

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 Z80 microprocessor, and to the 8080 microprocessor. CP/M is a registered trademark of Digital Research of Pacific Grove CA. Z80 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 discette, 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 times remain with SuperSoft

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

Application Note #1
 (Z80 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
         OUT    (0D1H),A      ;SEND A BLANK TO 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 & 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 PORT D1.
         RET                               ;RETURN TO CALLER.
EXAD3:   IN      A,(0D2H)     ;READ FROM PORT D2.
         RET
  
```

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

```

F605 CONSTANT EXAD2    ( ADDRESS OF EXAD2 )
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 ".".

The Ascii Character Set

<u>hex</u>	<u>dec</u>	<u>char</u>	<u>hex</u>	<u>dec</u>	<u>char</u>	<u>hex</u>	<u>dec</u>	<u>char</u>	<u>hex</u>	<u>dec</u>	<u>char</u>
0	0	null	20	32		40	64	@	60	96	`
1	1	soh	21	33	!	41	65	A	61	97	a
2	2	stx	22	34	"	42	66	B	62	98	b
3	3	etx	23	35	#	43	67	C	63	99	c
4	4	eot	24	36	\$	44	68	D	64	100	d
5	5	enq	25	37	%	45	69	E	65	101	e
6	6	ack	26	38	&	46	70	F	66	102	f
7	7	bell	27	39	'	47	71	G	67	103	g
8	8	bs	28	40	(48	72	H	68	104	h
9	9	ht	29	41)	49	73	I	69	105	i
A	10	lf	2A	42	*	4A	74	J	6A	106	j
B	11	vt	2B	43	+	4B	75	K	6B	107	k
C	12	ff	2C	44	,	4C	76	L	6C	108	l
D	13	cr	2D	45	-	4D	77	M	6D	109	m
E	14	so	2E	46	.	4E	78	N	6E	110	n
F	15	si	2F	47	/	4F	79	O	6F	111	o
10	16	dle	30	48	0	50	80	P	70	112	p
11	17	dcl	31	49	1	51	81	Q	71	113	q
12	18	dc2	32	50	2	52	82	R	72	114	r
13	19	dc3	33	51	3	53	83	S	73	115	s
14	20	dc4	34	52	4	54	84	T	74	116	t
15	21	nak	35	53	5	55	85	U	75	117	u
16	22	syn	36	54	6	56	86	V	76	118	v
17	23	etb	37	55	7	57	87	W	77	119	w
18	24	can	38	56	8	58	88	X	78	120	x
19	25	em	39	57	9	59	89	Y	79	121	y
1A	26	sub	3A	58	:	5A	90	Z	7A	122	z
1B	27	esc	3B	59	;	5B	91	[7B	123	{
1C	28	fs	3C	60	<	5C	92	\	7C	124	
1D	29	gs	3D	61	=	5D	93]	7D	125	}
1E	30	rs	3E	62	>	5E	94	^	7E	126	~
1F	31	us	3F	63	?	5F	95	_	7F	127	rubout

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 '80s. 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 end 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 OEMs 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 Z80 could be switched to a 6809 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 AM9511 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 Fortran 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 '80s. 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

Getting Started

SL5 is distributed as a set of CP/M text and COM files on a single diskette. It's 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 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 cntl-C or reset the system to bring up CP/M
2. Enter SL5<cr>

The prompt > should appear on your screen. If it doesn't 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 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 of 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 SYSMAX 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.

Handwritten notes, possibly bleed-through from the reverse side of the page. The text is extremely faint and illegible. Some faint markings resembling the number '12' and a signature are visible on the right side of the page.

Table of Contents

Tutorial

1.	Introduction	1
	DECIMAL	
2.	Numbers	1
3.	Simple Stack and Arithmetic Operators	1
	DUP SWAP + - *	
4.	Displaying a Message	2
	T" " CR	
5.	Colon Definitions...Creating New Words	2
	; Word Names	
6.	Program Control Words	3
	BEGIN END IF ELSE ENDIF DO..LOOP FORGET	
7.	Constants, Variables	4
	CONSTANT VARIABLE @ (fetch) ! (store) B@ B!	
8.	Writing a Program...a Simple Example	5

Reference

1.	Introduction	1
2.	The Stack Stack Operators	1
3.	Numbers	2
4.	Variables and Constants Constants Variables Memory operators	3
5.	Arithmetic and Logical Operators	4
6.	Conditional Operators	6
7.	Outer Interpreter	7
8.	: Definitions Inner Interpreter	7
9.	Branching IF..ELSE..ENDIF CASE..NOCASE..CASEND	8
10.	Loops BEGIN..END BEGIN..WHILE..REPEAT DO..LOOP +LOOP, EXIT Loop indexes (I,J,K) RECURSE	9
11.	Arrays ARRAY BARRAY	12
12.	I/O Introduction INFILE, OUTFILE FALLOC, NAMIT OPENR, OPENW, FLUSH, CLOSE GCH, TCH, T", TYPE CIN, COUT, C" RCH, WCH RBYTE, WBYTE READ, WRITE DELETE, RENAME ININIT, OUTINIT EOF WORD Numeric input Numeric output FLOAD, [end-of-file]	12

13.	User Defined Code Structures17
	;CODE	
14.	The Dictionary18
	Symbol Table	
	Vocabularies	
	Chaining	
	FIND	
	FORGET	
	', 'B, 'S, COMPILE	
15.	CODE Words19
16.	System Variables20
17.	Error messages.21

Assembler

1.	Introduction	1
2.	Execution of CODE Words	1
3.	Creation of CODE Words	1
	1. Using the Assembler1
	2. Exiting from a Code Word2
	3. Branching Within CODE Definitions3
	Forward Branching	
	Looping	
4.	Assembler Mnemonics	7
5.	Register Usage.11
6.	Examples of CODE Definitions.11

Debug

1.	Introduction.1
2.	DUMP - Memory Dump.1
3.	MODIFY - Memory Modify.1
4.	PSDMP, RSDMP - Stack Dumping.1
5.	*BREAK*, *UB* - Breakpointing3
6.	SYM', SYMDUMP - Dictionary Examination.3

CP/M Interface

1. Introduction	1
2. Loading SL5 object files	1
3. FLOAD	1
4. CALLCPM	2
5. Serial I/O	2
6. Disk I/O	2
7. OPENR, OPENW	3

Object Modules

1. Introduction	1
2. Compiling a Subset of the SL5 Kernel.	1
Outer Interpreter	
Compiler	
Console I/O	
File System	
User I/O	
The Symbol Table	
Deleting a Section of the Kernel	
3. Generating a ROM Based Controller Program	4
4. Generating a RAM COM Module with SYSMAKE	6
5. SYSMAKE Errors and Parameters	7
6. Generating a COM module with COMMOD	7
Creating Disk COM File	
Creating a COM File With an Initialization Routine	
Creating a COM File Without a Symbol Table	

Structure

1.	Introduction	1
2.	Memory Organization.	1
3.	Compilation of Words	3
1.	Colon Definitions.	3
1.	Literals	4
2.	T"	4
3.	Branching	5
	IF..ELSE..ENDIF	
	BEGIN..END	
	RECURSE	
	;: and ;CODE	
2.	CODE Definitions10
3.	CONSTANTs.10
4.	VARIABLEs.11
5.	Arrays11
4.	The Dictionary12
1.	The CURRENT & CONTEXT Pointers12
2.	Vocabularies12
	Their Internal Structure	
	. Vocabulary Chaining	
3.	Dictionary Reduction15

Glossaries

SL5 Glossary	1
Assembler Glossary17
File System Glossary18
Debug Glossary20

574

574

Tutorial

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 symbols <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 Glossary.

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: .

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.

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 + .

Plus 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.

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.

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 colon(:) 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 colon, a one word name, a list of already defined words or numbers, and a terminating semi-colon. 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 invokes 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.

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 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: : DOTEEST 0 DO I . LOOP CR ;

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

Type: 7 DOTEEST

For a still more general DOTEEST put both DO parameters on the Stack.

