level word is completely defined, each of its words can be defined, either as a test stub, or in a more complete form. This process continues until all words are defined. As each bottom-level word gets defined, it can be checked out immediately. By the time the checkout reaches the top-most levels, most bugs are fixed.

```
( ICP Inventory Control Program )
  :  ICP  INITIALIZE  BEGIN
       DISPLAYMENU  ANYINPUTYET  DUP  IF
         PROCESSOPTION
       ENDIF
     END  ;
```

This definition leaves INITIALIZE, DISPLAYMENU, ANYINPUTYET, and PROCESSOPTION yet to be defined. Suppose we define DISPLAYMENU as:

```
 :  DISPLAYMENU                          CR
       T" Select an option: "            CR
       T"   1    New Entry  "            CR
       T"   2    Update Previous Entry "  CR
       T"   3    Display Current Data  "  CR
       T"   4    Stop Execution        "  CR  ;
```

This definition could be checked immediately, as all of its words are already defined. The word ANYINPUTYET, as used in the definition of ICP, will check to see if there is a keyboard entry, and leave the data entered on the TOS, or a 0 if there is no entry yet. The entry is duplicated and checked with the IF, and, if it is a zero (false), the duplicate will cause END to branch to BEGIN. The predefined word CIN might be used in ANYINPUTYET.

The word PROCESSOPTION is the heart of ICP. It must do the file I/O and displays as requested, and leaves a true (non-zero) value on the stack for stopping the program, if requested. The predefined words CASE and NOCASE could be used very effectively in PROCESSOPTION. During the development stage of ICP, this word might be defined as a stub for testing, and then filled in, piece by piece, later. For example:

```
: PROCESSOPTION  ( Test stub for debugging.  )
       T" PROC " DUP .  4 = IF
                 1 ELSE 0  ( Leave a 1 for terminate.  )
       ENDIF  ;
```

Careful choice of word names, a few comments, and some sort of indentation convention will make programs almost self-documenting and easy to change. Avoid the temptation to save space by using obscure names or writing the definitions run together, i.e.

```
: ICP IIN BEGIN INP DUP IF PROC ENDIF END ;
```

This definition is not very easy to read, and not recommended.

There are many ways to write ICP that are equivalent to the above example. Avoid doing clever and tricky stack manipulations, or other shortcuts, until the program is debugged (if at all). SL5 programs are fast and may be fine without further tweaking. Even faster execution speed can be obtained by substituting CODE definitions (see the Assembler section) for a few key words.

During normal program development, SL5 programs are written directly on disk files using an editor, and the loaded using the predefined word FLOAD. See the Reference section for more information about loading. The SYSMAKE program included with SL5 can be used to create a CP/M COMmand file or a standalone ROMable module. See the SYSMAKE section for more details about these options.

This concludes the Tutorial section. Many other examples of SL5 programs can be found in other sections of this manual. Perhaps the best resource for SL5 examples is SL5 itself. Most of the system is written in SL5, and all source code is included with the system.

Reference

1. Introduction

This section describes the SL5 programming language. It is loosely organized by class of operator, and is intended to bridge the gap between the Tutorial section and the SL5 Glossary section. Two central elements of the SL5 programming system are a push-down stack and RPN (Reverse Polish Notation) logic. Programmers unfamiliar with either of these concepts are urged to review the material in the Tutorial section, and experiment with the system, using the console interpreter. Many examples in this section illustrates the use of SL5 operators from the console interpreter (see Example below). Each line of input is terminated with a Carriage Return. Numbers are pushed onto the stack for use by operators that follow. Dot/period (".") causes the top of the stack to be printed on the CRT. The right arrow (">") is a system prompt for more input. All numbers in the examples are decimal unless otherwise specified.

Example:

```
>3 4 5  ( Put 3 numbers on the stack.  )
>...    ( Print 3 stack elements
5 4 3 >4 3 + 2 * .
14 >    ( Result is 14 if base is decimal.  )
```

2. The Stack

The stack is the primary mechanism for data transfer in SL5. Operands are pushed onto the stack for subsequent processing by words (procedures) invoked from the outer interpreter, or executed from compiled words. RPN (Reverse Polish Notation) logic holds between words, although infix is used to order multiple operand words (e.g., +, -, ROT). The stack is formally defined as a last-in first-out queue. Use of the stack to transfer parameters between words allows the natural generation of re-entrant and recursive code, and clean handling of interrupts.

```
>5 .
5 >5 DUMP . .
5 5 >10 5 SWAP . .
10 5 >3 4 5 ROT . . .
3 5 4 >3 4 2DUP . . . .
4 3 4 3 >
```

Stack Operators

The table below describes the SL5 stack operators. The Top Of Stack is on the right in the diagrams.

Operand	Function	Stack before	after
DUP	Copy TOS	a	aa
DROP	Remove TOS	ab	a
SWAP	Reverse TOS, TOS-1	ab	ba
OVER	Copy TOS-1	ab	aba
ROT	Move TOS-2 to TOS	abc	bca
n ROLL 3 ROLL	TOS-n to TOS Fill vacated positions (Same as ROT)	abc	bca
n -ROLL 3 -ROLL	Opposite of ROLL	abc	cab
n PICK 3 PICK	Copy TOS-n to TOS	abc	abca
2DUP	Dup TOS and TOS-1	ab	abab
2DROP	Drop TOS and TOS-1	aab	a
2SWAP	Swap TOS,TOS-1 with TOS-2,TOS-3	abcd	cdab

3. Numbers

Signed numbers are stored as 15-bit integers, with the top bit used for the sign. A 16-bit word is also used to store unsigned numbers. There is no internal difference between signed and unsigned numbers. The type of operator determines how the number is handled. Overflow is not checked by runtime routines. The usual result is truncation.

BASE (Radix)

The variable BASE contains the current Radix used by input and output routines. Changing the base does not affect numbers already stored in memory, only the Radix with which the number is printed. DECIMAL, OCTAL, and HEX are predefined words to set the Radix. Other bases may be used, but the output may look strange.

```
>DECIMAL 10 HEX .
A >HEX 10 DECIMAL .
16 >
```

4. Variables and Constants

Constants

A constant is a word that pushes its value on the stack when executed. Constants are stored as 16-bit integers.

```
>10 CONSTANT XX
>XX .
10 >
```

The value of a constant can be changed by manipulating the dictionary and code segment, but that is a poor programming practice in RAM-based systems. The value of a constant cannot be changed in ROM-based code because the value is stored in the ROM section of memory. Use variables if you want to change the value at runtime.

Variables

A variable is a word that pushes the address of a 16-bit memory location on the stack. SL5 has 2 types of variables, RAM and ROM. Variables created during normal program development are RAM variables. The variable is stored in the code segment. A flag variable, ROMF, determines whether a system is created with ROM (ROMF=1) or RAM (ROMF=0) variables by the SYSMAKE program. When ROMF=1, the code segment contains a pointer to a memory location in a user-defined RAM area.

```
>10 VARIABLE XX
>XX @ .
10 >15 XX !
>XX @ .
15 >
```

Memory Operators

Memory words assume that an address is on the TOS. An additional operand may be required (e.g., !).

Operator	Example	Function
@	q @	TOS = contents of address q
!	m p !	Store m at address p
B@	q B@	Fetch 1 byte from memory
B!	m p B!	Store byte m at address p
@X	q @X	Fetch 1 word, and swap bytes
1+!	p 1+!	Increment contents of p by 1
1-!	p 1-!	Decrement contents of p by 1
+!	m p +!	Add m to contents of p

```
>10 VARIABLE TEST
>TEST .
4000 >  ( Address of TEST returned.  )
>TEST @ .
10 >5 TEST !
>TEST @ .
5 >TEST 1+!
>TEST @ .
6 >
```

5. Arithmetic and Logical Operators

All of the arithmetic and logical operators take their operands from the stack, and return the result to the stack. Logic is RPN, so precedence is implied by the order.

Operator	Example	Function
+	m n +	Add m and n, result on TOS
1+	m 1+	Add 1 to TOS
-	m n -	Subtract n from m (m-n)
1-	m 1-	Subtract 1 from TOS
--	m n --	Subtract - (m-n) = (n-m)
*	m n *	Multiply n by m
/	m n /	Integer divide m by n Quotient on TOS
/MOD	m n /MOD	Integer divide m by n Quotient on TOS Remainder on TOS-1
MOD	m n MOD	Remainder of m/n on TOS
MIN	m n MIN	Leaves smaller of m,n on TOS
MAX	m n MAX	Leaves larger of m,n on TOS
ABS	m ABS	Leaves absolute value on TOS
MINUS	m MINUS	Negate by taking 2's complement
COM	m COM	1's complement
&	m n &	16-bit logical AND
|	m n |	16-bit inclusive OR
X|	m n X|	16-bit eXclusive OR
<-L	m n <-L	Left shift m n bits, end off
->L	m n ->L	Right shift m n bits, end off

Examples: Arithmetic and logical operators

```
>5 3 + .
8 >5 1+ .
6 >5 3 * .
15 >1 0 & .
0 >1 1 & .
1 >0 1 X| .
1 >0 0 X| .
1 >1 1 | .
1 >3 5 + 2 * .
16 >0 1 | 0 & .
0 >
```

6. Conditional Operators

Conditionals take their operands from the stack, and return logical true or logical false as the result on the Top Of the Stack. In SL5, a logical true is defined as a non-zero value on TOS. A logical false is a zero TOS. Four unsigned operators are included, so that 16-bit numbers like addresses can be compared.

Operator	Example	Function
0=	m 0=	TRUE if m = 0
0<	m 0<	TRUE if m less than 0
0>	m 0>	TRUE if m greater than 0
NOT	m NOT	Equivalent to 0=
=	m n =	TRUE if m = n
<>	m n <>	TRUE if m not equal to n
<	m n <	TRUE if m less than n
>	m n >	TRUE if m greater than n
<=	m n <=	TRUE if m LT or equal to n
>=	m n >=	TRUE if m GT or equal to n
U>	m n U>	Unsigned greater than test
U<	m n U<	Unsigned less than test
U>=	m n U>=	Unsigned GT or equal test
U<=	m n U<=	Unsigned LEt or equal test

Examples: Conditional operators

```
>1 0= .
0 >0 0= .
1 >0 0< .
0 >5 3 = .
0 >5 5 = .
1 >5 5 <> .
0 >
```

7. The Outer Interpreter

The outer interpreter is the 'main-loop' of the SL5 programming system. It functions as the console executive, interpreter, and primary data entry system. This seven word colon definition is a classic example of the power of the Forth programming language.

The SL5 visible to the user on the CRT screen is the outer interpreter. A line of text is collected, then processed. Colon definitions, variables, and constants are entered into the dictionary -- which is a liked list of defined words. Previously-defined words that are not part of a new definition are executed immediately. Strings not found in the dictionary are treated as data. NUMBER is called to convert the ASCII string according to the current radix. If NUMBER fails, an undefined abort occurs. Valid numbers are pushed onto the stack if executing, or entered into the dictionary as literals if compiling.

Compilation is a special case of interpreting (see Section 8 : Colon definitions). Colon sets the system variable STATE to compile. Subsequent words in the input stream are compiled until the word ; resets STATE. A few words (e.g., ', T", IF, etc.) are executed even though SL5 is in the compile state. These are called immediate words or compiler directives.

8. Colon Definitions

Compiling is a special case of interpreting the input stream. When a : is encountered, the system variable STATE is set. The words that follow up to a ; are used to create a new word in the dictionary.

There is a class of words that executes during compilation. They are compiler directives, or words that act on the input stream (', T"). Tokens in the input stream that are not found in the

dictionary are converted to numbers, and stored as literals in the dictionary. A conversion failure results in a call to UNDEFINED.

Colon creates an entry in the symbol table, with a pointer to the first entry for that word in the code segment. The start of every colon definition is a call to the word $: to set up a context switch. The rest of the definition is a list of addresses and literal values. The addresses are pointers to the entry points of other words. The last entry is the address of $; which restores the context.

Example:

```
: 2TIMES 2 * ;

+----------------+
| $:             |
| Address of LIT |
| 0002           |
| Address of *   |
| Address of $;  |
+----------------+
```

Semicolon puts the address of $; in the dictionary, and resets STATE. SL5 resumes executing until another colon definition is found. A colon definition is a list of addresses of other words. Each word is executed, until $; flags the context-restore process.

The Inner Interpreter

NEXT is an SL5 code word that processes the list of addresses in a colon definition. It maintains an interpreter pointer (IP). The IP is similar to the PC ("Program Counter") in the CPU. NEXT is called at the end of every code word. Machine code sequences are executed directly by the CPU as assembly language routines. NEXT provides the linkage between words. A SL5 program can be viewed as an inverted pyramid, with machine code primitives at the bottom, and levels of SL5 words above. $: and $; handle the context switching between colon definitions, and NEXT links the words.

9. Branching

Conditional branching in SL5 is done with the IF..ENDIF and CASE..CASEND structures. These words generate test and branch instructions in compiled SL5 words. They can only be used inside of colon definitions. They are not defined in the context of interpreted code.

IF..ELSE..ENDIF

The words IF, ELSE, and ENDIF are compiler into test and branch instructions. In the compiled code, $IF tests the Top Of Stack. A TRUE TOS (<> 0) causes the words following IF to be executed. A FALSE TOS (=0) causes a jump over the words following IF to the words following ELSE, or to ENDIF if no ELSE clause is present.

```
>: TEST IF 1 .  ELSE 0 .  ENDIF ;
>0 TEST
0 >1 TEST
1 >FF TEST
1 >
```

IF statements may be nested several levels deep, but the clauses must be balanced. Every IF must have a ENDIF statement to close the structure.

```
X @ IF
    Y @ IF
        X @ Y @ / Z !
    ENDIF
ENDIF
```

CASE..NOCASE..CASEND

SL5 uses a CASE statement similar to the PASCAL CASE statement. This deviates from 'standard' Forth, but it is a much more readable structure. Any CASE structure must have a logical equivalent to a nested IF structure, but it does not need to be visible at the source code level. The verb CASE causes the contents of the TOS to be compared with test values specified in the CASE body. If a match is found, that element is executed, and the interpreter pointer IP is set to the word after CASEND. The NOCASE clause, if present, is executed when the TOS does not match any of the test values.

Example: Keyboard input

```
: KEYIN  ( 0DH = CR, 08H = BackSpace, 1BH = ESCape )
    GCH CASE
          0D =: 1 KRDY ! TBUF @ TBUFMAX ! CR  ;;
          08 =: 20 TCH 08 TCH 20 TCH TBUF 1-! ;;
          1B =: TBSTRT @ TBUF ! T" *ESC* " CR ;;
    DUP NOCASE =: DUP TCH TBUF @ ! TBUF 1+!   ;;
CASEND ;
```

10. Loops

Iteration operators are a central part of a high-level programming language. SL5 has three types of structures for repetitive execution: BEGIN..END, DO..LOOP, and RECURSE. BEGIN..END is repeat until test conditions is TRUE. DO..LOOP is repeat n times, and RECURSE is a structure to allow a word to call itself.

BEGIN..END

BEGIN is a compiler word that pushes the contents of the interpreter pointer IP on the stack. Words between BEGIN and END are compiled. END causes a test instruction with conditional branch to be compiled into the word. If the TOS is TRUE (<>0), the word following END is executed next. A FALSE TOS (=0) results in a jump back to the word following BEGIN, and the loop is repeated.

```
: WAIT-FOR-CR BEGIN GCH 0D = END ;
```

This loop will repeat until a CR is typed.

BEGIN..WHILE..REPEAT

The BEGIN..END structure has a test at the end of the loop. BEGIN..WHILE..REPEAT tests for iteration at the beginning of the loop. WHILE generates code to test the TOS, and jump to the word following REPEAT if TOS is FALSE (<>0). REPEAT generates an unconditional jump back to BEGIN, and the TOS is tested again.