Type: : DOTEEST DO I . LOOP CR ;

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

Type: 8 1 DOTEEST

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 DOTEEST

The second version of DOTEEST 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 DOTEEST versions two and one. Test it.

Type: DOTEEST

The error message DOTEEST ? indicates that DOTEEST 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.

Constants and 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 rommable 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 prededined 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 0 to 99 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).

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 topmost 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 PROCSOPTION is the heart of ICP. It must do the file I/O and displays as requested, and leave 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 PROCSOPTION. 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:

```
: PROCSOPTION ( test stub for debugging )
  T" PROC " DUP . 4 = IF
    1 ELSE 0 ( leave a 1 for terminate )
  ENDF ;
```

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 ENDF 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 fine without further tweaking. Even faster execution speed can be obtained 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 then loaded using the predefined word FLOAD. See the Reference Section for more information about loading. The SYSGEN program included with SL5 can be used to create a CP/M COM module or a stand-alone ROMMABLE module. See the SYSGEN 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 ■

5

5

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 Glossary. 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 illustrate the use of SL5 operators from the console interpreter (see example 1 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 1:

```
>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 reentrant and recursive code, and clean handling of interrupts.

```
>5 .
5>5 DUP . .
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 top of stack (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	Drop TOS & TOS-1	ab	abab
2DROP	Drop TOS & 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 run time routines. The usual result is truncation.

Base

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's 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 run time.

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 SYSGEN 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 addr q
!	m p !	Store m at address of p
B@	q B@	Fetch 1 byte from memory
B!	m p B!	Store byte m at addr of p
@X	q @X	Fetch a word and swap bytes
l+!	p l+!	increment contents of p by l
l-!	p l-!	decrement contents of p by l
+!	m p +!	add m to contents of p

```

>10 VARIABLE TEST
>TEST .
4000 > ( address of TEST returned)
>TEST @ .
10 >5 TEST !
>TEST @ .
5 >TEST l+!
>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
l+	m l+	add 1 to TOS
-	m n -	subtract n from m (m-n)
l-	m l-	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 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 (m,n) on TOS
ABS	m ABS	Leave absolute value on TOS
MINUS	m MINUS	Negate by taking 2's compl
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	rt 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 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
U<=	m n U<=	Unsigned LT or equal

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 linked 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 : definitions 8.). 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 S: 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 S; which restores the context.

EXAMPLE:

```
: 2TIMES 2 * ;
```

\$:
addr LIT
00C2
addr *
addr \$;

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 : definition. It maintains an instruction counter (IP). The IP is similar to the PC 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

The words IF ELSE and ENDIF are compiled into test and branch instructions. In the compiled code SIF 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

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 doesn't 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 IP set to the word after CASEEND. The NOCASE clause if present, is executed when the TOS does not match any of the test values.

EXAMPLE: keyboard input

```

: KEYIN
  ( 0D= <CR>, 08= backspace, 1B= 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+! ;;
  CASEEND ;

```

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 condition 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 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 results in a jump back to the word following BEGIN, and the loop is repeated.

```
: WAIT-FOR-CR BEGIN GCH OD = 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. REPEAT generates an unconditional jump back to BEGIN, and the TOS is tested again.

DO..LOOP

The DO 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, 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 LOOP's can be nested many levels deep but indexes are only provided for the three inner loops. See the Structure manual 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. Reentrant 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 Sysgen 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.

```
( assume 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 >
```

12.0 I/O

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 imbedded 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 buffers, 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 channel.

```
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 PENR 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
```