DO..LOOP

The D..LOOP is similar to the same structure in PASCAL or FORTRAN. The body of the loop is repeated n times. DO takes 2 parameters: limit+1 and start. Note that the first parameter is 1 greater than the limit.

```
>: DO-TEST 10 0 DO I .  LOOP ;
>DO-TEST
0 1 2 3 4 5 6 7 8 9 >
```

+LOOP, EXIT

The DO..LOOP was shown incrementing the loop counter by 1 on each iteration. +LOOP allows the counter to be incremented by any positive integer. It does not have to be an even multiple of the total count.

```
>: DO-TEST 10 0 DO I .  3 +LOOP ;
>DO-TEST
0 3 6 9 >
```

The word EXIT causes termination of a DO..LOOP at the end of the current iteration. It does not cause the words between EXIT and LOOP to be skipped, though.

```
: CRT-OUT
     TBUFMAX 1+ TBSTRT DO  ( Loop from start to max )
        I @ DUP 0D = IF
           DROP CR EXIT  ( Found end of line )
           ELSE TCH
        ENDIF
        LOOP ;
```

Loop indexes (I,J,K)

I, J, and K are words that push the current loop index on the stack. I is the index of the innermost loop, J and K are indexes for the next 2 loops. DO..LOOPs can be nested many levels deep, but indexes are only provided for the three inner loops. See the Structure section for DO..LOOP implementation details if you need an index on another level.

```
: MATRIX
     3 0 DO
        2 0 DO
           2 J * I + .
        LOOP CR
     LOOP ;

>MATRIX
0 1
2 3
4 5
>
```

RECURSE

SL5 is a stack language. Re-entrant and recursive code is a natural byproduct of the language. Normally, a word must be compiled before it can be used. RECURSE puts the address of the word being compiled into the dictionary, thus generating recursive code. SL5 has a fixed-length stack (usually 256 bytes). See the SYSMAKE section for details on how to increase the stack size.

```
: FIB
     DUP 60 < IF
        DUP .  SWAP OVER + RECURSE
        ELSE 2DROP
     ENDIF ;

>0 1 FIB
1 1 2 3 5 8 13 21 34 55 >
```

11. Arrays

The SL5 vocabulary has one-dimensional word and byte arrays. The index is 0-based; a 10 element array has indexes of 0 to 9. Both RAM and ROM arrays are available. The storage for RAM arrays is located in the code segment, along with the defining code. The storage for a ROM array is put in a user-defined memory area, along with ROM variables. The code segment for a ROM array contains a pointer to the first element of the array in memory.

ARRAY

```
100 ARRAY BUFFER
```

A 100 element array of 16-bit words is allocated, and a word, BUFFER, defined in the current vocabulary. When an array word is referenced, the sum of the TOS and the base address of the array is put on the stack.

```
( Assumes BUFFER is at address 3000H.  )
>10 BUFFER
```

```
3010 >10 BUFFER @ .
nnnn >  ( where nnnn is contents of 11th word.  )
```

BARRAY

BARRAY functions like ARRAY, except an element is a byte instead of a word. 100 BARRAY VEC1 allocates 100 8-bit bytes of storage. Word operations like @, instead of B@, can be used on a BARRAY. The address returned is the first byte used. The second byte is computed by the operation.

```
>100 BARRAY VEC1
>10 1 VEC1 B!
>1 VEC1 B@ .
10 >
```

12. I/O

Introduction

The SL5 programming system interfaces to a number of commonly available disk operating systems. In general, the specific operating system is transparent to the user. Exceptions are noted in the Interface section. Serial and disk I/O in SL5 is performed through a uniform library of procedures. The code is designed to be symmetric, so that the specific device is transparent to most routines. Files and devices are assigned to channels using NAMIT. All I/O, except serial output, is buffered. The device driver routines are embedded in logical record routines called by character I/O words.

INFILE, OUTFILE

There are 2 default channels or data streams defined. Associated with INFILE and OUTFILE are SL5 control bytes, logical record buffer, and an operating system-dependent control area. At coldstart, INFILE and OUTFILE are initialized to the console device.

FALLOC, NAMIT

Additional I/O channels can be allocated with FALLOC. FALLOC reserves space in memory for buffers and control fields, and assigns a name to the area.

```
FALLOC INFILE
FALLOC FILE1
```

SL5 has a FALLOC INFILE to set up that channel. FALLOC FILE1 would set up another channel addressed by the name FILE1. Most of the I/O library can be used with additional channels. GCH, TCH, T", CIN, COUT, C" can only be used with the default data streams INFILE and OUTFILE.

NAMIT links a filename or a device to a channel. Subsequent calls to OPENR or OPENW establish the link between the channel and the operating system.

```
INFILE NAMIT test.txt
INFILE NAMIT #CRT
OUTFILE NAMIT #CRT
OUTFILE NAMIT #LIST
```

OPENR, OPENW, FLUSH, CLOSE

OPENR enables a channel for input. If the name is a disk file, the open command is passed on to the operating system. A serial device is opened by setting a parameter in the control field for the channel. OPENW opens a channel for output. SL5 does not permit concurrent reads and writes on the same channel. FLUSH writes any bytes left in the buffer out to the file or device. CLOSE resets the channel status, and issues a call to the operating system that updates the directory for the file associated with the channel. FLUSH and CLOSE should always be invoked before attempting to re-use a channel.

GCH, TCH, T", TYPE

GCH reads the next character from the default input channel INFILE. Buffering of logical records is handled by the I/O system. The CP/M EOF character 1AH is returned on end of file. This character is normally Control-Z on the keyboard. TCH transmits a character to the default output channel OUTFILE. T" transmits a string of characters to the output channel OUTFILE. One space must follow T" before the string starts. " terminates the string, and one space must precede the ". TYPE outputs a string to the output device. nnnn TYPE causes the string at address nnnn to be output. The first byte of the string contains the string length.

```
>T" This is a message.  " CR
This is a message.
>
```

CIN, COUT, C"

CIN, COUT, C" is character in / output streams to the console device and can only be used with the default data streams INFILE and OUTFILE.

CIN Read a character from the keyboard, and return its ASCII value. The character is checked before returning it, to see if it is one of the special characters.

c COUT the character whose ASCII value corresponds to c is displayed on the console.

C" Output a string to the console. One space must precede the string.

RCH, WCH

RCH Read a ASCII character from the selected channel. (see READ)

WCH Write a ASCII character to the selected channel. (see WRITE)

RBYTE, WBYTE

RBYTE Read a byte from the the selected channel. (see READ)

WBYTE Write a byte onto the selected channel. (see WRITE)

READ, WRITE

READ reads a physical sector from the selected channel into a system buffer. Disk sectors are 128 or 256 bytes, depending on the operating system. A line is treated as a physical record if a serial device is attached to the channel.

WRITE writes one physical record to the file or device attached to the selected channel. The user should be familiar with disk I/O conventions and the host operating system before calling READ and WRITE directly. The SL5 character-level words (GCH, RCH, RBYTE, etc.) call READ and WRITE.

DELETE, RENAME

Channel-name DELETE removes the file associated with the channel from the operating system directory, and returns the space occupied by the file. Channel-name NAMIT must be invoked before DELETE to set the filename. Use old new RENAME to rename old to new.

ININIT, OUTINIT

ININIT and OUTINIT initialize the default input and output channels (INFILE and OUTFILE) to the console device. These words are called on cold-start. If used to re-assign the channel, FLUSH and CLOSE should be called first, to empty the buffer and update the directory, respectively.

EOF

The word EOF returns a 0 (=FALSE) while there is more data in the file, and a 1 (=TRUE) when the physical end of file is reached. The character I/O words also return the CP/M end of

file character 1AH (Control-Z). Continuing to read characters from a channel after the end of file is reached results in a fatal error, and a system restart.

WORD

The SL5 word WORD scans the current system input buffer, and gets the next token from the buffer. The variable DELIMITER contains the character used to separate strings of characters. The user can set DELIMITER before calling WORD. WORD resets DELIMITER to space (20H) before returning. WORD puts the string length and string of the next token at the end of the dictionary. HERE points to the length, and HERE 1+ is the location of the first character in the string.

Example: Get a filename from the console:

```
: GET-NAME
     T" Enter filename (1-8 chars) " CR
     WORD
     T" The current file is "
     HERE @ 1+ 0 DO
        HERE 1+ I + B@ DUP TCH I NAME B! LOOP CR ;

>GET-NAME
Enter filename (1-8 chars)
SL5
The current file is SL5
>
```

Numeric Input

NUMBER can be used, after WORD is called, to convert the string WORD leaves in the dictionary to a number. The radix is defined by BASE (see Section 3: Numbers). If the number input is larger than the maximum value that can be stored in 16 bits, high-order digits are lost. AFFFF converted as a HEX number results in FFFF returned by NUMBER.

Numeric Output

The Forth vocabulary has several words to print numbers on the output device. The word . (dot/period) prints the top of the stack as a signed integer. It is printed according to the current radix stored in BASE.

```
>10 .
10 >-3 .
-3 >HEX 100 DECIMAL .  HEX
256 >
```

X. prints the top of the stack as an unsigned 16-bit integer. It is used for addresses and other numbers that are stored as 16-bit unsigned numbers, instead of 15-bit signed numbers. B. prints a byte (8 bits), instead of a 16-bit word. B. is used in the dump routine.

```
>AAAA X.
AAAA >88 B.
88 >9988 B.
88 >  ( Note that the high-order byte is lost.  )
```

FLOAD, [END-OF-FILE]

FLOAD loads an SL5 source file. The input stream is switched from the console device to the disk file specified after FLOAD. The text is interpreted and compiled into the dictionary. A physical end of file switches input back to the console. The word [END-OF-FILE] can be used to create a logical end of file at any place in the source text. This can be helpful when one wishes to debug part of a program.

13. User-defined Code Structures

BARRAY

Same as ARRAY, with the exception that the cells are one byte in length, instead of two bytes.

Example: BARRAY

```
: BARRAY HERE 5 + CONSTANT DP+! ;: @ + ;

>20 BARRAY TEST
>100 1 TEST B!
>1 TEST B@ .
100 >
```

BARRAY is a colon definition that creates byte arrays in memory. 20 BARRAY TEST causes a 20-byte array named TEST to be set up. Subsequent references to TEST invoke the code between ;: and ; in BARRAY.

;CODE

The ;: word creates new classes of structures that execute words. Occasionally, the new word class must be faster than colon definitions allow. ;CODE creates new words that invoke assembly language sequences (see Assembly Language section).

14. The Dictionary

Symbol Table

SL5 has a symbol table separate from the code segment. The symbol table is a linked list of the symbols defined in vocabularies, a flag byte, and a pointer to the first word of the code body in the code segment. The symbol table links are relative, so the table can be moved. It grows down toward the code segment.

Vocabularies

A vocabulary is a set of words linked together. System vocabularies include FORTH, ASSEMBLER, DEBUG, and SYSMAKE. A vocabulary is created by the word VOCABULARY. Words are added to it with DEFINITIONS. Invoking the vocabulary name causes it to be the context vocabulary, or the vocabulary searched by FIND.

```
VOCABULARY ALFA defines the SL5 vocabulary
ALFA -- ALFA is the context vocabulary.
ALFA DEFINITIONS -- Declare SL5 as the current vocabulary
New definitions are added to SL5.
```

Two system variables, CONTEXT and CURRENT, point to the heads of vocabularies. CONTEXT points to the last vocabulary invoked. It is used by FIND for dictionary searches. CURRENT points to the vocabulary that is being added to when new words are defined. The two pointers can point to the same vocabulary as when a user vocabulary is being built.

Chaining

Vocabularies can be chained together, to extend the scope of the dictionary search. FORTH is always the base vocabulary. Other vocabularies are chained to FORTH (e.g., ASSEMBLER, EDITOR). This chain can be extended into tree structures, to develop separate sections of programs, or limit the length of a dictionary search. The chain links through vocabulary heads, so words defined in a vocabulary after it is chained are still within the scope of the search.

FIND

The SL5 word FIND searches the symbol table for a match with the string that WORD left at the end of the dictionary. It links through chained vocabularies, searching from the last word defined. FIND returns FALSE (=0) if the search fails. If a word is multiply-defined in the dictionary, the last definition is found.

FORGET

FORGET nnnn erase all words in a vocabulary from the word nnnn. This is a sequential erase. The vocabulary head is set to the word before nnnn, if any.

' ,'B, 'S, COMPILE

These words return a pointer to the code segment of a word. ' nnnn returns the address of the first word in a colon definition, or the parameter field of a variable or constant. It can be used to change the value of a constant, or the address of a variable, but the practice is not recommended. 'B returns the address of the code field of a word. 'S is the same as 'B during system generation. COMPILE is 'B ,]. It gets the address of the entry point of a word, and compiles it into the dictionary. All of these words can be used to produce tricky, obscure code. It is recommended that their use be restricted to the few systems programming applications where they are really needed. See the source of the SL5 kernel for examples of their use.

15. Code Words

A code word is an assembly language routine with a symbol table entry. It is similar to an assembly language subroutine, except that the last instruction is a jump to NEXT, the inner interpreter, instead of a return instruction. Many of the words defined in the Forth vocabulary are code words.

A nice strategy for program development is to write an entire application in SL5 colon definitions, debug it, and then recode a few words as code words where speed requirements and frequency of use dictate. Colon definitions have a fair amount of overhead. NEXT is called between every word in the definition. Also, $: is called to switch context on entry, and $; is invoked to switch context on exit from the colon definitions. This overhead ranges from 100 microseconds on an Intel 8080 system, to a few microseconds on a MC68000 system. Code words are difficult to write, debug, modify, and they are not machine-independent (see Assembly Language section).

16. System Variables

The SL5 programming system has several variables that contain important information for the operating system. System variables can be accessed and changed by any word. The user must understand the SL5 system before modifying system variables. The table below describes system variables.

Variable	Function
DP	Current dictionary pointer
CURRENT	Head of current dictionary
CONTEXT	Points to context vocabulary
CVOC	Current vocabulary pointer
SYMTP	Top of symbol table
SYMPTR	Last entry in symbol table
RSIZE	Return stack size (constant)
SSIZE	Data stack size (constant)
RESTARTAD	Address of RESTART (SYSMAKE)
GOQIAD	Address of outer interpreter (SYSMAKE)
STATE	0=interpret, 1=compile
BASE	Number radix
UPPER	Upper/lower=0, upper only=1
DELIMITER	Delimiter character used by WORD
SSTACK	Parameter stack (a BARRAY)
RSTACK	Return stack (a BARRAY)
INFOF	Information control byte: Bit 0=1 -- REDEF message printed Bit 2=1 -- Print source during FLOAD All bits defaults to 1.

17. Error Messages

During the execution or compilation of programs, a variety of error conditions are checked for. These can be classified into 3 groups: (1) those which are informative, (2) general fatal conditions, and (3) fatal conditions in the file system. When an error of type 2 or 3 occurs, the word RESTART is executed after the message is displayed.

Informative messages (type 1)

REDEF nnnn	The word nnnn was just redefined, with the previous definition now being inaccessible. This message can be turned off by setting bit 0 of the variable INFOF to 0.

General fatal error messages (type 2)

nnnn ?	The word nnnn could not be executed or compiled because it is not defined.
D/0 ABORT	The previous division (/, /MOD, MOD, U/MOD) operation was undefined (division by zero).
STACK UNDERFLOW ABORT	The parameter stack is in an underflow state, i.e., more items were removed than there were placed on the stack.
RETURN STACK UNDERFLOW ABORT	The return stack is in an underflow state.
UNBALANCED NESTING ABORT	The word just defined did not contain proper balancing of IF..ENDIF, BEING..END, DO..LOOP, CASE..CASEND constructions.