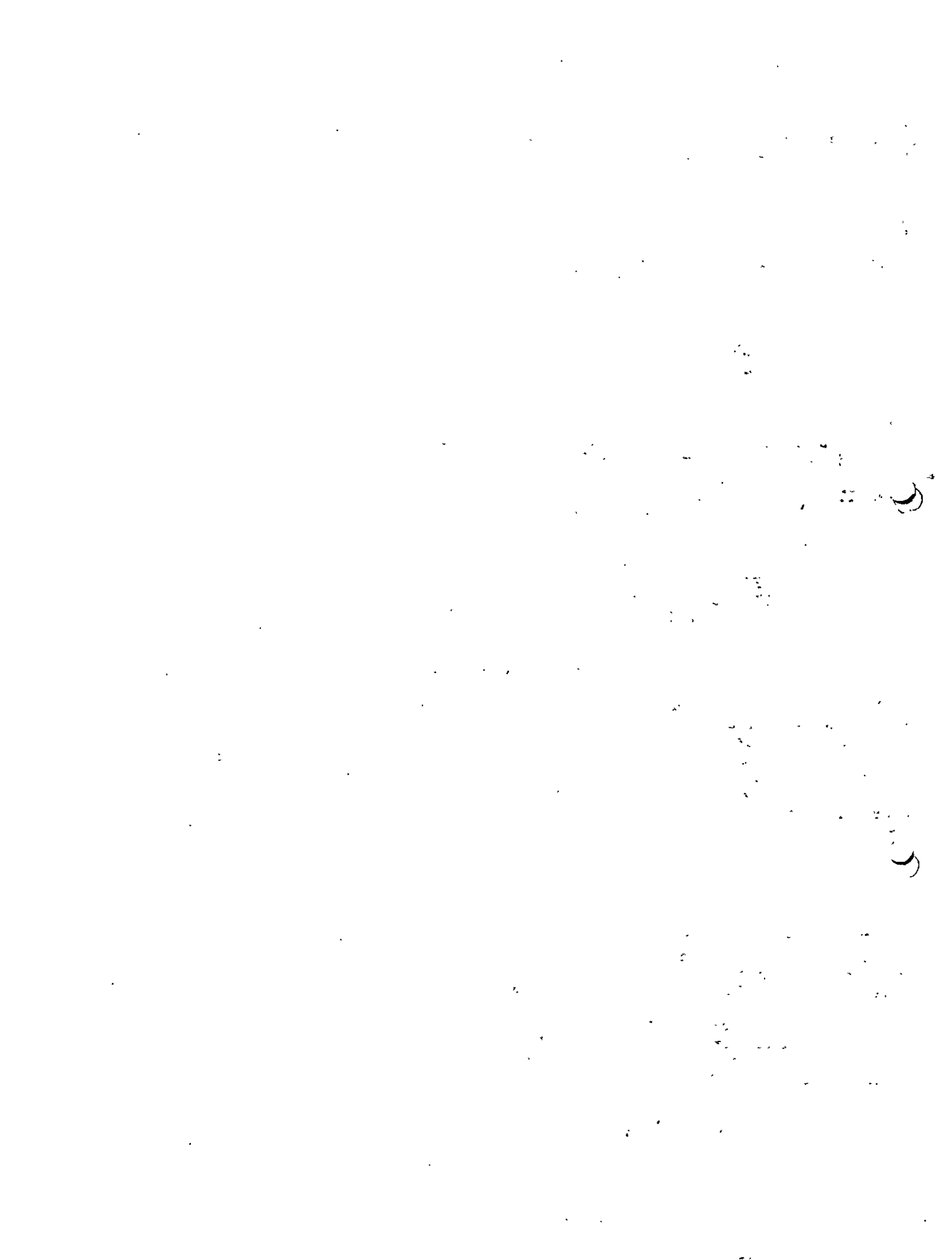
OPENR, OPENW, FLUSH, CLOSE

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. TCH and CLOSE should always be invoked before attempting to reuse 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 ASCII EOF character 1A 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. The 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. nnn 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
```



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 file is created 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

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.

ININIT, OUTINIT

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

EOF

The word EOF returns a 0 while there is more data in the file and a 1 when the physical end of file is reached. The character I/O words also return the ASCII end of file character 1A 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 blank (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 (1-7 chars) " cr
  WORD
  T" The current file is "
  HERE @ 1+ 0 DO
  HERE 1- 1 + B@ DUP TCH 1 NAME B! LOOP CR ;

>GET-NAME
Enter filename (1-7 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 Numbers above). 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.



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 : definitions allow. ;CODE creates new words that invoke assembly language sequences (see Assembly language section).

14.0 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 SYSGEN. 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 define 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 search fails. If a word is multiply defined in the dictionary the last definition is found.

FORGET

ssss FORGET erases all words in a vocabulary from the word ssss. This is a sequential erase. The vocabulary head is set to the word before ssss, 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 isn't recommended. 'B returns the address of the code field of a word. 'S is the same as 'B during code 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.

15.0 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 definition. This overhead ranges from 100 microseconds on an 8080 to a few microseconds on a 68000. Code words are difficult to write, debug, modify, and they are not machine independent (see ASSEMBLY language section).

16.0 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 vocabulary
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	addr. RESTART (SYSGEN)
GOQLAD	addr. outer interpreter (SYSGEN)
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 ON - REDEF message printed Bit 2 ON - Print source during FLOAD all bits default to ON.

17.0 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/O 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, BEGIN..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 ■

5

5

the
sto.
or.
(1)

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 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 Z80 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 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 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

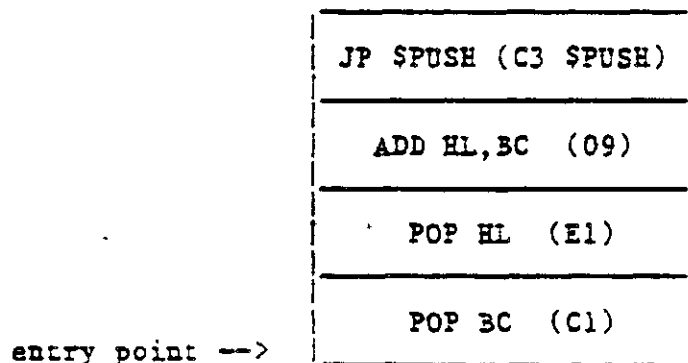
During the writing of CODE definitions, the machine instructions which compose the definition can be specified by placing the actual opcodes in the code segment or by using the SL5 assembler mnemonics. To place the opcodes in

code segment, the two words , and b, may be used. The word "B" will store a word (16 bits) in the code area while "B," will store a byte (the lower 8 bits of a word).
 NOTE: the word "B," reverses the bytes before storing them)

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

CODE + C1 B, E1 B, 09 B, C3 B, SPUSH, EDOC

C1, E1, and 09 are the Z80 opcodes which when executed will pop the BC and HL registers off of the stack and add them together. The "C3 SPUSH" 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.



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 SL5 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 SPUSH 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 set 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
          SPUSH 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 the assembler word IF, tests a Z80 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 at execution time will cause the machine instructions between the IF, and the ELSE, to be executed. If cc isn't true the code following the ELSE, (if present) will be executed. In both cases, the code following the ENDIF, is executed.

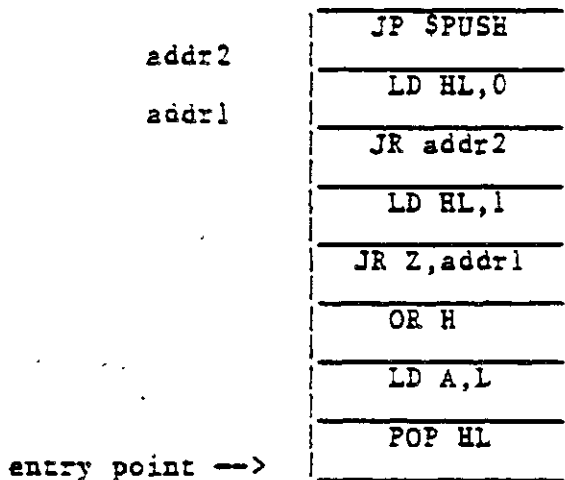


Figure 3-4. Memory diagram of the word 0<>.

```
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 SNEXT. 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 Z80 condition code (cc) rather than a boolean stack value before looping again. The code between BEGIN, and END, is executed repeatedly until the 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, SNEXT JP EDOC
    
```

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

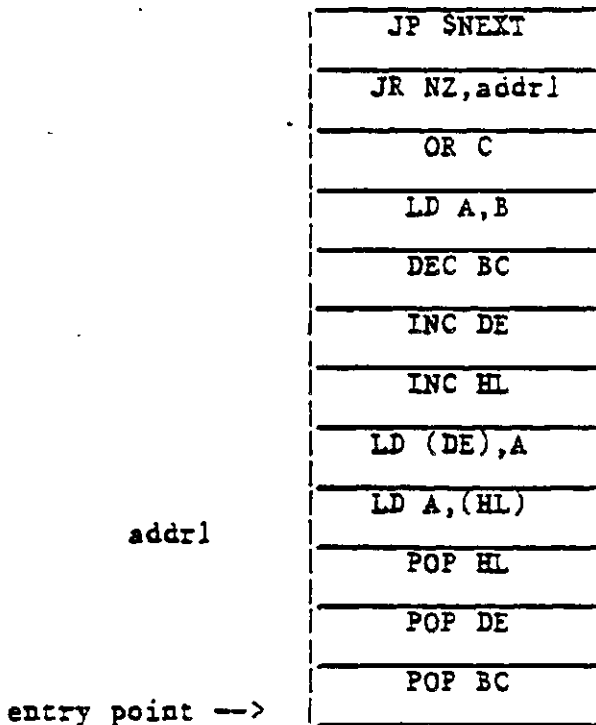


Figure 3-7. Memory image of the BMOVE example.

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,  SNEXT 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,
            SNEXT 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 appendix compares the SL5 assembler mnemonics with the ZILOG mnemonics. Refer to the "MOSTEK Z80 PROGRAMMING MANUAL" for more details on the operation and usage of the Z80 instruction set.

In the first part of this appendix the format of the SL5 mnemonics is discussed and in the second part the SL5 mnemonics are listed with their ZILOG 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 assembler clean. These differences are listed in figure A-1 (where two forms are shown, either may be used).

<u>ZILOG</u>	<u>SL5</u>
(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 Z80 registers {A,B,C,D,E,H,L,(HL)}.
- rp1 - any of the register pairs {BC,DE,HL,SP}.
- rp2 - any of the register pairs {BC,DE,HL,AF}.
- rp3 - any of the register pairs {BC,DE,IX,SP}.
- rp4 - any of the register pairs {BC,DE,IY,SP}.
- cc - any of the 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.

<u>ZILOG</u>	<u>SL5</u>
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
AND (IY+n)	n IY+ AND
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
CPDR	CPDR
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
IMO				IMO
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	rpl,nn		nn rpl	LD
LD	IX,nn		nn IX	LD
LD	IY,nn		nn IY	LD
LD	rpl,(adr)		adr rpl	LD
LD	IX,(adr)		adr IX	LD
LD	IY,(adr)		adr IY	LD
LD	(adr),rpl		rpl 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
RRCA				RRCA
RRD				RRD
SBC	r		r	SBC
SBC	n		n	SBC
SBC	HL, rpl	rpl	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 Z80 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* DUP + ;
CODE 2*  HL POP  HL HL ADD  $PUSH JP  EDOC
```

The next example tests the top of the stack to see if it is negative. If it is negative, the value 1 is returned, otherwise the value 0 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 is loaded into the register pair HL, otherwise a 0 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 port 40 continuously until a non-zero value appears. Then port 41 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  
          SPUSH JP    EDOC
```

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

Debug ■

5

5

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 final address. DUMP will then 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#EYLa^#V
0110 D5 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 the 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", 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#EYLa~#V
0100 11 22 33 44 <cr>
0110 D5 C3 04 01 C3 04 01 CD DC 01 82 67 CD 0C 01 84 DC..C..M...gM...
0110 40 41 42 43 44 45 46 1B <escape>
```

>120 100 DUMP <cr>

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

4. PSDMP and 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* and *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 stacks, 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 definition 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".

Handwritten notes and a small circular stamp or mark on the right side of the page.

Handwritten notes in the middle right section of the page.

Handwritten notes and a small circular stamp or mark on the right side of the page, similar to the one above.