Fatal error messages from the file system (type 3)

READ PAST EOF	An attempt to read beyond the end of a file just failed.
FILE NOT OPENED FOR READING	A request to read (RCH, RBYTE, or READ) from a file not opened for reading failed.
FILE NOT OPENED FOR WRITING	A request to write (WCH, WBYTE, or WRITE) onto a file not opened for writing failed.
FILE DOESN'T EXIST	The file specified on the last OPENR doesn't exist in the file system. Files being opened via OPENR have to be previously created (OPENW does this).
FILE CAN'T BE CREATED	An attempt to create a new file via the OPENW routine failed, due to a lack of disk space or directory space.
DISK WRITE ERROR	The previous WRITE command failed because of a system error.

Assembler

1. Introduction

In the previous sections, you were shown how to define a process in terms of other, previously defined processes or words, and this was a "colon definition". There is another method of defining words in the machine language of the particular processor being used. This is called a "code definition". Many of the SL5 primitives (such as SWAP, +, *, etc.) are defined in this manner.

Note: This manual assumes that the reader is familiar with the Zilog Z-80 assembly language.

2. Execution of CODE definitions

The execution of code definitions is very different from the way in which colon definitions are executed. The main difference is that the body of the code word is executed directly by the CPU, while colon bodies are interpreted by the inner interpreter.

3. Creation of CODE Words

One of the major differences between CODE words and colon definitions is that the code segment is created through the execution of words which place machine instructions in the code segment while, with colon definitions, addresses are placed in the code segment by the compiler.

```
CODE nnnn ...  words ...  EDOC
```

Figure 3-1. Format of CODE definitions

Shown in Figure 3-1 is the format of CODE definitions. Upon execution of the word CODE, nnnn is added to the dictionary, and CONTEXT is set to the assembler. The words which follow may place machine instructions in the code segment, which will be executed when "nnnn" is subsequently executed. The final part of a CODE definition (EDOC) resets the CONTEXT vocabulary to the CURRENT vocabulary, so that the newly-defined word may be used immediately.

3.1. Using the Assembler

During the writing of CODE definitions, the machine instructions which compose the definition can be specified by placing the actual opcodes in the code segment, or by using the SL5 assembler mnemonics. To place the opcodes in the code segment, the two words, and B, may be used. The word "," will store a word (16 bits) in the code area, while "B," will store a byte (the low-order 8 bits of a word).

Note: The word "," reverses the bytes before storing them.

Example: The word "+" could be defined as:

```
CODE +  C1 B,  E1 B,  09 B,  C3 B,  $PUSH ,  EDOC
```

C1, E1, and 09 are the Zilog Z-80 opcodes which, when executed, will pop the BC and HL registers off of the stack, and add them together. The "C3 $PUSH" will then jump to the inner interpreter, and push the sum which is in HL onto the stack. The memory image of the word "+" is shown below.

```
                +----------------------+
                | JP $PUSH  (C3 $PUSH) |
                | ADD HL,BC       (09) |
                | POP HL          (E1) |
Entry point -->| POP BC          (C1) |
                +----------------------+
```

Another, more readable, method of assembling opcodes is by using the mnemonic assembler. To use the mnemonic assembler, you must first load it (the mnemonic assembler occupies approximately 4000 decimal bytes):

```
>FLOAD ASSEM.SL5 <cr>
```

You can now define CODE words by using the Zilog Z-80 mnemonics to specify machine instructions (see Appendix A for further details on the mnemonics).

Example: The definition of the word "+" can now be written as:

```
CODE + BC POP   HL POP   BC HL ADD   $PUSH   JP    EDOC
```

3.2. Exiting from a CODE Word

After a code definition has finished executing, it needs to return to the control of the calling word. Since code definitions are normally called from colon definitions, the return must be made to the colon definition, or rather the "inner interpreter" which executes colon definitions.

Returning control to the inner interpreter is normally accomplished by branching to one of the entry points listed in Figure 3-2.

Entry name	Description
$NEXT	The standard exit from a code word. The inner interpreter continues executing where it was before.
$PUSH	Push the register pair HL on the parameter stack and branch to $NEXT.
$NEXTHL	Similar to $NEXT, except that the alternate register is selected.

Figure 3-2. Summary of inner interpreter entry points

Example: The code definition of a word which adds 3 to the value on the top of the stack, and pushes the sum, might be written:

```
CODE 3+    HL POP     3 BC LD    BC HL ADD
           HL PUSH    $NEXT JP         EDOC
```

When the above example is executed, the sum is pushed onto the stack before the inner interpreter is invoked. The same thing could have been written as:

```
CODE 3+    HL POP     3 BC LD   BC HL ADD
           $PUSH      JP        EDOC
```

3.3. Branching Within CODE Definitions

There are three assembler constructions which enable you to branch within a CODE word. In order to use these words, the mnemonic assembler must be present.

Forward Branching

In colon definitions, the words IF ELSE and ENDIF composed the forward branching construction. In code definitions, the words are IF, ELSE, and ENDIF,. The difference between the two constructions (outside of the trailing ",") is that the assembler word IF, tests a Zilog Z-80 condition code (cc) during execution, rather than a Boolean stack value.

```
cc IF, ..true part..  {ELSE, ..false part..} ENDIF,
```

Figure 3-3. Format of the IF,-ELSE,-ENDIF, conditional

The format for the conditional branch is shown above (the portion enclosed by the "{}" is optional). The cc part is the condition code (C, NC, Z, NZ, P, M, PO, or PE) which, if TRUE (<>0) at execution time, will cause the machine instructions between the IF, and the ELSE, to be executed. If cc is not TRUE, the code following the ELSE, (if present) will be executed. In both cases, the code following the ENDIF, is executed.

```
             +------------+
Address 2 | JP $PUSH    |
Address 1 | LD HL,0     |
```

```
                | JR Addr2   |
                | LD HL,1    |
                | JR Z,Addr1 |
                | OR H       |
                | LD A,L     |
Entry point -->| POP HL     |
               +------------+
```

Figure 3-4. Memory diagram of the word 0<>

Example:

```
CODE 0<> HL POP  L A LD  H OR  NZ IF,
           1 HL LD
           ELSE, 0 HL LD
         ENDIF, $PUSH  JP   EDOC
```

The memory image of the above example is given in Figure 3-4. When the word 0<> is executed, the top of the stack is tested for a non-zero value. If it is non-zero, a 1 is loaded into the register pair HL. If the Top Of Stack is zero, a 0 is loaded into HL. The inner interpreter entry point $PUSH then places the value of HL on the stack, and drops into $NEXT. Another way of writing the above example would be:

```
CODE 0<> HL POP  L A LD  H OR  NZ IF,
           1 HL LD
         ENDIF, $PUSH  JP   EDOC
```

In this version, the fact that the register pair HL already contains the FALSE value (=0) if the TOS is zero is taken advantage of.

Looping

There are two types of loop constructions which can be utilized in CODE words. The first type is the conditional loop, and its format is shown below (Figure 3-5). The BEGIN..END, assembler construction is very similar to the BEGIN,..END construction used in colon definitions, with the exception that END, tests a Zilog Z-80 condition code (cc) rather than a Boolean stack value before looping again. The code between BEGIN, and END, is executed repeatedly until the Zilog Z-80 condition code (C, NC, Z, NZ, P, M, PO, or PE) is TRUE when END, is executed.

```
BEGIN, ...  cc END,
```

Figure 3-5. Format of the BEGIN,..END, construct

An example of the conditional loop is shown below, with the memory image of the example given in Figure 3-7.

```
CODE BMOVE
     BC POP  DE POP  HL POP  BEGIN,
     (HL) A LD  A (DE) LD  HL INC
     DE INC  BC DEC  B A LD  C OR
     Z END, $NEXT JP   EDOC
```

Figure 3-6. Example using the BEGIN,..END, conditionals to define the word BMOVE

```
                +-------------+
                | JP  $NEXT   |
                | JR  NZ,Addr1|
                | OR  C       |
                | LD  A,B     |
                | DEC BC      |
                | INC DE      |
                | INC HL      |
                | LD (DE),A   |
         Addr1  | LD  A,(HL)  |
                | POP HL      |
                | POP DE      |
Entry point -->| POP BC      |
               +-------------+
```

Figure 3-7. Memory image of the BMOVE example

In the example, the code between the BEGIN, and the Z END, is executed repeatedly until the register pair BC becomes zero. Notice that, if BC is initially zero, the loop is executed 65536 times, instead of not executing at all, which is not what we wanted to do. To cure this, a pre-test is needed, and this is shown in Figure 3-8.

```
CODE BMOVE
     BC POP  DE POP  HL POP  B A LD  C OR
     NZ IF,
          BEGIN,
               (HL) A LD  A (DE) LD  HL INC
               DE INC  BC DEC  C A LD  B OR
          Z END,
     ENDIF, $NEXT JP   EDOC
```

Figure 3-8. The BMOVE example with a pre-test for zero.

The format of the other type of loop, the unconditional loop, is given in Figure 3-9, and an example of its usage is shown in Figure 3-10.

```
BEGIN, ...  REPEAT,
```

Figure 3-9. Format of the unconditional assembler loop

```
CODE WAIT   BC POP  BEGIN,
          20 IN  C CP  Z IF,
             $NEXT JP
          ENDIF,
     REPEAT,  EDOC
```

Figure 3-10. Example using BEGIN,-REPEAT,

In the above example, the code between the BEGIN, and the REPEAT, is executed until the data read from port 20 is equal to the TOS upon entry of WAIT. Notice that no code follows the REPEAT,. This is because it normally would not be executed.

4. Assembler Mnemonics

This section compares the SL5 assembler mnemonics with the Zilog Z-80 mnemonics. Refer to the "Mostek Z-80 Programming manual" for more details on the operation and usage of the Zilog Z-80 instruction set.

In the first part of this section, the format of the SL5 mnemonics is discussed and, in the second part, the SL5 mnemonics are listed with their Zilog Z-80 counterparts in alphabetic order.

The SL5 mnemonics differ from their Zilog counterparts as follows:

(1) Because of the nature of SL5, the operands must precede the mnemonic.

Example: The instruction: LD C,3
is written as: 3 C LD

(2) Certain operand types have been changed, so as to keep the SL5 assembler clean. These differences are listed in Figure A-1 (where two forms are shown, either may be used).

Zilog	SL5
(adr)	adr ^ ^ adr
(IX+n)	n IX+ IX+ n
(IY+n)	n IY+ IY+ n
(n)	n ^ ^ n
I	IR

Figure A-1. Operand differences

Notation

```
  r,r1,r2 -- Any of the Z-80 registers {A,B,C,D,E,H,L,(HL)}
      rp1 -- Any of the Z-80 register pairs {BC,DE,HL,SP}
      rp2 -- Any of the Z-80 register pairs {BC,DE,HL,AF}
      rp3 -- Any of the Z-80 register pairs {BC,DE,IX,SP}
      rp4 -- Any of the Z-80 register pairs {BC,DE,IY,SP}
       cc -- Any of the Z-80 condition codes {C,NC,Z,NZ,PO,PE,P,M}

        b -- A  3-bit integer
  n,n1,n2 -- An 8-bit integer
       nn -- A 16-bit integer
      adr -- A 16-bit address
```

The following is an alphabetically-sorted list of assembler mnemonics comparing the Zilog form with the SL5 form. Any duplications which appear imply that either form shown may be utilized.

Zilog	SL5
ADC r	r ADC
ADC n	n ADC
ADC HL,rp1	rp1 HL ADC
ADC (IX+n)	n IX+ ADC
ADC (IY+n)	n IY+ ADC
ADD r	r ADD
ADD n	n ADD
ADD HL,rp1	rp1 HL ADD
ADD IX,rp3	rp3 IX ADD
ADD IY,rp4	rp4 IY ADD
ADD (IX+n)	n IX+ ADD
ADD (IY+n)	n IY+ ADD
AND r	r AND
AND n	n AND
AND (IX+n)	n IX+ AND
ADD (IY+n)	n IY+ ADD
BIT b,r	r b BIT
BIT b,(IX+n)	n IX+ b BIT
BIT b,(IY+n)	n IY+ b BIT
CALL adr	adr CALL
CALL cc,adr	adr cc CALL
CCF	CCF
CP r	r CP
CP n	n CP
CP (IX+n)	n IX+ CP
CP (IY+n)	n IY+ CP
CPD	CPD
CPDR	CPDR
CPI	CPI
CPIR	CPIR
CPL	CPL
DAA	DAA
DEC r	r DEC
DEC rp	rp DEC
DEC IX	IX DEC
DEC IY	IY DEC
DI	DI
DJNZ n	n DJNZ
EI	EI
EX AF,AF'	AF' AF EX
EX DE,HL	HL DE EX
EX (SP),HL	HL (SP) EX
EX (SP),IX	IX (SP) EX
EX (SP),IY	IY (SP) EX
EXX	EXX
HALT	HALT
IM0	IM0
IM1	IM1

```
IM2             IM2
IN A,(n)        n IN
IN A,(n)        n ^ A IN
IN r,(C)        (C) r IN
INC r           r INC
INC rp          rp INC
IND             IND
INDR            INDR
INI             INI
INIR            INIR
JP adr          adr JP
JP cc,adr       adr cc JP
JP (HL)         (HL) JP
JP (IX)         (IX) JP
JP (IY)         (IY) JP
JR n            n JR
JR cc,n         n cc JR
LD r1,r2        r2 r1 LD
LD r,n          n r LD
LD r,(IX+n)     n IX+ r LD
LD r,(IY+n)     n IY+ r LD
LD (IX+n),r     r n IX+ LD
LD (IY+n),r     r n IY+ LD
LD (IX+n1),n2   n2 n1 IX+ LD
LD (IY+n1),n2   n2 n1 IY+ LD
LD A,(BC)       (BC) A LD
LD A,(DE)       (DE) A LD
LD A,(adr)      adr ^ A LD
LD (BC),A       A (BC) LD
LD (DE),A       A (DE) LD
LD (adr),A      A adr ^ LD
LD A,I          IR A LD
LD A,R          R A LD
LD I,A          A IR LD
LD R,A          A R LD
LD rp1,nn       nn rp1 LD
LD IX,nn        nn IX LD
LD IY,nn        nn IY LD
LD rp1,(adr)    adr ^ rp1 LD
LD IX,(adr)     adr ^ IX LD
LD IY,(adr)     adr ^ IY LD
LD (adr),rp1    rp1 adr ^ LD
LD (adr),IX     IX adr ^ LD
LD (adr),IY     IY adr ^ LD
LD SP,HL        HL SP LD
LD SP,IX        IX SP LD
LD SP,IY        IY SP LD
LDD             LDD
LDDR            LDDR
LDI             LDI
LDIR            LDIR
NEG             NEG
NOP             NOP
```

```
OR r            r OR
OR n            n OR
OR (IX+n)       n IX+ OR
OR (IY+n)       n IY+ OR
OTDR            OTDR
OTIR            OTIR
OUT (n),A       n OUT
OUT (n),A       A ^ n OUT
OUT (C),r       r (C) OUT
OUTD            OUTD
OUTI            OUTI
POP rp          rp POP
PUSH rp         rp PUSH
RES b,r         r b RES
RES b,(IX+n)    n IX+ b RES
RES b,(IY+n)    n IY+ b RES
RET             RET
RET cc          cc RET
RETI            RETI
RETN            RETN
RST n           n RST
RL r            r RL
RL (IX+n)       n IX+ RL
RL (IY+n)       n IY+ RL
RLA             RLA
RLC r           r RLC
RLC (IX+n)      n IX+ RLC
RLC (IY+n)      n IY+ RLC
RLCA            RLCA
RLD             RLD
RR r            r RR
RR (IX+n)       n IX+ RR
RR (IY+n)       n IY+ RR
RRC r           r RRC
RRC (IX+n)      n IX+ RRC
RRC (IY+n)      n IY+ RRC
RRA             RRA
RRAC            RRAC
RRD             RRD
SBC r           r SBC
SBC n           n SBC
SBC HL,rp1      rp1 HL SBC
SBC (IX+n)      n IX+ SBC
SBC (IY+n)      n IY+ SBC
SCF             SCF
SET b,r         r b SET
SET b,(IX+n)    n IX+ b SET
SET b,(IY+n)    n IY+ b SET
SLA r           r SLA
SLA (IX+n)      n IX+ SLA
SLA (IY+n)      n IY+ SLA
SRA r           r SRA
SRA (IX+n)      n IX+ SRA
SRA (IY+n)      n IY+ SRA
SRL r           r SRL
SRL (IX+n)      n IX+ SRL
SRL (IY+n)      n IY+ SRL
SUB r           r SUB
SUB n           n SUB
SUB (IX+n)      n IX+ SUB
SUB (IY+n)      n IY+ SUB
XOR n           n XOR
XOR r           r XOR
XOR (IX+n)      n IX+ XOR
XOR (IY+n)      n IY+ XOR
```