Interface ■

CP/M Interface

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. Boot obtains the relocation address from CP/M at memory address 6, so it always uses all of available memory regardless of the system that SL5 was generated on.

The SL5 I/O subsystem (see 1.12 REFERENCE) 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 Guide 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 fatal errors, issues a message and reboots. The CP/M manual lists these error messages. SL5 traps and reports additional errors (see Reference 1.12).

Loading SL5 Object Files

SL5 object modules are created by SYSGEN and COMMOD as standard CP/M COM 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, a warm boot must be executed to go back to CP/M. The modules are loaded into the TPA at address 100H. SL5 does not use the area from 0 to 100H, 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 thorough knowledge of CP/M and SL5 I/O is required. The SL5 I/O subsystem should be sufficient for most applications.

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 DEV from CP/M
>FLOAD USER1.SL5 - load another program from SL5
> - continue executing in SL5
```

SL5 has 2 systems for creating new COM 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 SAVE utility stores the object file. The second system, SYSGEN, compiles the kernel and user code directly onto a disk file. SYSGEN is used to create new kernels, applications programs without the development words, and rommable object files (see Object Module section).

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 05 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 5 contains a jump to a location in CP/M that is the entry point of the program I/O handler. On return the A register contains a status byte. \$PUSH JP pushes the status on the parameter stack, and executes SNEXT- the inner loop of SL5.

Serial I/O

SL5 uses 4 CP/M serial character functions: check console status, read console character, write console character, and write character to the I/O device. These 4 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
```

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 described in the Reference section (1.12). The SL5 error message is displayed on the console, and RESTART- the warm start routine is executed.

OPENR, OPENW

CP/M has one open 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.

Getting Started

SL5 is distributed as a set of CP/M text and COM files on a single diskette. Its 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 SYSCENed 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 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 cntl-C or reset the system to bring up CP/M
2. Enter SL5<cr>

The prompt > should appear on your screen. If it doesn't try again, then give us a call.



Object Modules ■



Object Modules

1. Introduction

The SL5 system disk has 2 routines for creating complete execution files or object modules, the COMMOD routine 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. 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 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 FLND 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 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 DOS take up several thousand bytes of memory. In many cases the host DOS 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 ( 8251 character output word - device addr ED control EC data )
    OED ZIN 1 &
END OEC ZOUT ;

: CIN BEGIN ( serial input - device addr ED control EC data )
    OED ZIN 2 &
END OEC ZIN 7F & DUP COUT ;

: CIS ;

```

(CTYPE, SC")

```

: CTYPE DUP IF
    0 DO
        DUP B@ COUT 1+
    LOOP DROP
    ELSE 2DROP
ENDIF ;

: SC" R> DUP B@ SWAP 1+ SWAP 2DUP + >R CTYPE ; ( type string to console )

```

CH COUT ;

```

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

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

: UGCH GCH ;

```

```

: OUTINIT ;
: INIT 0 TBUF ! 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 if the symbol table isn't 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 kernel is compiled with ?INTERP, ?COMPILE, ?FILESYS, and ?CRT on (set to 1). ?OWNIO is set to 0. Compiler words can be deleted without effecting 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

(See example 1 below)

1. Write the application program on a file.
2. Debug the program using DEV.COM and FLOAD.
3. Compile a RAM system with SYSMAKE.COM and test.
4. Write user I/O words and debug with DEV.COM
5. Add OWNIO section to KERNEL.SL5
Set ?COMPILE, ?INTERP, ?FILESYS to 0
Set ?OWNIO to 1 (edit KERNEL.SL5 source to change these constants)
6. Compile a ROM system with SYSMAKE.COM
Set the variable GOQLAD to the address of the driver word.
: DRIVER (your driver word) ;
'B DRIVER GOQLAD 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 DOS 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

```
.. SYSMAKE          ( load compiler 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
```

```
>l 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.s15 (compile kernel.s15 )
```

```
      :
```

```
kernel.s15 is compiled onto driver.com
```

```
      :
```

```
More input (Y/N) ? y
```

```
Enter source file name >driver.s15 ( compile driver.s15 )
```

```
      :
```

```
driver.s15 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.S15 and test.
4. Set conditional compile switches to delete unneeded 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 : 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

>l infof ! (turn off compiler source listing)
>ramgen

Enter object file name >myprog.com
Enter Kernel source file name >kernel.s15 (compile kernel.s15)

kernel.s15 is compiled onto myprog.com

More input (Y/N) ? y
Enter source file name >myprog.s15 (compile myprog.s15)

myprog.s15 is compiled onto myprog.com

more input (Y/N) ? n

do you want a symbol table (Y/N) ? y

(do not generate symbol table)

Successful compilation

Program size = 1F4B / 8011

(hex / decimal notation)

Dictionary size = AF3 / 2803

Total memory used = 2A3E / 10814

>

(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 nnnn 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 it's 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 underlined 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 400 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 COM modules 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, the symbol table is discarded,  
          otherwise it is preserved. Default = 1  
  
USERPAD - This can be set to point to an initialization  
          routine. On default it points to a dummy  
          routine.
```


Example 3 : Create a disk com file with COMMOD.SL5

```
A>sl5 ( load the kernel object file from CP/M )
>l 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 isn't 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: Create a disk COM file with a user initialization routine

```
sl5 ( load the kernel object file from CP/M )
>fload TEST.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 isn't 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
```

Example 5: Using COMMOD to create a disk COM file without a symbol table.

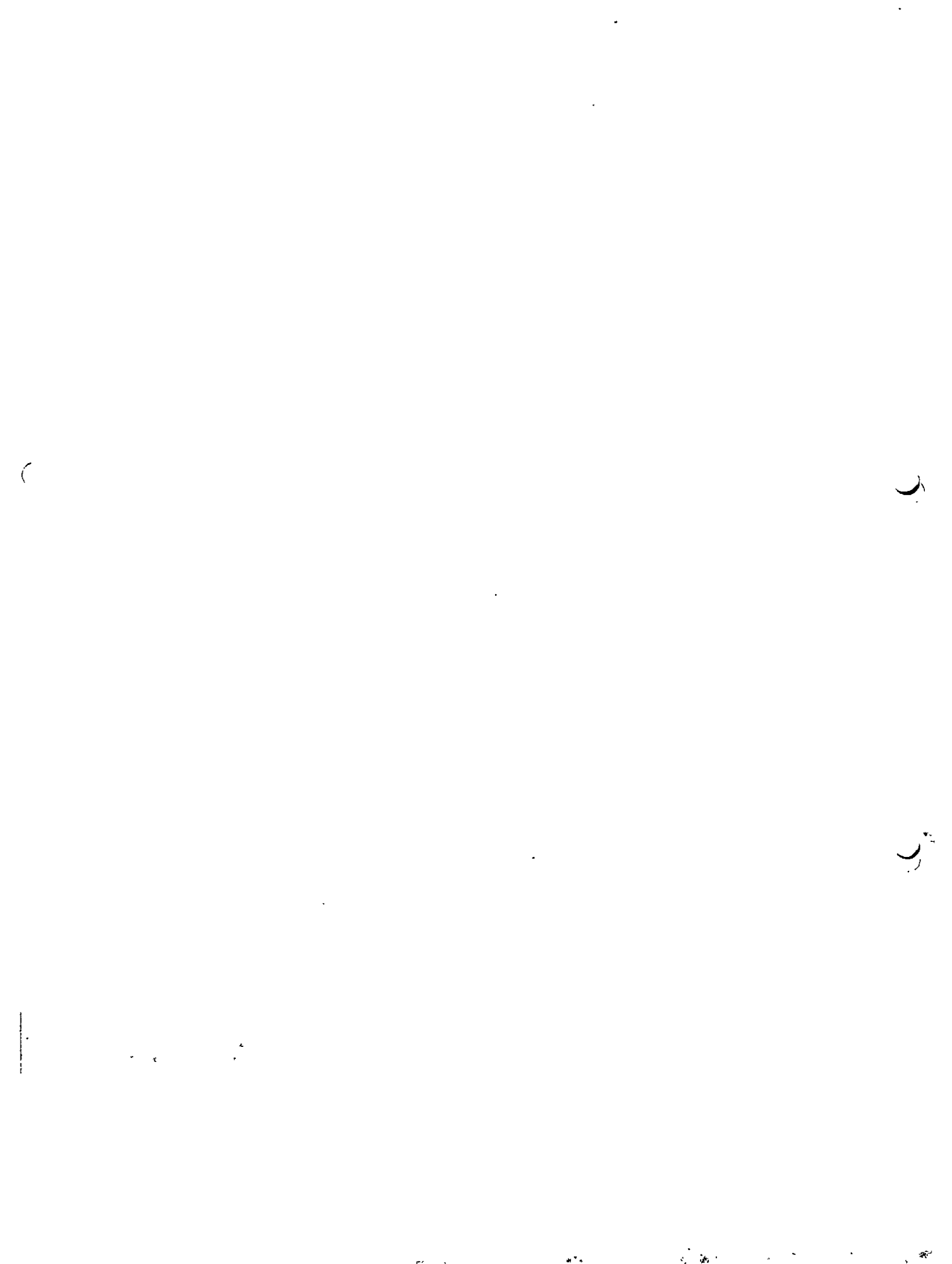
```
A>sl5          ( load the kernel object file from CP/M )
>fload TEST3.SL5      ( load user program on top of kernel )
: DRIVER BEGIN ( your main loop ) 0 END ;
>fload COMMOD.SL5     ( load the commod program )