5. Register Usage

Figure B-1 is a list of the Zilog Z-80 registers and their usage. Any register designated as unused may be altered within a code word. Any register which is reserved (SP, DE', and HL') may also be used inside a code definition, if its contents is saved upon entry and restored upon exiting from the code word.

```
A  -- Unused 8-bit register
BC -- Unused 16-bit register pair
DE -- Unused 16-bit register pair
HL -- Unused 16-bit register pair
IX -- Unused 16-bit register
IY -- Unused 16-bit register
SP -- Parameter stack pointer

Alternate register set
----------------------
A'  -- Unused 8-bit register
BC' -- Unused 16-bit register pair
DE' -- Return stack pointer
HL' -- Interpreter Pointer (IP)
```

Figure B-1. Register usage

6. Examples of CODE Definitions

The first example multiplies the Top Of the Stack by 2, and pushes the product on the data stack. The equivalent colon definition would be: : 2* DUMP + ;

```
CODE 2* HL POP  HL HL ADD  $PUSH  JP   EDOC
```

The next example test the Top Of the Stack, to see if it is negative. If it is negative, the value 1 (TRUE) is returned; otherwise, the value (FALSE) is returned.

```
CODE 0< HL POP  H 7 BIT  NZ IF,
             1 HL LD
             ELSE,  0 HL LD
             ENDIF, $PUSH  JP   EDOC
```

The parameter is popped, its sign bit is then checked and, if it is set (indicating a negative number), a '1' (TRUE) is loaded into the register pair HL; otherwise, a '0' (FALSE) is loaded into HL. Then, $PUSH is jumped to, which will push HL on the stack and continue.

the word 0< could have been written several different ways which are more efficient in terms of speed and memory usage. One of these is shown below.

```
CODE 0<
     BC POP  0 HL LD  B 7 BIT  NZ IF,
        HL INC
     ENDIF, $PUSH  JP   EDOC
```

The final example polls I/O port 40 (decimal) continuously until a non-zero value appears. Then I/O port 41 (decimal) is read and the data is pushed on the stack.

```
CODE IN41
     BEGIN,
          40 IN  A OR
     NZ END,  41 IN  A L LD  0 H LD
     $PUSH  JP   EDOC
```

For more examples of code definitions, examine the definitions of the words (SWAP, +, DUP, etc.) in the kernel.

Application Note #1

(Z-80 Version)

Branching to Externals

Sometimes, it becomes necessary to branch to (or "CALL") an external assembly language routine from a word, and return to that word after some action is performed. This note will illustrate two ways in which this can be accomplished by the use of an additional code word.. The assembler is assumed to be present in the following examples.

The first example branches to a specified address EXAD, where the following code exists.

```
EXAD:  LD      A,20H           ; A Space character
       OUT     (0D1H),A        ; Send it to I/O port D1
       RET                     ; Return to caller
```

In the SL5 portion of the program, the following words are defined.

```
F600 CONSTANT EXAD      ( Address of routine )

CODE BRANCH-EXAD       EXX  DE PUSH  HL PUSH  EXAD CALL
                       HL POP  DE POP  $NEXTHL JP   EDOC
```

Whenever the word BRANCH-EXAD is executed, the alternate register pairs DE and HL are saved on the stack before, and are restored after calling EXAD.

The second example to be given is slightly more complex than the previous one, because it branches to an address placed on the stack, and passes parameters to and from the external. This example will assume that the following assembly language routines exist, starting at F605.

```
EXAD2:  LD      A,C             ; Load A with the data
        OUT     (0D1H),A        ; Send it to I/O port D1
        RET                     ; Return to caller

EXAD3:  IN      A,(0D2H)        ; Read data from I/O port D2
        RET                     ; Return to caller
```

In the SL5 portion of the program, the following words are present.

```
F609 CONSTANT EXAD3     ( Address of EXAD3 )

CODE BRANCH    HL POP  BC POP
               EXX  DE PUSH  HL PUSH  EXX
               HERE 5 + DE LD  DE PUSH  (HL) JP
               EXX  HL POP  DE POP  EXX
               A L LD  0 H LD  $PUSH  JP   EDOC

: TEST1 " A EXAD2 BRANCH DROP ;
: TEST2 0 EXAD3 BRANCH .  ;
```

Whenever TEST1 is executed, the letter A (41 hexadecimal) will be sent to port D1. Whenever TEST2 is executed, port D2 is read, and the value is displayed via the ".".

Debug

1. Introduction

The debugging utilities are provided to aid the programmer in discovering and fixing the "bugs" which ail their programs. The debugging routines may be loaded at any time by executing:

```
>FLOAD DEBUG.SL5 <cr>
```

the words which are described in this section are now loaded, and ready to be used.

2. DUMP

The DUMP utility allows you to examine any portion of your CPU's memory. The form of DUMP is:

```
e s DUMP
```

Where "s" is the Starting memory address, and "e" is the End address. DUMP will display 16 bytes of memory on a line with the starting address given first, followed by the 16 bytes in numeric and ASCII character form (any codes between 0-20 hex are shown as a "."). All numbers are displayed in hexadecimal notation.

```
Example: >120 100 DUMP <cr>

0100 C3 4E 06 E5 D9 4E 23 46 23 C5 D9 C9 E1 5E 23 56  CN.eYN#F#EYIa^"V
0110 C5 C3 04 01 C3 04 01 CD 0C 01 82 67 CD 0C 01 84  UC..C..M...gM...
```

3. MODIFY

The word "MODIFY" lets you examine and alter the contents of memory on a byte-to-byte basis, starting at a selected address. The form of the MODIFY command is:

```
s MODIFY
```

MODIFY will then proceed by displaying the contents of 16 bytes starting at address "s", and ask for input, which can be

1. A two-digit hexadecimal number, which will replace the byte which is shown above it.
2. A Carriage Return, which causes MODIFY to proceed to the next 16 bytes.
3. An "escape" (ESC=Ctrl-[), which takes you out of MODIFY, back to the outer interpreter.
4. Any other character, which causes MODIFY to proceed to the next byte.

Example: If you executed the following:

```
>100 MODIFY <cr>

0100 C3 4E 06 E5 D9 4E 23 46 23 C5 D9 C9 E1 5E 23 56  CN.eYN#F#EYIa^"V
0100 11 22 33 __ __ __ 44 <cr>
0110 C5 C3 04 01 C3 04 01 CD 0C 01 82 67 CD 0C 01 84  UC..C..M...gM...
0110 40 41 42 __ 43 44 __ 45 46 __ 1B <ESC>

>120 100 DUMP <cr>

0100 11 22 33 E5 D9 4E 44 46 23 C5 D9 C9 E1 5E 23 56  ."3eYNDF#EYIa^"V
0110 40 41 42 01 43 44 01 45 46 01 1B 67 CD 0C 01 84  @AB.CD.EF..gM...
```

4. PSDMP, RSDMP -- Stack Dumping

The words PSDMP and RSDMP allow you to examine the contents of either the data stack or

the return stack, easily. The contents are displayed as 2 bytes to a line, in both hexadecimal form and character form. The Top Of the Stack is shown first, while the bottom is shown last.

Example:

If the parameter stack contained the values 2, 96, and 4463 (hexadecimal), then executing:

```
>PSDMP <cr>
0002 ..
0096 ..
4463 Dc
>
```

5. *BREAK*, *UB* -- Breakpointing

Breakpoints can be set anywhere within colon definitions by inserting the word "*BREAK*" into the definition. When the breakpoint subsequently executes, control is passed to the outer interpreter, so that you may examine or alter the environment (the stack, variables, etc.). To continue execution after the breakpoint, you only need to execute the word "*UB*".

Example:

Suppose we defined a word called "EX1", which added the two numbers on the stack, and displayed the sum afterwards.

```
: EX1 + .  ;
```

To set a breakpoint before displaying the sum, we only need to alter the definitions, so as to resemble:

```
: EX1 + *BREAK* .  ;
```

Now when the word EX1 is executed:

```
>2 3 EX1 <cr>
BREAK: EX1 1139
>DUP .  <cr>
5
>*UB* <cr>
5
>
```

When the breakpoint occurred in the above example, the word in which the breakpoint occurred was shown, along with the breakpoint address (in this case, "EX1 1139"). The Top Of Stack was then examined by executing "DUP .". Control was then returned to the word following the "*BREAK*" in EX1, by the execution of "*UB*". The Top Of Stack was then displayed via the "." in EX1.

6. SYM', SYMDUMP -- Dictionary Examination

There are two words, which are included in the debugging package, that let you examine the symbol table. The first to be described is "SYM'", which displays the entry of a particular word.

```
Example: >SYM' LEMMON <cr>
9067  F39A  7876  2011  00  LEMMON
```

In the above example, the address of the dictionary entry was first shown (9067), followed by the relative link address (F39A) and the absolute link address (7876). The execution address was then displayed (2011), followed by the flags (00) and the word itself. All numbers are displayed in hexadecimal notation, regardless of the current radix.

The other dictionary examination tool is called "SYMDUMP". This word, when executed, displays the whole dictionary, starting at CONTEXT, in the same format as "SYM'".

CP/M Interface

1. Introduction

CP/M from Digital Research is a general-purpose microcomputer disk operating system. it contains 4 modules: BIOS, BDOS, CCP, and TPA. BIOS contains the device drivers for disk and serial I/O units. BDOS is the program-level interface to CP/M, and also contains the kernel of the operating system. CCP is the console executive. The TPA is the execution area for user programs.

SL5 is loaded into the TPA like any compiler or program utility. On initial load, SL5 initializes stack pointers, registers, and system variables. The boot routine also relocates the symbol table into high memory, overwriting the CCP area (of CP/M Version 2). Boot obtains the relocation address from CP/M at memory address 0006H, so it always uses all of available memory, regardless of the system that SL5 was generated on.

The SL5 I/O subsystem (see Reference section, "12. I/O") is designed to run under a number of disk operating systems, besides CP/M. User calls to I/O words are translated into CP/M calls, and passed on to the BDOS. While the SL5 I/O structure is similar to CP/M calls, there are differences. Systems programmers should consult the SL5 Reference section, and CP/M Interface section, before writing code to call BDOS directly.

The user of the SL5 I/O subsystem does not have to have a working knowledge of BDOS. Character and file I/O can be done by using SL5 I/O words. CP/M traps some data errors, issues a message, and reboots. The CP/M Manual lists these error messages. SL5 traps and reports additional errors (see Reference section, "12. I/O").

2. Loading SL5 Object Files

SL5 object modules are created by SYSMAKE and COMMOD as standard CP/M COMmand files. They are loaded and executed by entering the filename from the console while in CP/M. Since the CCP section is overwritten by SL5 programs (when SL5 runs under CP/M Version 2), a warm boot must be executed to go back to CP/M. The modules are loaded into the TPA at address 0100H. SL5 does not use the area from 0000H to 0100H ("Page Zero"), or the CP/M BDOS, BIOS areas.

User programs can overlay BDOS and BIOS; no bounds checking is done by SL5. If the user program subsequently attempts to use the SL5 I/O subsystem or BDOS, the results are unpredictable, and usually undesirable. Programmers can write code to call BDOS and BIOS directly, but a throughout knowledge of CP/M and SL5 I/O is required. The SL5 I/O subsystem should be sufficient for most applications.

3. FLOAD -- Loading SL5 Source Files

There are several files on the SL5 distribution disk. SL5.DOC describes each file, and how it is used. SL5.COM is the compiled kernel. DEV.COM is the compiled kernel plus the assembler and debugging vocabulary. With FLOAD, the user can compile additional source files or libraries into the TPA. New files are compiled, extending the symbol table and code segment. A fatal error occurs if the symbol table collides with the code segment.

```
Example:

A>DEV              ( Load DEVelopment version of SL5 from CP/M.  )
>FLOAD USER1.SL5   ( Load another SL5 program, USER1, from DEV.  )
>                  ( DEV now ready to execute USER1 words.  )
```

SL5 has 2 systems for creating new CP/M COMmand files. After a new library or application program is debugged, it can be saved as an executable file. COMMOD saves the memory image after compressing out the free space between the code segment and the symbol table.

On initial load, the symbol table is relocated into high memory. The CP/M Version 2 SAVE utility stores the object file. The second system, SYSMAKE, compiles the kernel and user code directly onto a disk file. SYSMAKE is used to create new kernels, applications programs without the development words, and ROMable object files (see Object Module section).

4. CALLCPM

The interface between SL5 I/O and CP/M's BDOS is through an SL5 code word, which is an assembly language routine.

```
CODE CALLCPM
     BC POP  DE POP  0005 CALL  0 H LD  A L LD
     $PUSH   JP      EDOC
```

On entry, the TOS contains a function code which is put into the C register. The NOS is an information block address (FCB address). Location 0005H contains a jump to a location in CP/M that is the entry point of the BDOS. On return, the A register contains a status byte. $PUSH JP pushes the status on the parameter stack, and executes $NEXT -- the inner loop of SL5.

5. Serial I/O

SL5 uses four CP/M serial character functions: check console status, read console character, write console character, and write character to the list device. These four functions map into the character I/O words described in the Reference section. Since all character I/O is performed through CP/M, SL5 programs are easily transported across different machines running under CP/M.

```
Serial devices:
       #CRT -- Console input and output
       #List -- Printer output
```

6. Disk I/O

The CP/M interface contains a number of functions for disk I/O, including open, close, read, write, create, rename, etc. These functions map into SL5 disk I/O words described in the Reference section. The function calls and details of CP/M BDOS interface are transparent to the user for the most part. CP/M traps and handles a few disk I/O errors, including fatal read and write errors. The error codes are described in the CP/M Manual. CP/M usually must be rebooted after one of these errors. The SL5 I/O subsystem traps some additional errors, displayed in the Reference section ("12. I/O"). The SL5 error message is displayed on the console, and RESTART -- the warm start routine -- is executed.

7. OPENR, OPENW

CP/M has one open file routine, and allows concurrent reads and writes on the same file. Most other microcomputer operating systems require that a file be opened for either reading or writing, but not both. SL5 has separate open read and open write words. The user can easily modify the kernel to allow concurrent reads and writes, but the code will not be transportable to other operating systems SL5 is implemented on. OPENW executes a delete function before open, to prevent file name duplication in the directory.

Object Modules

1. Introduction

The SL5 distribution disk has 2 routines for creating complete execution files or object modules, the COMMOD utility and the SYSMAKE routine. This section contains a discussion of ways to recreate, extend, or collapse the SL5 development system, along with custom code to create a system tailored to a particular product or application. This could be anything from adding a library of words to expand the development environment, to a small subset of the system for an EPROM-based controller. COMMOD is used to create a memory image that is then "SAVEd" as a CP/M object module (COMmand file). SYSMAKE is a cross-compiler, and creates an entirely new system with a new symbol table.The new system can be ROM-based, and may be collapsed as desired.

2. Compiling a Subset of the SL5 Kernel

The SL5 kernel, written in SL5, can be divided into 6 sections: SL5 primitive words, the interpreter, the compiler, console I/O, the file system, and user I/O. Compiler directives IFTRUE-OTHERWISE-IFEND control compilation of each section, based on the value of constants defined in the kernel source.

The SL5 primitive library is a set of 70 SL5 words and system variables. Most of these words are needed in every kernel. The library occupies about 1400 bytes in the code segment. The other five sections can be compiled or left out, depending on the application. If a user program references a word that is left out of the kernel, an undefined error message is printed.

Section	Size (bytes)	Function
Primitive library	1400	Basic function library
Interpreter	1K	Outer interpreter
Compiler	1K	Compile new words
Console I/O	500	Console I/O words
File system	1.5K	Interface to CP/M disk operating system.
User I/O	100	Direct I/O words

Outer Interpreter

The outer interpreter is the main loop of SL5. It processes input commands from the console, converts strings to numbers, and executes words defined in the dictionary. it must be compiled in any system that needs to access the dictionary. The outer interpreter can be used in an application or control program to process commands and select different drivers. Functions like WORD, NUMBER, and FIND can reduce the amount of custom programming needed to implement control functions. The outer interpreter section requires console I/O, and either the file system or user I/O section.

Compiler

The compiler section is a 1K library of words that supports creation of new SL5 words and vocabularies. It is needed in a development system. it can be deleted from systems that do not need to create new words at execution time.

Console I/O

The console I/O library is a set of about 20 words that input and output characters, numbers, and messages on the CRT. It must be included in a system that contains the interpreter or the compiler. It requires either the file system or a user I/O section.

File System

The SL5 file system is the link to the host disk operating system. it also contains channel I/O, providing buffered input and output to serial devices. This section can be deleted from a kernel that does not use the disk system, but a user I/O section must be added if the console I/O or outer interpreter section is compiled.

User I/O

The SL5 file system and the host disk operating system take up several thousand bytes of memory. In many cases, the host disk operating system cannot be executed from ROM. A user I/O section can be substituted for the file system, to support console I/O. A sample user I/O section is listed below. The direct I/O words CIN, COUT, and CIS are installation-dependent. The I/O address and status mask depend on hardware addresses and the UART used.

```
( ---------------------------- )
(       OWNIO Section          )
( ---------------------------- )
       ?OWNIO IFTRUE

80 BARRAY TBUFF
0 VARIABLE TBUFP
0 VARIABLE BUFSIZE

: COUT BEGIN
  ( i8251 character output word -- Device addr ED control EC data )
     0ED ZIN  1 &
END 0EC ZOUT ;

: CIN BEGIN ( Serial input -- Device addr ED control EC data )
    0ED ZIN  2 &
END 0EC ZIN  7F &  DUP COUT ;

: CIS ;
       ( CTYPE, $C" )
: CTYPE DUP IF
       0 DO
           DUP B@ COUT 1+
       LOOP DROP
       ELSE 2DROP
ENDIF ;

: $C" R> DUP B@ SWAP 1+ SWAP 2DUP + >R CTYPE ;  ( Type string to console )

: TCH COUT ;

: TGET " > COUT  0 TBUFP !  BEGIN
      CIN  DUP TBUFP @ TBUFF B!  CASE
           0D =: TBUFP 1+!  0A COUT  1                      ;:  ( CR  )
           1B =: C" *ESC*"  0D COUT  0A COUT  0 TBUFP  ! 0 ;:  ( ESC )
           08 =: TBUFP @ IF                                    ( BS  )
                    TBUFP 1-!
                 ENDIF 0 ;:
           NOCASE =: TBUFP 1+! 0 ;:
      CASEND
END TBUFP @ BUFSIZE ! 0 TBUFP ! ;

: GCH TBUFP @ BUFSIZE @ >= IF
     TGET RECURSE
     ELSE TBUFP @ TBUFF B@ TBUFP 1+!
ENDIF ;

: UGCH GCH ;

: OUTINIT ;
: ININIT 0 TBUFP ! 0 BUFSIZE ! ;
```

```
        IFEND
( ---------------------------- )
(        End OWNIO Section     )
( ---------------------------- )
```

The Symbol Table

The SL5 symbol table is a linked list of the symbol names and code segment addresses of all the words in the dictionary. The symbol table is used by the outer interpreter and compiler. The symbol table occupies approximately 25% of the space used by an SL5 program. It can be removed in SYSMAKE by responding with a 'N' to the question: "Do you want a symbol table (Y/N) ?". In COMMOD, the variable ?SYMTAB should be set to 0 (FALSE) if the symbol table is not needed.

Deleting a Section of the Kernel

```
Compiler Control Constants
------------------------------------------
?INTERP         Compile interpreter words
?COMPILE        Compile compiler words
?CRT            Compile console I/O words
?FILESYS        Compile the SL5 file system
?OWNIO          Compile user I/O words
```

The standard SL5 is compiled with ?INTERP, ?COMPILE, ?FILESYS, and ?CRT on (set to 1 (TRUE)). ?OWNIO is set to 0 (FALSE). Compiler words can be deleted without affecting the rest of the kernel. The compiler library requires the interpreter, the file system, and the CRT words. The interpreter library requires the CRT and either the file system or user I/O section. The CRT library needs the file system or a user I/O library.

3. Generating a ROM-based Controller Program

1. Write the application program on a file.
2. Debug the program using DEV.COM and FLOAD.
3. Compile a RAM system with SYSMAKE, and test.
4. Write user I/O words, and debug with DEV.COM.
5. Add OWNIO section to KERNEL.SL5.

   Set ?COMPILE, ?INTERP, and ?FILESYS to 0 (FALSE).

   Set ?OWNIO to 1 (TRUE). (Edit KERNEL.SL5 source to change these constants.)
6. Compile a ROM system with SYSMAKE.COM.

   Set the variable GOQIAD to the address of the driver word.

   ```
   : DRIVER ( Your driver word.  ) ;

   'B DRIVER GOQIAD T!
   ```

   When the program is loaded from disk, or jumped to on coldstart, the driver program will be executed after initialization of SL5.
7. Load the program from disk, and test.
8. Blast the object code into EPROM, and test.

Note: the words T! and T@ should be used in SYSMAKE when storing values in variables or memory locations during SYSMAKE execution. The words @ and ! should still be used when compiling.

```
Example: 0 VARIABLE XYZ
         99 XYZ T!
         : TEST 99 XYZ ! ;
```

The program should be tested and debugged using DEV.COM before compiling with SYSMAKE.COM. The ROM object file can be loaded and tested using the disk operating system before blasting EPROMs. After the program is tested, generate the production version with SYSMAKE.COM.

Note: The examples that follow include prompts, output messages, and console input as displayed on the console CRT during execution.

Example 1: Generating a ROM module DRIVER.COM:

```
A>SYSMAKE                              ( Load SYSMAKE program from CP/M.  )

SYSMAKE version 1.2    Z80 - CP/M
(C) Copyright 1980 The Stackworks

Enter 'STAT' to examine parameters
Enter 'RAMGEN' to generate a RAM based system
Enter 'ROMGEN' to generate a ROM based system

>1 infof !      ( Turn off compiler source listing.  )
>romgen
Enter first address in ROM >4000
Enter highest RAM address  >E000

Enter object file name >driver.com
Enter Kernel source file name >kernel.sl5        ( Compile KERNEL.SL5.  )
(...)
   kernel.sl5 is compiled onto driver.com
(...)
More input (Y/N) ? y
Enter source file name >driver.sl5               ( Compile DRIVER.SL5.  )
(...)
   driver.sl5 is compiled onto driver.com
(...)
More input (Y/N) ? n
Do you want a symbol table (Y/N) ? n    ( Do not generate symbol table.  )

Successful compilation

Program size = CF1 / 3313                       ( Hex/decimal notation.  )
Variable space used = 2C6 / 1736
Total memory used = FB7 / 5049

A>                                              ( Back to CP/M.  )
```

4. Generating a RAM COM module with SYSMAKE

1. Write the program.
2. Compile and debug it using DEV.COM.
3. Compile using SYSMAKE and all of KERNEL.SL5, and test.
4. Set conditional compile switches to delete un-needed sections of the kernel.
5. Compile with SYSMAKE, load and test.

Note: the words T! and T@ should be used in SYSMAKE when storing values in variables or memory locations during SYSMAKE execution. The words @ and ! should still be used when compiling.

```
Example: 0 VARIABLE XYZ
         99 XYZ T!
         : TEST 99 XYZ ! ;
```

Example 2: Generating a RAM-based COM file with SYSMAKE:

```
A>SYSMAKE                              ( Load SYSMAKE program from CP/M.  )

Sysmake version 1.2    Z80 - CP/M
(C) Copyright 1980 The Stackworks

Enter 'STAT' to examine parameters
Enter 'RAMGEN' to generate a RAM based system
Enter 'ROMGEN' to generate a ROM based system

>1 infof !      ( Turn off compiler source listing.  )
>ramgen

Enter object file name >myprog.com
Enter Kernel source file name >kernel.sl5        ( Compile KERNEL.SL5.  )
(...)
   kernel.sl5 is compiled onto myprog.com
(...)
More input (Y/N) ? y
Enter source file name >myprog.sl5               ( Compile MYPROG.SL5.  )
(...)
   myprog.sl5 is compiled onto myprog.com
(...)
More input (Y/N) ? n
Do you want a symbol table (Y/N) ? n    ( Do not generate symbol table.  )

Successful compilation

Program size = 1F4B / 8011                      ( Hex/decimal notation.  )
Variable space used = AF3 / 2803
Total memory used = 2A3E / 10814

A>                                              ( Back to CP/M.  )
```

5. SYSMAKE Errors and Parameters

There are a few error messages that are generated by SYSMAKE when an undefined action takes place, or one of SYSMAKE's data structures overflows. Both of these are described here, along with a remedy. Most error conditions cause the system to abort to the operating system after the message is issued.

General SYSMAKE errors

TARGET REDEF nnnn	This informative message is the same as the compilation message "REDEF nnnn", which means that the word was just redefined in the vocabulary. No adverse action takes place.
nnnn IS UNDEFINED	This informs you that the word nnnn was used without being previously defined.

Data structure overflows

COMP. BUFFER OVERFLOW	A special buffer inside SYSMAKE called the "compiler buffer" overflowed because a large word was just defined. There are two ways to resolve this problem: (1) separate the word into one or more smaller definitions, (2) re-enter SYSMAKE and set the variable "CB-SIZE" to a larger value than its initial value.
SYS ERROR #1	This message is generated when the number of defined variables exceeds the number allowed. Re-enter SYSMAKE and increase the value of the variable "TVAR-NUM" before compiling.
SYS ERROR #2	Re-enter SYSMAKE and increase the value of the variable TSYMSP before compiling.
SYS ERROR #3	This message is generated when more memory is required to complete compilation. If more memory is not available, re-enter SYSMAKE and try decreasing the values of the variables CB-SIZE, TVAR-NUM, and TSYMSP.

Setting SYSMAKE parameters

The following output demonstrates how to examine and modify SYSMAKE parameters. All user input is in uppercase characters for clarity.

```
A>SYSMAKE                          ( Load SYSMAKE program from CP/M.  )

Sysmake version 1.2    Z80 - CP/M
(C) Copyright 1980 The Stackworks

Enter 'STAT' to examine parameters
Enter 'RAMGEN' to generate a RAM based system
Enter 'ROMGEN' to generate a ROM based system

>STAT

All values are shown in hex/decimal form
Maximum number of variables (VAR-MAX) = 40 / 64
Temporary symbol table space (TSYMSP) = 200 / 512
Compiler buffer size (CB-SIZE) = 200 / 512
Starting program address (TDP) = 100 / 256    ( Auto.  set during
ROMGEN.  )

>60 VAR-MAX !           ( Set maximum number of variables to 60 hex.  )
>400 TDP !              ( Set starting address to 0400 hex.  )
>RAMGEN                 ( Now, generate a RAM-based system.  )
```

6. Generating a COM Module with COMMOD

The COMMOD program provides a quick, easy way to generate CP/M COMmand files for execution. COMMOD can discard the symbol table if desired, or it can leave it intact, to be used along with the outer interpreter. COMMOD can also execute an initialization routine which would display a message and/or initialize variables before entering the outer interpreter.

Note: If the symbol table is discarded, the outer interpreter is not functional, and an entry point for the program should be specified as the initialization routine.

Parameters for COMMOD

COMMOD has two parameters: ?SYMTAB and USERPAD. These are variables, and can be set after COMMOD.SL5 has been loaded.

?SYMTAB	If set to 0 (FALSE), the symbol table is discarded; otherwise, it is preserved. Default = 1 (TRUE).
USERPAD	This can be set to point to an initialization routine. On default, it points to a dummy routine.

Example 3: Creating a Disk COM File with COMMOD.SL5:

```
A>sl5                                   ( Load the kernel file from CP/M.  )
>1 infof !                              ( Turn off source listing.  )

>fload TEST.SL5                      ( Load user program on top of kernel.  )

>fload COMMOD.SL5                       ( Load the COMMOD program.  )
```

Set the variable ?SYMTAB to 0 if the symbol table is not needed. Execute 'MAKECOM program.com' when ready.

```
>makecom TEST.COM                        ( Generate program on TEST.COM.  )

Program size = 2D09 / 11529  ( Size shown in hexadecimal/decimal format.  )
A>                          ( SL5 returns to CP/M after saving TEST.COM.  )
```

Example 4: Creating a COM File With a user Initialization Routine:

```
A>sl5                                   ( Load the kernel file from CP/M.  )

>fload TEST2.SL5                     ( Load user program on top of kernel.  )
(...)
: MYINIT CR T" Welcome to my program version 1.0" CR ;

>fload COMMOD.SL5                        ( Load the COMMOD program.  )
```

Set the variable ?SYMTAB to 0 if the symbol table is not needed. Execute 'MAKECOM program.com' when ready.

```
>'b MYINIT userpad !             ( Set initialization address to MYINIT.  )
>makecom TEST2.COM                       ( Generate program on TEST2.COM.  )

Program size = 2D09 / 11529  ( Size shown in hexadecimal/decimal format.  )
A>                          ( SL5 returns to CP/M after saving TEST2.COM.  )
A>test2
Welcome to my program version 1.0
A>
```

Example 5: Creating a COM File Without a Symbol Table:

```
A>sl5                                   ( Load the kernel file from CP/M.  )

>fload TEST3.SL5                     ( Load user program on top of kernel.  )
(...)
: DRIVER BEGIN ( You main loop.  ) 0 END ;

>fload COMMOD.SL5                        ( Load the COMMOD program.  )
```

Set the variable ?SYMTAB to 0 if the symbol table is not needed.

Execute 'MAKECOM program.com' when ready.

```
>0 ?SYMTAB !                            ( Symbol table will be
discarded.  )

>'b DRIVER userpad !              ( Set initialization address to
DRIVER.  )

>makecom TEST3.COM                       ( Generate program on
TEST3.COM.  )

Program size = 2D09 / 11529   ( Size shown in hexadecimal/decimal
format.  )
A>                          ( SL5 returns to CP/M after saving
TEST3.COM.  )
```

Structure

1. Introduction

In the first part of this manual, you were shown how to create and use words and structures. This section describes what, actually, goes on when you create and use words and structures.

2. Memory Organization

Given in Figure 2-1 is a typical memory layout for a RAM-based system and, in Figure 2-2, for ROM-based systems. See the Interface section for the actual location of these areas in your system.

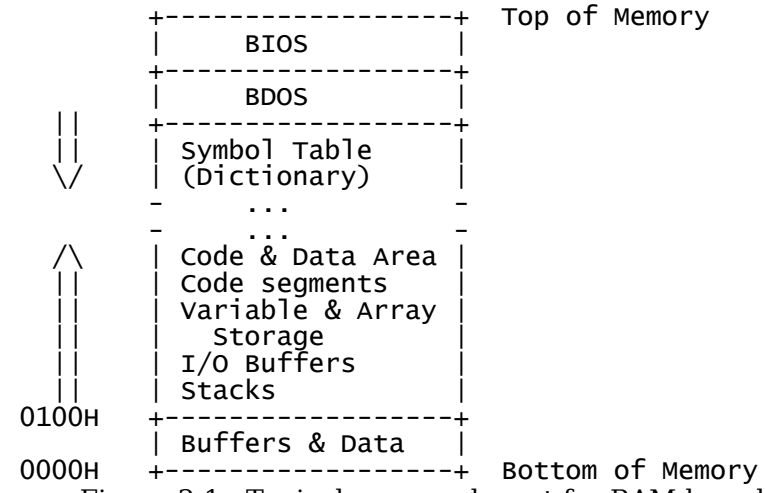

Figure 2-1. Typical memory layout for RAM-based systems

The symbol table (or dictionary) is a downward growing linked list of all the defined words, along with the code segment and a flag byte.