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

>0 ?syntab !        ( symbol table will be discarded )
>`b DRIVER userpad ! ( set initialization routine 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 ■



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

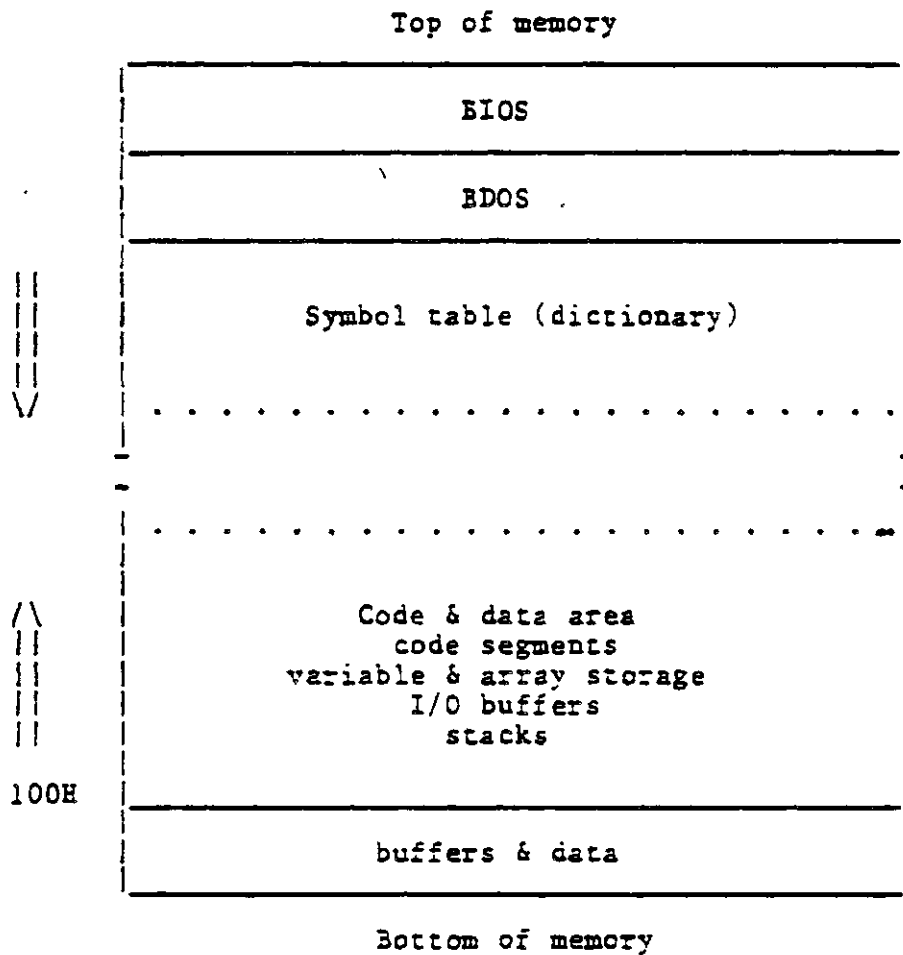


Figure 2-1. Typical memory layout for RAM-based systems.

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

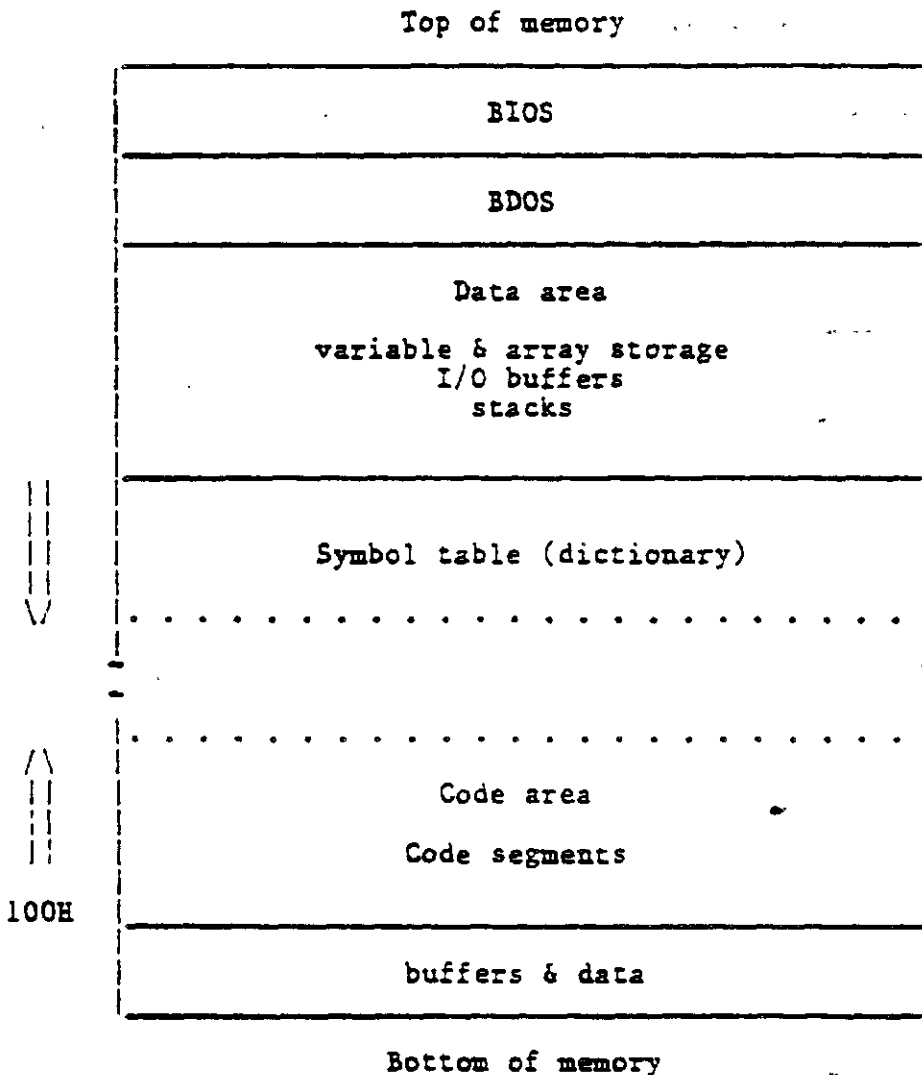


Figure 2-2. Memory layout for ROM-based systems.

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 isn't 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

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).