The code and data area contains the code segments of defined words, the I/O buffers, and the stacks (return and parameter). This area grows upward, towards the dictionary. If they collide, a fatal error results.

The parameter stack is a stack which grows downward as more values are placed onto it, and contracts upward when values are removed from it. The return stack is used for storing the interpreter pointers, and loop indexes.

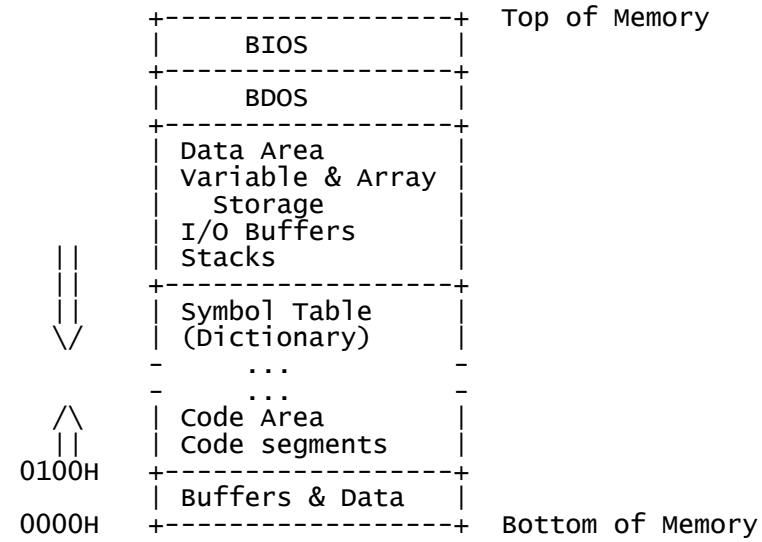

Figure 2-2. Typical memory layout for ROM-based systems

In a ROM environment, the data area is separated from the code area, so that the values of variables can be altered. The code segment area lies in ROM, while the dictionary and data area must be in RAM. The dictionary may be omitted for systems where the outer interpreter is not used.

3. Compilation of Words

Whenever a new word is defined, a search is first performed on the CONTEXT vocabulary, to see if the word has been previously defined. If the word is found, the message "REDEF nnnn" is displayed, to inform you that you are redefining that word. This message is only an informative one. Then, the word is added to the end of the CURRENT vocabulary and the CONTEXT vocabulary is set to the CURRENT vocabulary, except for CODE words, in which case CONTEXT is set to the ASSEMBLER vocabulary. What happens next varies for colon, CODE,CONSTANT, VARIABLE, and ARRAY definitions.

3.1. Colon Definitions

When a colon definition is defined, a CALL $: machine instruction is stored at the start of the new word's code segment. The variable STATE is set to 1, to indicate compile mode. During compile mode, the code segment address of words referenced inside the definition will be stored in the code segment of the word being defined, unless the referenced word has its precedence bit set.

Some words have a precedence bit set, to force execution during compile mode. One of these words is semi-colon (;). It resets the STATE to 0 (execution mode), updates the CONTEXT vocabulary, so that the word just defined is accessible, and places the address of $; in the code body (see Figure 3-1).

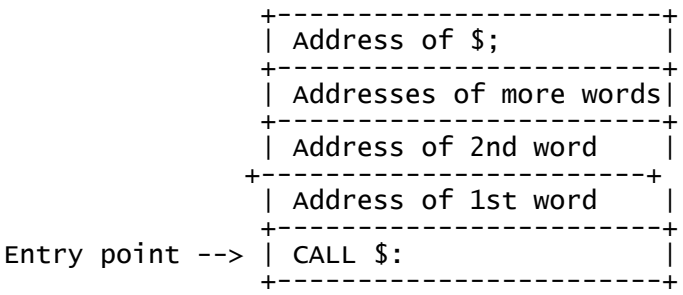

Figure 3-1. Memory format of colon definitions

When the new word is executed, a jump to the entry point is made, which causes the CPU to execute the CALL $: instruction. $: pushes the value of the interpreter pointer (IP) on the return stack, and sets the IP to the memory location following the CALL $: instruction. The inner interpreter will now execute (jump to) the addresses in the colon definition. When $; is executed, the previous value of the interpreter pointer IP is popped from the return stack.

3.1.1. Literals

During execution, literals are numbers which you type in, and are pushed onto the stack. During compilation, literals are processed in a different way, so that the number will be pushed on the stack when the word is executed, rather than when it is defined. The address of LIT and the value is compiled into the code segment.

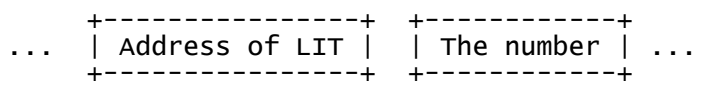

Figure 3-2. Compiled literals

Upon execution of LIT, the literal is placed on the stack, and the interpreter pointer IP is incremented to the memory location past the number, so that the inner interpreter will not execute the number.

3.1.2. T"

What happens with T" during compilation is similar to what was described for literals. The address of the word $T" is placed in the code segment, along with the string and the string's length.

Example:

If T" purple" appeared in a colon definition, the following would be placed in the body:

address of $T”		6	p	u	r	p	l	e

When executed, $T" will print

purple

and increment the interpreter pointer IP to the location following the "e" in purple.

3.1.3. Branching Within Colon Definitions

In this section, the internal structure of some of the branching constructions is described. The constructions discussed here should give you some insight into the structures which are not described in this section.

IF..ELSE..ENDIF

When IF..ELSE..ENDIF is compiled, IF ELSE and ENDIF are, actually, executed, rather than their addresses being compiled into the definition, because their precedence bit is set. IF ELSE and ENDIF put addresses of words and transfer addresses in the colon body so that, when the word is executed, the proper branching will occur.

```
Example: NINE=

: NINE= 9 = IF 1 ELSE 0 ENDIF .  ;
```

Which, when executed, will test the value on the Top Of the Stack, and type a 1 (TRUE) if it is a 9, or a 0 (FALSE) if the Top Of Stack is anything else. The definition body would then resemble:

```
                  +------------------+
                  | Address of $;    |
          address2| Address of .     |
                  | 0                |
          address1| Address of LIT   |
                  | address2         |
                  | Address of $ELSE |
                  | 1                |
                  | Address of LIT   |
                  | address1         |
                  | Address of $IF   |
                  | Address of =     |
                  | 9                |
                  | Address of LIT   |
Entry point --> | CALL $:            |
                  +------------------+
```

When NINE= is executed and the Top Of Stack (TOS) is 9, $IF will increment the interpreter pointer IP to skip over address1, LIT, will push the 1 on the stack, $ELSE will then set the interpreter pointer IP to address2 to avoid execution of the false part of the branch, and "." will print the TOS which is a 1. If the number which was on the Top Of the Stack upon entry of NINE= was not a 9, $IF will set the interpreter pointer IP to address1, and LIT will push a 0 onto the stack, which "." will print.

BEGIN..END

BEGIN..END executes during compilation, and places addresses (along with data) in the code segment.

```
: N-1.  BEGIN DUP .  1- DUP 0= END DROP ;
```

When N-1. is executed, it will copy all the numbers starting from the value on Top Of the Stack down to 1.

```
Example: >9 N-1.  <cr>
```

will produce: 9 8 7 6 5 4 3 2 1

```
                    +-----------------+
                    | Address of $;   |
                    | Address of DROP |
           address2 | address1        |
                    | Address of $IF  |
                    | Address of 0=   |
                    | Address of DUP  |
                    | Address of 1-   |
                    | Address of .    |
           address1 | Address of DUP  |
Entry point --> | CALL $:         |
                    +-----------------+
```

Figure 3-3. Memory diagram of the word N-1.

Figure 3 shows the memory image of the word N-1. The word END puts the address of $IF followed by the loop address in the code segment. The word $IF will set the interpreter pointer IP to addresss2 when the Top Of the Stack is true (non-zero) at the end of the loop.

RECURSE

When the compiler directive RECURSE is encountered during the compilation of a word, it executes immediately, and places the address of the word under construction in the definition. This enables the word to call itself recursively during execution.

```
: N-1.  DUP IF
        DUP .  1- RECURSE
        ELSE DROP
ENDIF ;
```

Execution of N-1. will cause the numbers starting from the value on the stack on entry of N-1. down to "1" to be typed.

```
Example: >7 N-1.  <cr>
```

will produce: 7 6 5 4 3 2 1

```
                    +-------------------+
           address3 | Address of $;     |
           address2 | Address of DROP   |
                    | address3          |
                    | Address of $ELSE  |
                    | Address of N-1.   |
                    |   (address1)      |
                    | Address of 1-     |
                    | Address of .      |
                    | Address of DUP    |
                    | address2          |
                    | Address of $IF    |
                    | Address of DUP    |
Entry point --> | CALL $:           |
 (address1)         +-------------------+
```

Figure 3-4. Memory diagram of the word N-1. using RECURSE

N-1. cannot be used instead of RECURSE because the CONTEXT dictionary (see Section 4.1, "The CURRENT and CONTEXT Pointers") does not contain the word N-1. until the ";" is processed. The logic behind this is that it is often desirable to be able to redefine a word using the previous definition.

3.1.4. ;: and ;CODE Constructions

The most common way to terminate a colon definition is by the use of the word ";". Two other words can be used -- ;: and ;code.

When the compiler directive ;: is used, it places two things in the word under definition before terminating compilation. The first is the address of a word $;: and the second is the machine instruction CALL $: .

Example: The definition of the word BARRAY:

```
: BARRAY HERE 5 + CONSTANT DP+! ;: @ + ;
```

Which, when used:

```
9 BARRAY BEX1
```

will define a new word called BEX1 . Figure 5 shows the memory image of these two words.

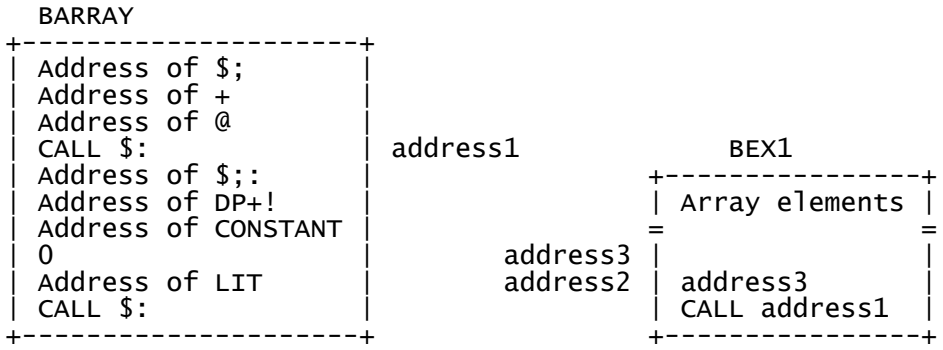

Figure 3-5. Illustration of BARRAY and BEX1

When $;: is executed, address1 is placed in the address field of the CALL instruction in BEX1, and execution of BARRAY is terminated. When BEX1 is subsequently executed, CALL address1 will push address2 on the stack, and branch to address1. CALL $: at address1 will, in turn, set the interpreter pointer IP up and start executing, as if it were a colon definition.

The compiler directive ;CODE is identical to ;: except that it defines a CODE sequence, instead of a colon sequence. Thus, the form:

```
: nnnn ...words...  ;CODE ...machine instructions...  EDOC

nnnn mmmm
```

produces the memory images shown in the figure below.

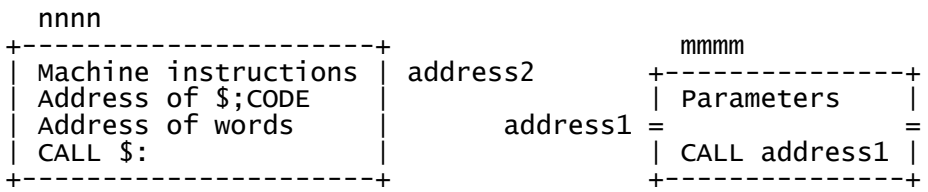

Figure 3-6. Memory format of ;CODE constructions

3.2. CODE Definitions

When you define a code word, SL5 remains in the execution mode (STATE=0), unlike colon definition where you are in compilation mode (STATE=1). Words are assembled into the code segment, to be executed by the CPU. Upon execution of the word "EDOC", CONTEXT is reset to the CURRENT vocabulary, which enables you to reference the new word immediately.

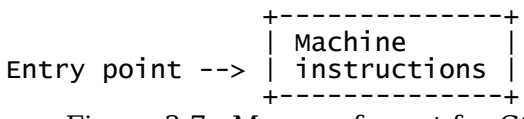

Figure 3-7. Memory format for CODE definitions

3.3. CONSTANTs

A constant pushes its value onto the stack. When you define a CONSTANT, the memory image resembles that of Figure 8. When the constant is executed, the word $CONSTANT picks up "convalue" and pushes it on the stack.

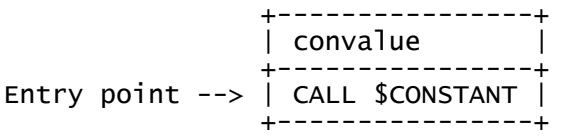

Figure 3-8. Memory format for constants

3.4. VARIABLEs

When a new variable is defined, a code segment of the form shown in Figure 9 is created. Subsequent execution of the variable results in address1 being placed on the parameter stack, to be used by words such as "!", "@", "1+!", etc.

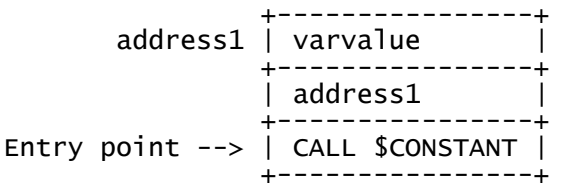

Figure 3-9. Memory format of variables

3.5. Arrays

In this section, the term "arrays" refer to the structures which are defined via the word ARRAY. Structures which are defined through the word BARRAY are identical to those of ARRAY, with the exception that the size of the elements in a BARRAY are 8 bits, while those of an ARRAY are 16 bits.

The memory image of an array is given in Figure 10 for RAM-based systems. In a ROM-based system, the elements would lie in the data area. When the array is accessed, the address of the desired element is left on the stack. The address is computed as "2*n + address1", where n is the element number.

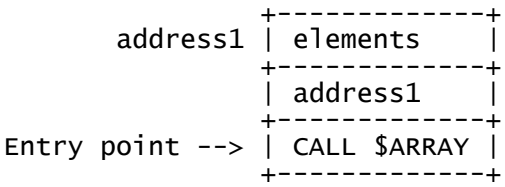

Figure 3-10. Memory format of arrays

4. The Dictionary

4.1. The CURRENT and CONTEXT Pointers

When you type in words at the top-level, a search is made in a vocabulary for the word. It is executed or compiled, depending on the value of the variable STATE. The CONTEXT vocabulary is the vocabulary searched. The variable CONTEXT points (via a negative offset from the variable SYMTP) to the first entry in the vocabulary list. The CONTEXT vocabulary is set by executing:

```
nnnn
```

where nnnn is the name of a vocabulary, such as ASSEMBLER or FORTH.

New words are added to the CURRENT vocabulary. There is a variables named CURRENT which points to a memory location which, in turn, points to the last word defined in the CURRENT vocabulary. The CURRENT vocabulary is set when you execute:

```
nnnn DEFINITIONS
```

where nnnn is the name of the new CURRENT vocabulary. Note that the CONTEXT vocabulary is also set by executing nnnn. The CURRENT and CONTEXT pointers can refer to the same vocabulary.

4.2. Vocabularies

Every word that is defined in SL5 has an entry in a vocabulary. A vocabulary is a linked list of these entries, with the first element of that list being the most-recently defined word in that vocabulary.

The Internal Structure of vocabularies

There are two different types of entries found in vocabularies. The first and most common is the symbol entry, which contains the symbol, the execution address, and a flag byte.

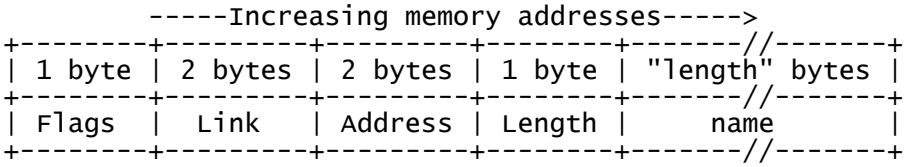

Figure 4-1. A symbol entry

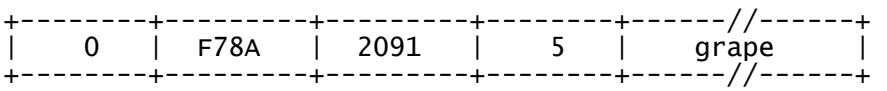

Figure 4-2. Example of a symbol entry, for the word "grape"

Every symbol entry has a flag byte (see Figure 3), followed by a two-byte link field (which, when added to the value of the variable SYMTP, gives the address of the next entry in the vocabulary), followed by the execution address, the length of the word, and the symbol.

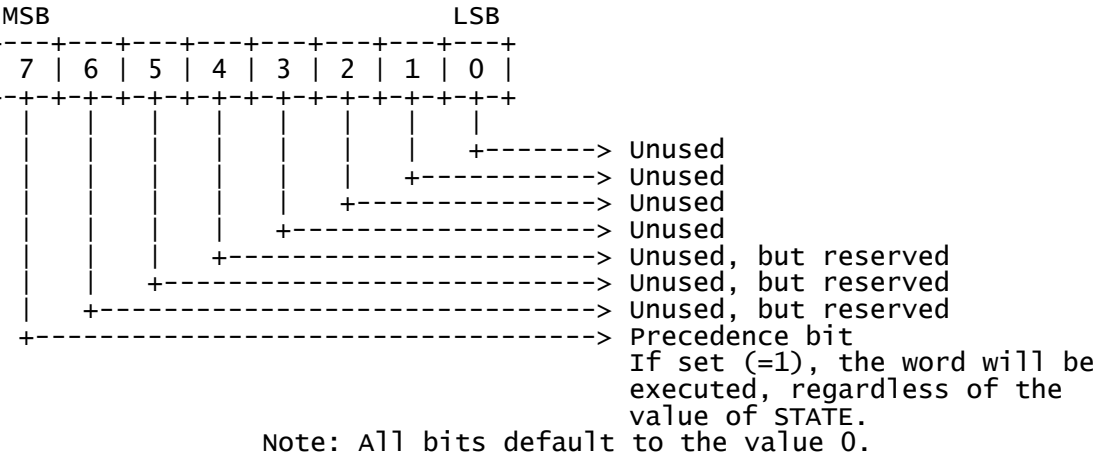

Figure 4-3. Explanation of the flag bits

The other type of entry found in a vocabulary is the vocabulary base, which appears only once in each vocabulary. For every vocabulary, except FORTH, the format of this header is given in Figure 4. The FORTH vocabulary is not chained to any vocabulary, so its "vlink" field is set to 0000H.

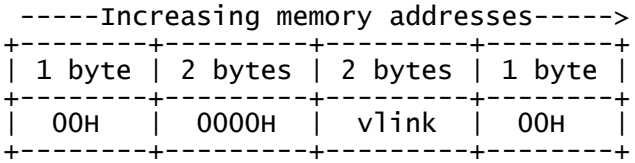

Figure 4-4. Vocabulary base format

Vocabulary Chaining

When a new vocabulary is defined, it is chained to the CURRENT vocabulary. Chaining enables the words in the CURRENT vocabulary (and the vocabulary it was chained to, if present) to be referenced while using the new vocabulary.

When the new vocabulary is created (VOCABULARY nnnn), a vocabulary base (see Figure 4) is created in the new vocabulary, with the "vlink" field pointing to the head of the CURRENT vocabulary. An example of vocabulary chaining is shown in Figure 5, with the ASSEMBLER and the USERV vocabularies chained to the FORTH vocabulary.

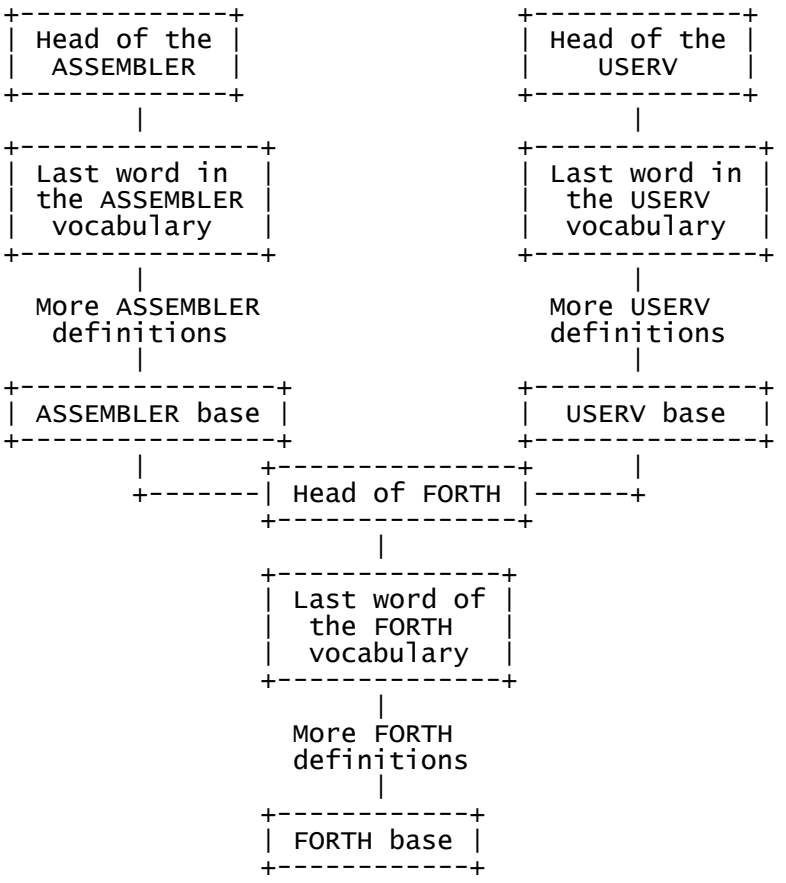

Figure 4-5. Diagram of chained vocabularies

4.3. Dictionary Reduction

In the course of debugging programs, it is often necessary to free up memory which is occupied by previous versions of the program. FORGET deletes definitions from the symbol table (dictionary).

```
Example: >FORGET lemmon <cr>
```

"lemmon" will be removed from the dictionary. Their code segments are also discarded. If the word "lemmon" does not exist, the informative message

```
lemmon ?
```

will appear, without any words being removed from the dictionary. FORGET should be used cautiously, for it can produce undesirable and/or fatal effects if two or more vocabularies are intertwined.

Glossaries

In the following glossaries, the words are described in a semi-alphabetic order. The following notation is used for each entry:

```
nnnn    params nnnn params      ---> values    (chars)       description
```

where:

nnnn	is the name of the word being described.
params	are the word's parameters (parameters which appear before the word are assumed to be on the stack, while parameters appearing after the word are parsed from the input buffer).
values	are the value returned.
chars	are characteristics of the word.
description	is a verbal description of what the word does.

Not every word will have parameters/values/characteristics.

Parameter notation

c	A seven-bit ASCII character code.
f	A Boolean flag. FALSE=0, TRUE=non-zero. All words which return a flag (0= , <= , etc.) return FALSE=0 and TRUE=1.
e g m n o p q r s t u v w x y z	A 16-bit integer.
nnnn	a name of a word witch consists of 1-255 non-blank character delimited on the right by a blank or a Carriage Return.
ssss	A string consists of 1-255 character.

Characteristics

C	The word may only be used within a Colon definition.
P	The word has its Precedence bit set, and is executed immediately, even in compilation mode.
V	The word is a Variable. The default value is shown, following the V, in decimal notation.
X	The word is a Constant.

SL5 Glossary

! `m p !`
Store m at address p.

@ `p @ ---> m`
Fetch the contents of address p, and put it on the stack.

@X `p @X ---> m`
Fetch the contents of the word at address p, and reverse the order of the bytes before placing it on the stack.

" `" c ---> n (P)`
The ASCII value of the character that follows is returned.

' `' nnnn ---> p (P)`
A compiler directive that fetches the address of the parameter field of the word nnnn. The parameter field is the first word in a colon definition after the code field.

'B `'B nnnn ---> addr (P)`
Returns the execution address of the word nnnn. It is similar to ' except that the address is 3 less.

'S `'S nnnn ---> addr (P)`
Returns the address of the symbol table entry of the word nnnn.

(`( ssss ) (P)`
Ignore a comment that will be delimited by a right parenthesis or a Carriage Return.

+ `m n + ---> p`
16-bit signed integer addition. p = m + n

- `m n - ---> p`
16-bit signed integer subtraction. p = m - n

-- `m n -- ---> p`
16-bit signed integer subtraction. p = n - m

***** `m n * ---> p`
16-bit signed integer multiply. p = m * n

/ `m n / p`
16-bit integer divide. The quotient is truncated to 16 bits, and the remainder is dropped. p = m / n

/MOD `m n /MOD ---> p q`
16-bit integer divide. Quotient is on Top Of Stack, remainder below it (NOS). Remainder has sign of dividend. p = m / n

| `m n | ---> p`
16-bit logical inclusive OR of m and n.

& `m n & ---> p`
16-bit logical AND of m and n.

->L `m n ->L ---> p`
Logical right shift of m by n bits.

<-L `m n <-L ---> p`
Logical left shift of m by n bits.

1+ `m 1+ ---> n`
Add one to the value on Top Of Stack.

1+! `p 1+!`
Increment the value stored at p by one.

1- `m 1-  ---> n`
Subtract 1 from m.

1-! `p 1-!`
Decrement the value stored at p by one.

+! `m p +!`
Add the integer value m to the value stored at address p.

, `m ,`
Store m into the dictionary, at the location specified by the dictionary pointer (DP). Increment the dictionary pointer by 2.

-ROLL `n -ROLL`
Move the Top Of Stack into the nth position in the stack. The rest of the stack moves up. (3 -ROLL = ROT ROT)

[`[`
Switch STATE from compile to interpret (see] below). Words in a colon definition up to] are interpreted, instead of compiled.

] `]`
Switch STATE from interpreter back to compile. Words between and] are interpreted, even though the machine state is compile.

. `m .`
Display the Top Of the Stack as a 16-bit signed number, according to the current radix specified by the variable BASE.

? `p ?`
Display the contents of address p on the console, according to the current radix.

2DROP `m n 2DROP`
Remove the top two stack elements.

2DUP `m n 2DUP  ---> m n m n`
Duplicate the top two stack elements.

2SWAP `m n p q 2SWAP  ---> p q m n`
Swap the top two pairs of stack elements.

: `: nnnn`
Create a new word (entry) in the dictionary with the label nnnn. Set the system variable STATE to compile (1). Subsequent words up to ; will be compiled into the dictionary, instead of executed (interpreted).

; `;  (CP)`
Terminate a colon definition, update the dictionary linked list, and set the variable STATE to 0 (interpret mode).

;: `;:  (CP)`
Used to create a user-defined data or code structure. ;: causes the words between ;: and the next ; to be compiled, and their address specified in the code segment of any words created by nnnn.

;CODE `;CODE  (CP)`
;CODE has the same effect as ;: above, except that the code defined is assembly language, instead of SL5.

0< `m 0<  ---> f`
f is TRUE if m is negative.

0= `m 0=  ---> f`
f is TRUE if m = 0.

0> `m 0>  ---> f`
TRUE if m is greater than zero.

< `m n <  ---> f`
f is returned as TRUE if m is less than n.

<= m n <= ---> f
f is TRUE is m is less than or equal to m.

= m n = ---> f
TRUE if m is equal to n.

<> m n <> ---> f
f is TRUE if m is not equal to n.

> m n > ---> f
TRUE if m is greater than n.

>= m n >= ---> f
TRUE if m is greater than or equal to n.

ABORT ABORT
Common word for fatal error exits. The message "ABORT" is displayed before RESTART is executed.

ABS m ABS ---> n
Absolute value of m.

ARRAY n ARRAY nnnn
Define a word nnnn and allocate n words of storage. Subsequent references to nnnn cause the Top Of Stack to be added to the base address of the defined storage area, and placed on the Top Of the Stack.

ASSEMBLER ASSEMBLER
Set the context vocabulary to the ASSEMBLER vocabulary.

B! m p B!
Store the byte (low 8 bits of m) at address p.

B@ p B@ ---> m
Fetch the contens of the byte at address p and put it on the top of the Stack.

B, m B,
Store the byte m into the dictionary at the address of the dictionary pointer DP. Increment the dictionary pointer DP by one.

B. m B.
Display m as an unsigned 8-bit hexadecimal number.

BARRAY n BARRAY nnnn
Same as ARRAY, with the exception that the cells are one byte in length, instead of two bytes.

BASE BASE (V16)
A system variable that contains the current conversion radix.

BEGIN BEGIN..END (CP)
A compiler directive that defines the start of a loop. The word END tests the Top Of the Stack, and branches back to BEGIN if the Top Of Stack is FALSE (=0), or to the word following END if TRUE (=1). The word WHILE can be used for a pretest, as in BEGIN..WHILE..REPEAT.

BLANK n q BLANK
Fill n bytes of memory, starting at address q, with the space character (20H).

BMOVE p q m BMOVE
Move m bytes starting at address p into the area specified by address q.

BSWAP n BSWAP ---> m
Swap the order of the two bytes of n and place the result on the stack.

CASE CASE (CP)

```
          1 =: ...  ;;
          2 =: ...  ;;
          3 =: ...  ;;
 DUP NOCASE =: ...  ;;
```

CASEND
the CASE statement is a one-of-n execution select. it deviates from the standard Forth CASE statement which is really a nested IF statement. NOCASE is executed if the Top Of Stack does not match any of the values in the CASE body.

CASEND CASEND (CP)
Terminates a CASE statement sequence.

CHECK CHECK
Test for return or parameter stack underflow. ABORT is called on underflow.

CIN CIN ---> c
Read a character from the keyboard, and return its ASCII value c. The character is checked before returning it, to see if it is one of the special characters.

CIS CIS
The keyboard is checked for input and, if a character has been typed, the character is read and examined, to see if it is one of the special characters. This routine is very useful as a real-time breakpoint utility.

CODE CODE nnnn
Define a word nnnn which is written in assembly language, using the SL5 system assembler. Creates a dictionary entry, and sets the context vocabulary to ASSEMBLER. (See the Assembler glossary.)

COM m COM ---> n
One's complement of m.

COMPILE COMPILE nnnn (P)
Compile the address of the word nnnn into the code segment. This is useful for compiling words which have their precedence bit set by IMP or IMMEDIATE.

CONSTANT n CONSTANT nnnn
Create a word nnnn which, when executed, places the value n on the Top Of the Stack.

CONTEXT CONTEXT (V)
A pointer to the CONTEXT vocabulary, where dictionary searches begin. (See the word "FIND".)

COUNT p COUNT ---> m n
Returns the byte address m and the length n of the ASCII string at address p. COUNT assumes that the first byte at address p contains the string length.

COUT c COUT
the character whose ASCII value corresponds to c is displayed on the console.