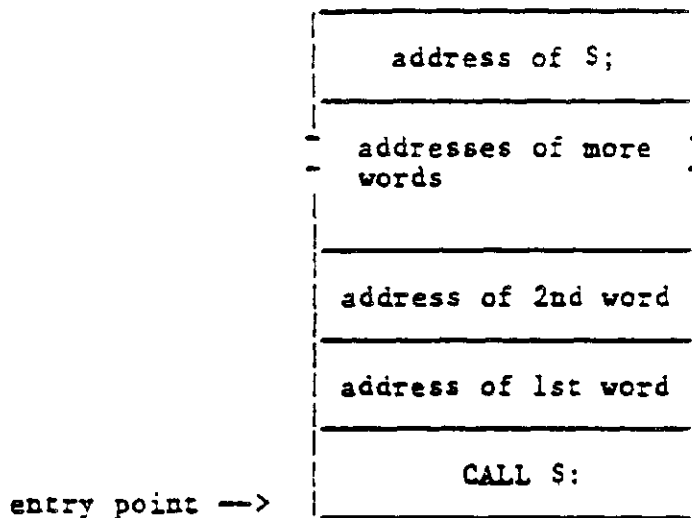


Figure 3-1. Memory format of colon definitions.

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 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 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.

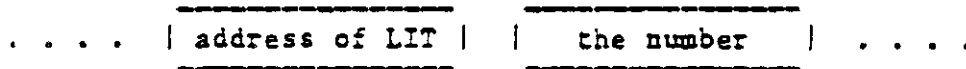


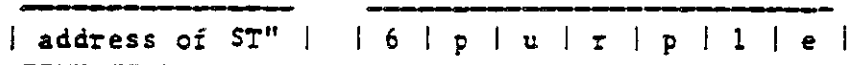
Figure 3-2. Compiled literals.

Upon execution of LIT, the literal is placed on the stack and the interpreter pointer is incremented to the memory location past the number so that the inner interpreter won't 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:



When executed \$T" will print

purple

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

3.1.3. Branching Within Colon Definitions

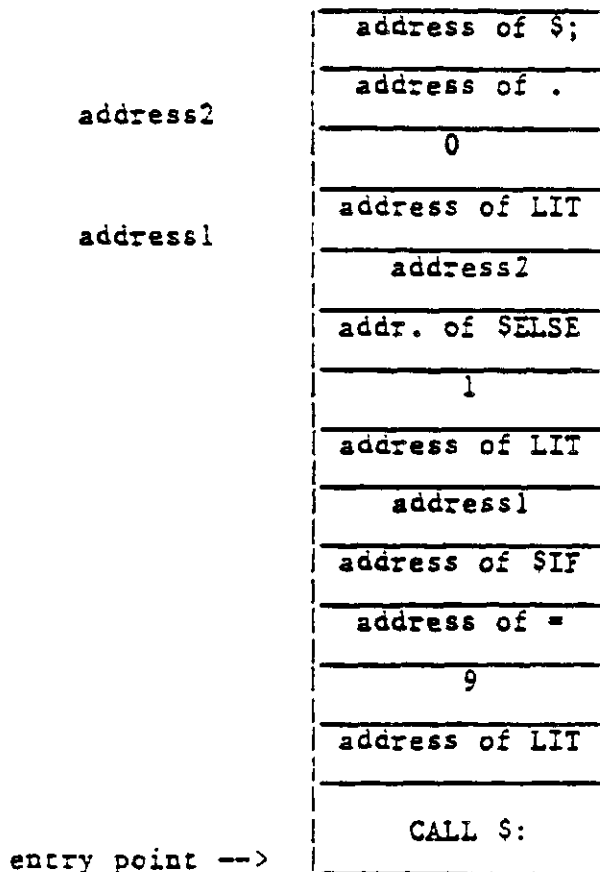
In this section, the internal structure of some of the branching constructions is described. The constructions discussed here should give you some insight into the structures which are not described in this section.

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 1 if it is a 9 or a 0 if the top of stack is anything else. The definition body would then resemble:



When NINE= is executed and the top of stack (TOS) is 9, \$IF will increment the interpreter pointer to skip over address1, LIT will push the 1 on the stack, \$ELSE will then set the interpreter pointer 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 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 print 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

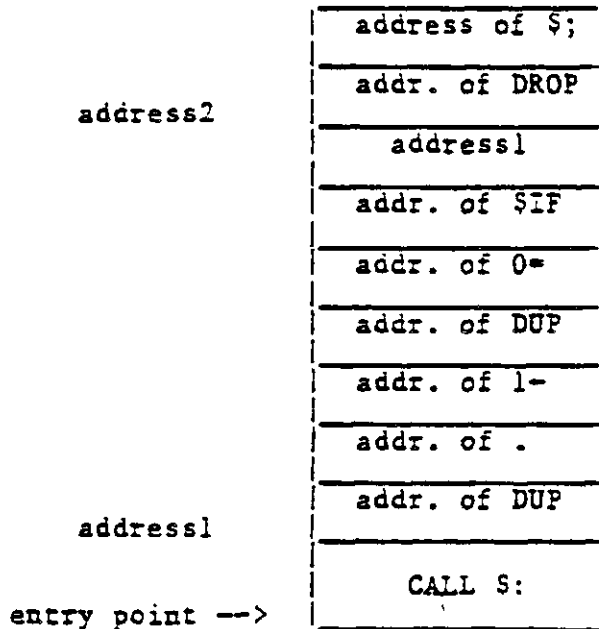


Figure 3-3. Memory diagram of the word N-1.

Figure 3-3 shows the memory image of the N-1. The word END puts the address of SIF followed by the loop address in the code segment. The word SIF will set the IP to address2 when the TOS 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 8 6 5 4 3 2 1

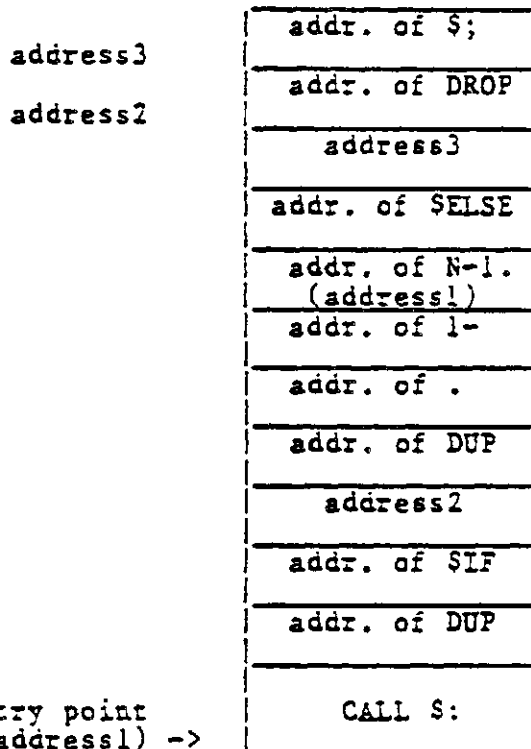


Figure 3-4. Illustration of N-1. using RECURSE.

N-1. cannot be used instead of RECURSE because the CONTEXT dictionary (see section 4.1) 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 3-5 shows the memory image of these two words.

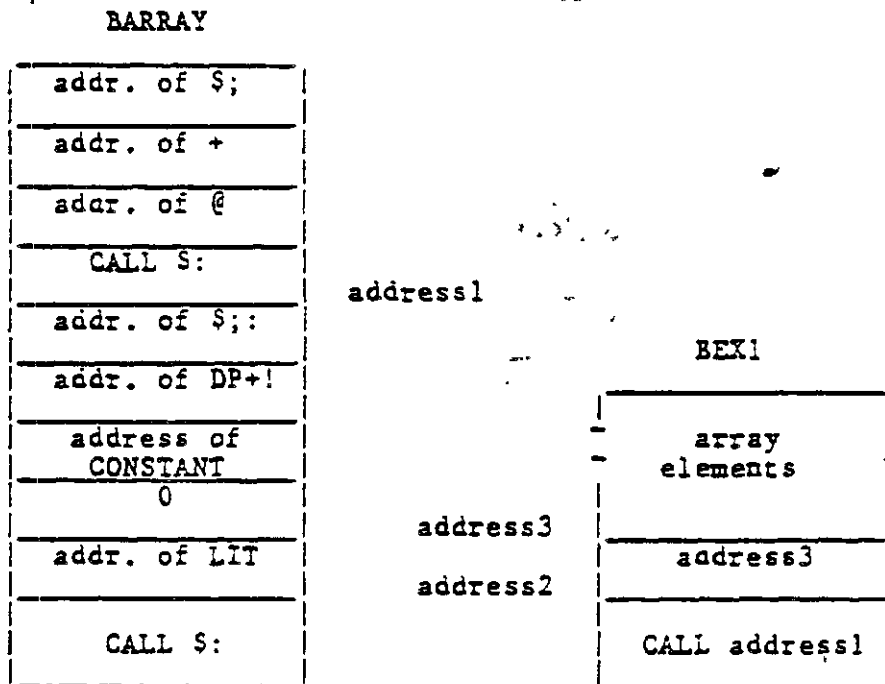


Figure 3-5. Illustration of BARRAY & BEX1.