CR CR
Output a Carriage Return - Line Feed sequence to the output file designated by OUTFILE.

CTYPE m n CTYPE
Output a string of n characters, starting at address m, to the console.

C" C" ssss" (P)
Output the string ssss to the console. One space must precede the string.

CURRENT CURRENT (V)
A variable pointing to the CURRENT vocabulary, which is the vocabulary which new words are added to.

CVOC CVOC (V)
CURRENT vocabulary pointer.

DECIMAL DECIMAL
Set the number radix to base 10.

DEFINITIONS nnnn DEFINITIONS
Set the current vocabulary to nnnn. New words will be added to the nnnn

vocabulary.

DELIMITER DELIMITER (V32)
A variable that contains the current character used as a string delimiter by string routines (WORD).

DO m n DO ... LOOP CP (CP)
m n DO ... o +LOOP
Defines an iterative loop that is executed m - n times. The upper limit m is one greater than the terminal value. o +LOOP increments the loop index by o each time through the loop. o does not have to be an integral value of the count.

DP DP (V)
A variable pointing to the next available word in the dictionary or code segment.

DP+! m DP+!
Increment the value of the dictionary pointer (DP) by m and store the result in DP.

DROP m DROP
Remove the top element of the stack.

DUP m DUP ---> m m
Duplicate the top element of the stack.

ELSE IF..ELSE..ENDIF (CP)
Defines the start of the FALSE part of a IF..ELSE..ENDIF clause.

END BEGIN ... END (CP)
Marks the end of a BEGIN..END loop. The Top Of Stack is tested upon execution of END for a zero value (FALSE). If it is zero, execution resumes with the word following the corresponding BEGIN statement. If it is non-zero (TRUE), execution continues with the word following the END statement.

ENDIF IF..ELSE..ENDIF (CP)
Marks the end of a IF statement.

EXECUTE p EXECUTE
A word used in the outer interpreter to execute/compile the word pointed to by address p, depending on the value of the variable STATE and the precedence bit of the word. p is a pointer to the word's symbol table entry.

EXIT DO-EXIT-LOOP (C)
Force termination of the current loop on this iteration, by increasing the loop index to the terminal value.

FILL m p n FILL
Store the value n into m bytes, starting at address p.

FIND FIND ---> n 1 / 1
Search the symbol table starting at CONTEXT for a match with the string at the dictionary pointer. The string is usually left by WORD. If found, return the code address, and a 1 (=TRUE) on Top Of Stack. Otherwise, return a 0 (=FALSE), to flag that the symbol is not in the dictionary.

FLOAD FLOAD nnnn (P)
Open the file nnnn and switch the input stream to disk input from that file. On EOF or [END-OF-FILE], return to the console for input. Generally used to compile code edited into a source file and saved on disk.

FORGET FORGET nnnn (P)
Delete nnnn and all subsequent dictionary entries.

FORTH FORTH
Sets the CONTEXT vocabulary to FORTH, which is the central vocabulary.

GCH GCH ---> c
Read a character from the system input file, and return its ASCII character code.

GO n GO

Branch to the address n. If n is the address of a colon definition or a code definitions which returns to the inner interpreter, GO functions more as an assembly language "CALL" statement.

GOQIAD GOQIAD (V)
A variable pointing to the outer interpreter word. Can be set so that any word can be branched to after a RESTART, other than the outer interpreter (INTRLP).

GO-OPSYS GO-OPSYS
Exit to the CP/M operating system.

HERE HERE ---> p
Returns the value of the dictionary pointer (DP).

HEX HEX
Set the radix to base 16 (hexadecimal).

I DO ... I ... LOOP (C)
Returns the value of the innermost loop index.

IF IF..ELSE..ENDIF (CP)
The start of a conditional clause. Code is compiled to test the Top Of the Stack, and branch to the true part (IF), the false part (ELSE), or ENDIF.

IMMEDIATE IMMEDIATE
Sets the precedence bit of the word which was just defined.

IMP IMP nnnn
Mark the word nnnn as immediate. nnnn will be executed whenever it is encountered. Specifically, it is executed when STATE=1, or during compilation.

INFILE INFILE
the system input file's fib. This can be altered so the input (such as GCH) will be taken from a device or a disk file.

INFOF INFOF (V)
Information control byte. When bit 0 is on, the system prints an error message when a word is redefined. Bit 2 on enables listing of source text during FLOAD. INFOF defaults to all bits being set.

ININIT ININIT
Reset the system input file (INFILE) to the console.

INTRLP INTRLP
The outer interpreter loop.

J J (C)
Returns the value of the next outer DO..LOOP index.

K K (C)
Returns the value of the third DO..LOOP index.

LINK n LINK
Adds a new word to the symbol table, and the active vocabulary chain. The name of the word is pointed to by dictionary pointer (DP) and n is the execution address of the word. The flag byte is initialized to 0.

LIT LIT
A word that is compiled into the code segment before every literal. Contains code to push the next word onto the stack at execution time, and move the interpreter pointer IP to the following word.

LITERAL n LITERAL
If STATE=1 (compilation mode), the word LIT is compiled along with n, so that n is placed on the stack upon execution of LIT. If STATE=0 (interpreter mode), n is left on the stack.

LOOP DO ... LOOP (CP)
Increment the loop index by 1, and exit from the loop if the index is greater than or

equal to the limit.

+LOOP m +LOOP (CP)
Add m to the loop index. The loop is exited when the value of the index is equal to or greater than the limit.

MAX m n MAX ---> q
Leaves the greater of the two signed integers m and n on the stack.

MIN m n MIN ---> q
Leaves the lesser of the two signed integers m and n on the stack.

MINUS m MINUS ---> n
Leave the two's complement of m on the stack.

MOD m n MOD ---> r
Remainder of m / n with same sign as m.

NOCASE CASE-NOCASE-CASEND
An otherwise branch for the CASE statement. See the word "CASE".

NOT m NOT ---> f
Equivalent to 0=.

NUMBER NUMBER ---> n 1 / 0
the character string left by WORD is converted to a number according to the current radix defined in the variable BASE. NUMBER converts signed and unsigned 16-bit integers. The result is left on the stack with a 1 (=TRUE) on Top Of the Stack if the conversion succeeds. A 0 (=FALSE) is left on the stack if the number cannot be converted with the current radix.

OCTAL OCTAL
Set the radix to base 8 (octal).

OUTFILE OUTFILE
The system output file's fib. This can be changed so as to re-route all system output to a device or a disk file. OUTFILE defaults to the console.

OUTINIT OUTINIT
The system output file (OUTFILE) is set to the console.

OVER m n OVER ---> m n m
Duplicate the second stack element, and put it on the Top Of Stack.

PICK m PICK ---> n
Copy the mth stack value onto the Top Of the Stack (2 PICK = OVER).

R> R> ---> n
Pop the top value from the return stack, and push it onto the parameter stack.

>R m >R
Move the Top Of the Stack to the top of the return stack.

RECURSE RECURSE (CP)
Causes the word under construction to be executed at execution time.

REPEAT BEGIN..WHILE..REPEAT (CP)
Compile an unconditional jump back to BEGIN. See BEGIN.

RESTART RESTART
The stacks are cleared, STATE is set to interpret, INFILE and OUTFILE are set to the CRT, and the outer interpreter is invoked.

RESTARTAD RESTARTAD (V)
A variable pointing to the address of the system RESTART routine.

RMOVE p q m RMOVE
Move m bytes in memory from address p to address q. The move is carried out from the last byte in the vector to the first.

ROLL n ROLL

Fetch the nth stack element, and push it onto the Top Of the Stack.

ROT `m n p ROT ---> n p m`
Rotate the top 3 stack elements.

RP@ `RP@ ---> n`
Push the value of the return stack pointer onto the parameter stack.

RP! `n RP!`
Set the return stack pointer to n.

RSIZE `RSIZE (K)`
The size of the return stack which is implemented as a BARRAY in SL5.

SET `m p SET nnnn`
Define a word nnnn which, when executed, will cause the value m to be stored at address p.

SP@ `SP@ ---> n`
Push the value of the parameter stack pointer onto the stack.

SP! `n SP!`
Set the parameter stack pointer to n.

SPACE `SPACE`
Output a space character (20H) to the system output file.

SPACES `n SPACES`
Output n space characters to the system output file.

SWAP `m n SWAP ---> n m`
Exchange the top two elements on the stack.

SSIZE `SSIZE (K)`
Returns the size of the parameter stack which is implemented as a BARRAY in SL5.

STATE `STATE (V0)`
A variable that determine whether a word is interpreted (STATE=0) or compiled (STATE=1). Words that have their precedence bit set are executed when STATE=1.

SYMTP `SYMTP (V)`
The address of the top of the symbol table.

SYMPTR `SYMPTR (V)`
A pointer to the last entry in the symbol table.

TCH `c TCH`
the ASCII character associated with c is sent to the system output file.

TYPE `m n TYPE`
Output a string of n characters starting at address m to the system output file.

T" `T" ssss"`
Output the string ssss to the system output file. One space must precede the string.

U< U<= U> U>+ `m n U< ---> f`
Unsigned 16-bit integer comparisons. Used to compare addresses and other 16-bit integers that are treated as unsigned numbers.

U/MOD `m n U/MOD ---> p q`
16-bit unsigned integer divide. Quotient is on Top Of Stack, remainder below it (NOS). p = m / n

UPPER `UPPER (V1)`
When UPPER is set to 1 (=TRUE), the routine WORD converts characters to upper case. UPPER defaults to 0 (=FALSE).

UNDEFINED UNDEFINED
The string left by WORD is displayed, along with the undefined message on the console. RESTART is then executed.

VARIABLE m VARIABLE nnnn
A word that creates a dictionary entry for the word nnnn, allocates a word in memory, and initializes that memory word to n. See @ and !.

VLIST VLIST
List the dictionary, starting at CONTEXT. Every entry has its execution address, flag byte, and name displayed on a separate line.

VOCABULARY VOCABULARY vvvv
Creates a vocabulary chain vvvv with the head linked to the current vocabulary. vvvv DEFINITIONS makes vvvv the current vocabulary into which new definitions are added.

WHILE BEGIN..WHILE..REPEAT (CP)
A pretest for loop iteration. A TRUE Top Of Stack (<>0) causes the words between WHILE and REPEAT to be executed. REPEAT generates an unconditional jump back to BEGIN. A FALSE Top Of Stack (=0) results in a jump to the word following REPEAT.

WORD WORD
Scan the input buffer for the next token, which is the string of characters up to a delimiter. The delimiter is the character in the variable DELIMITER. WORD resets DELIMITER to space (20H). The string is stored in a system area, with the size, followed by the string.

X. a X.
Print the top element on the stack as an unsigned 16-bit integer, in base 16 (hexadecimal).

X| m n X| ---> o
Logical exclusive OR of m and n.

ZIN p ZIN ---> n
Read value n from I/O port p (Zilog Z-80 version only).

ZOUT n p ZOUT
Output n to I/O port p (Zilog Z-80 version only).

Assembler Glossary

BEGIN, BEGIN,
Mark the beginning of an assembler loop.

CODE CODE nnnn
Create a dictionary entry for a code word nnnn, and set the CONTEXT vocabulary to ASSEMBLER.

EDOC EDOC
Terminate a code definition, and set the CONTEXT vocabulary to the CURRENT vocabulary.

ELSE, ELSE,
Mark the beginning of the false part of an "IF," construction. The code which follows is executed only if the condition specified at the "IF," is false (=0).

END, END,
Mark the end of a conditional loop. If cc is TRUE (<>0) during execution, the loop will not be re-executed.

ENDIF, ENDIF,
Mark the end of a conditional forward branch.

IF, cc IF,

Assemble a conditional branch which will execute the code following "IF," only if cc is TRUE (<>0); otherwise, a branch to the code following "ENDIF," or "ELSE," is made.

REPEAT, REPEAT,
Mark the end of an unconditional loop.

File System Glossary

[END-OF-FILE] [END-OF-FILE]
Terminate compilation of a file which was loaded with FLOAD.

BUFAD fib^ BUFAD ---> n
Return the buffer address associated with fib^.

BUFLEN fib^ BUFLEN ---> n
Return the address of the buffer length associated with fib^.

BUFP fib^ BUFP ---> n
The address of the buffer pointer associated with fib^ is returned.

BUFSIZE BUFSIZE ---> n (K)
A predefined constant, whose value is the size of all the buffers.

CLOSE fib^ CLOSE
Close the file designated by fib^ which was previously opened or created.

DELETE fib^ DELETE
The file designated in fib^ is removed from the directory.

EOF fib^ EOF ---> f
The Boolean flag f is returned as TRUE (=1) if the end-of-file was reached on the file specified by fib^. If the end-of-file has not been processed yet, the FALSE value (=0) is returned.

EOFCHR EOFCHR ---> n (K)
A constant which determines the end-of-file character which RCHlooks for.

FALLOC FALLOC nnnn
Create a fib which can be subsequently referenced by the name nnnn.

FCB fib^ FCB ---> n
Return the address of the File Control Block (FCB) associated with fib^.

FLOAD FLOAD filename
Load the file specified by filename from the disk.

FLUSH fib^ FLUSH
Flush the buffer associated with fib^. The buffer is written out to disk if the buffer pointer associated with the file is non-zero; otherwise, no action is performed.

NAMIT fib^ NAMIT nnnn
The file name associated with fib^ is set to nnnn. nnnn is in the form of d:ffffffff.eee, where "d" is the drive (A-P) followed by a colon (":"). "ffffffff" is the file name, and "eee" is the extension name. The "d" and the "eee" fields are not mandatory. Serial devices are specified when "ffffffff" is one of the following:

```
#CRT -- Console/keyboard
#LST -- Printer/keyboard
```

OPENR fib^ OPENR
Open the file designated by fib^ for reading.

OPENW fib^ OPENW
Create a new file, which is designated in fib^, and open it for writing. The previous version of the file is deleted from the directory, if it existed. If the operation fails, an error message is issued, and an abort to RESTART is made.

RBYTE fib^ RBYTE ---> n
The next byte is read from the file designated by fib^, and returned.

RCH fib^ RCH ---> c
The next ASCII character is read from the file described by fib^.

READ fib^ READ
The next sector is read from the file designated by fib^ into the file's buffer.

RENAME ofib^ nfib^ RENAME
The file which is described by ofib^ is renamed to the file specified nfib^ in the directory.

RESET fib^ RESET
the file associated with fib^ is reset. This means that the file is rewound, and the end-of-file flag is reset.

WBYTE n fib^ WBYTE
The byte n is written onto the file described in fib^.

WCH c fib^ WCH
The ASCII character c is written to the file designated by fib^.

WRITE fib^ WRITE
The buffer associated with fib^ is written onto the file designated by fib^. If the file is a serial device, BUFP should contain the size of the buffer which is to be written out.

Debug Glossary

DUMP s e DUMP
Dump memory from s to e on the terminal (or which device is selected with OCHAN#). The dump is displayed as lines composed of an address, followed by the 16 bytes which start at that address in their numeric value, followed by the 16 bytes displayed in their ASCII code. All numbers are displayed in hexadecimal notation.

MODIFY s MODIFY
Modify memory, starting at address s.

PSDMP PSDMP
Dump the data stack. The Top Of Stack is displayed first, and the bottom is displayed last.

RSDMP RSDMP
Dump the return stack. The top of the return stack is displayed first, and the bottom of the return stack last.

SYMDUMP SYMDUMP
Dump the symbol table (the dictionary), starting from context, proceeding to the first definition. Each symbol is displayed on one line, with the address of the entry in the symbol table first, followed by the link being displayed as relative to SYMTP, and its absolute location in memory. The address of the code/colon body is then displayed, followed by the length of the symbol's name and the symbol. All numbers are displayed in hexadecimal notation.

SYM' SYM' nnnn
Display the word nnnn as it would be shown in SYMDUMP.

BREAK *BREAK* (C2)
Cause a breakpoint to occur.

UB *UB*
Return from a breakpoint.