When S: 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 S: at address1 will in turn set the interpreter pointer 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 nnnn
```

produces the memory images shown in the figure below.

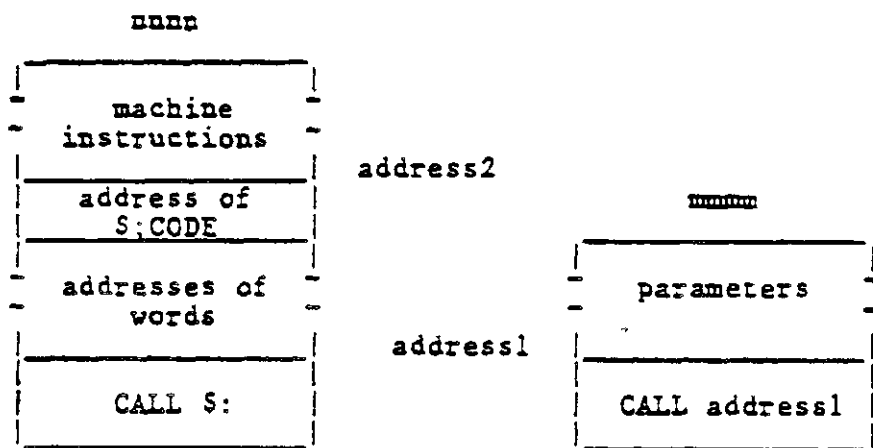


Figure 3-6. Memory format of ;CODE constructions.

3.2. CODE Definitions

When you define a code word, SL5 remains in the execution mode (STATE=0) like colon definitions 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.

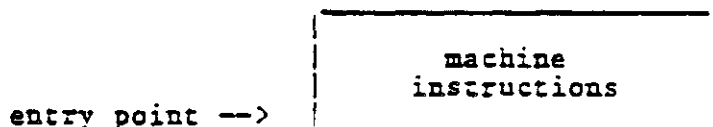


Figure 3-7. Memory format for CODE definitions.

3.3. CONSTANT

A constant pushes its value onto the stack. When you define a CONSTANT, the memory image resembles that of figure 3-8. When the constant is executed, the word \$CONSTANT picks up "convalue" and pushes it on the stack.

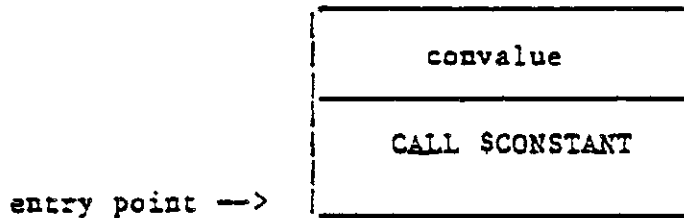


Figure 3-8. Memory format for constants.

3.4. VARIABLE

When a new variable is defined, a code segment of the form shown in figure 3-9 is created. Subsequent execution of the variable results in address1 being placed on the parameter stack to be used by words such as "!", "@", "l+!" ...etc.

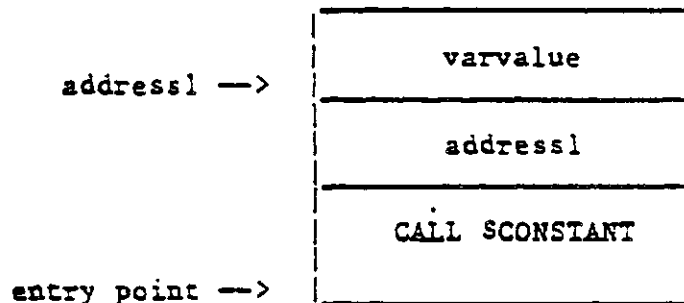


Figure 3-9. Format of variables.

3.5. Arrays

In this section, the term "arrays" refers 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 3-10 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.

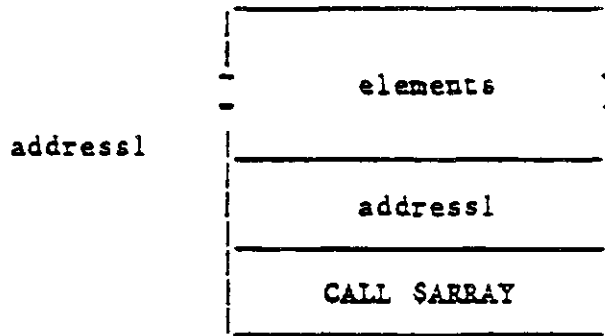


Figure 3-10. Memory image of arrays.

4. The Dictionary

4.1. The CURRENT & 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 (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 variable 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.

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.

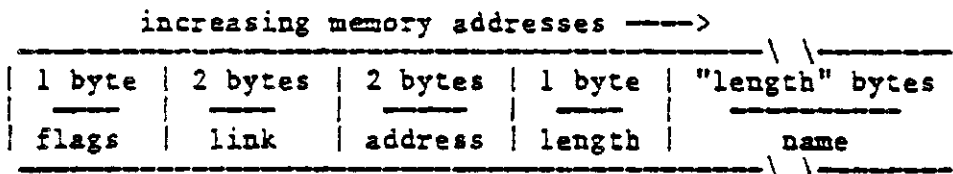


Figure 4-1. A symbol entry.

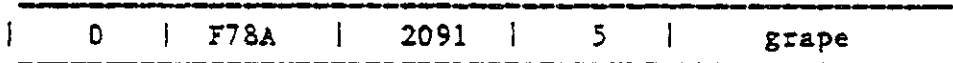
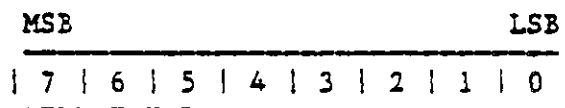


Figure 4-2. Example of a symbol entry for the word "grape".

Every symbol entry has a flag byte (figure 4-3) 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.



bit	usage
0-3	unused.
4	unused but reserved.
5	unused but reserved.
6	unused but reserved.
7	precedence bit. If set (1), the word will be executed regardless of the value of STATE.

NOTE: All bits default to the value 0.

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 4-4. The FORTH vocabulary is not chained to any vocabulary, so its "vlink" field is set to 0.

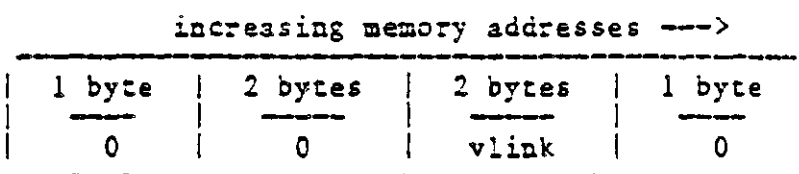


Figure 4-4. Vocabulary base format.

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 4-4) 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 4-5 with the ASSEMBLER and the USERV vocabularies chained to the FORTH vocabulary.

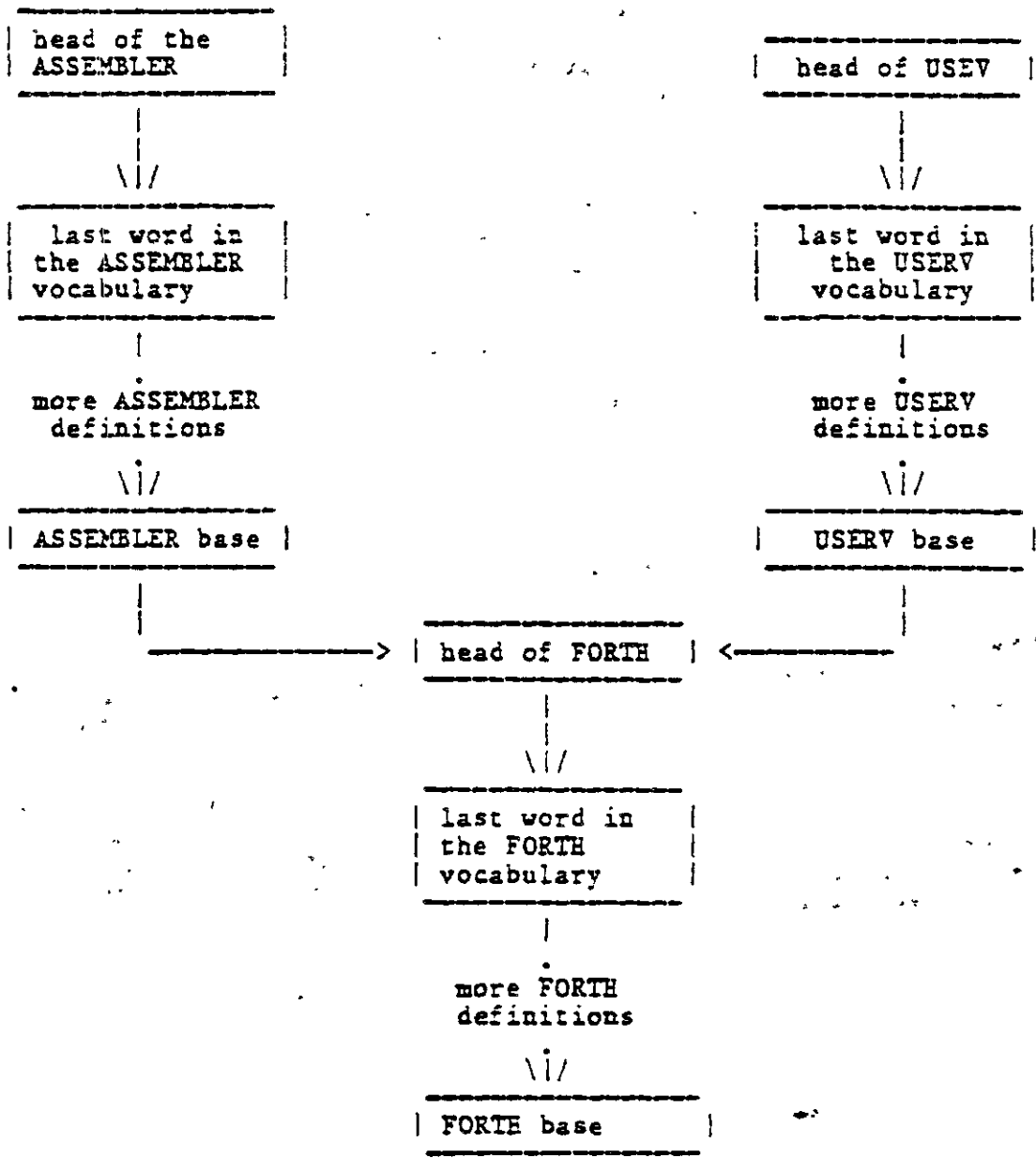


Figure 4-5. Diagram of chained vocabularies.

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>`

In the above example all words defined since and including the word 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.

ز

ز



Glossary

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

nnnn parms nnnn parms ---> values (chars)
description

where: nnnn is the name of the word being described.
parms 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 values 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, TRUE = 1.
e g m n a 16 bit integer.
p q r
: u v
- x y z
nnnn a name of a word which consists of 1-255 non-blank characters delimited on the right by a blank or a carriage-return.

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.
- K The word is a constant.

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

^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 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. Remainder has sign of dividend m. $p=m/n$

$m \ n \ | \ \longrightarrow \ p$
 16 bit Logical inclusive OR of m and n.

$m \ n \ \& \ \longrightarrow \ p$
 16 bit logical AND of m and n.

$m \ n \ \rightarrow L \ \longrightarrow \ p$
 Logical right shift of m by n bits.

$m \ n \ \leftarrow L \ \longrightarrow \ p$
 Logical left shift of m by n bits.

$m \ 1+ \ \longrightarrow \ n$
 Add one to the value on top of stack.

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

$m \ 1- \ \longrightarrow \ n$
 Subtract 1 from m.

$p \ 1-!$
 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.

$n \ \text{-ROLL}$
 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 interpret 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 if m is less than or equal to n.

= $m \ n \ = \ \longrightarrow \ f$
 True if m is equal to n .

<> $m \ n \ <> \ \longrightarrow \ f$
 f is true if m is not equal to n .

> $m \ n \ > \ \longrightarrow \ f$
 True if m is greater than n .

>= $m \ n \ >= \ \longrightarrow \ f$
 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 $m \ ABS \ \longrightarrow \ n$
 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@ $p \ B@ \ \longrightarrow \ m$
 Fetch the contents 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 DP. Increment the dictionary pointer 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 stack, and branches back to **begin** if the **TOS** is false (0), or to the word following **END** if true. 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** \longrightarrow **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 **TOS** does not match any of the values in the case body.

CASEND **CASEND** (CP)
Terminates a **CASE** statement sequence. See the word "**CASE**".

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

CIN **CIN** \longrightarrow **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 realtime 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.

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 (truth) 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 l / l
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 on TOS. Otherwise return a 0 to flag that the symbol isn't 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 definition which returns to the inner interpreter, GO functions more as a "call" statement.

GOQLAD GOQLAD (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 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 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-intrepreter 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 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 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 \longrightarrow q
 Leaves the greater of the two signed integers m and n on the stack.

MIN m n MIN \longrightarrow q
 Leaves the lesser of the two signed integers m and n on the stack.

MINUS m MINUS \longrightarrow n
 Leave the two's complement of m on the stack.

MOD m n MOD \longrightarrow 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 \longrightarrow f
 Equivalent to 0=.

NUMBER NUMBER \longrightarrow 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 on TOS if the conversion succeeds. A 0 is left on the stack if the number cannot be converted with the current radix.

OCTAL OCTAL
 Set the I/O number radix to 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 TOS.

PICK m PICK ---> n
 Copy the mth stack value onto the TOS (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 TOS 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 TOS.
 (3 ROLL = ROT)

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.

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

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

UPPER UPPER (V1)
When UPPER is set to 1, the routine "WORD" converts characters to upper case. UPPER defaults to 1.

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 TOS (<>0) causes the words between WHILE and REPEAT to be executed. REPEAT generates an unconditional jump back to BEGIN. A false TOS (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 blank (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.

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

ZIN

p ZIN \longrightarrow n
Read value n from port p (Z80 version only).

ZOUT

n p ZOUT
Output n to port p (Z80 version only).

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.
END,	cc END, Mark the end of a conditional loop. If cc is "true" 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, otherwise a branch to the code following "ENDIF," or "ELSE," is made.
REPEAT,	REPEAT, Mark the end of an unconditional loop.

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

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

BUFLEN fib^ BUFLLEN ---> 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 EOF hasn't been processed yet, the false value (0) is returned.

EOFCHR EOFCHR ---> n (K)
 A constant which determines the end-of-file character which RCH looks 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 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:fffffff.eee where "d" is the drive (A-Z) followed by a colon, "fffffff" is the file name, and "eee" is the extension name. The "d" and the "eee" fields are not mandatory. Serial devices are specified when "fffffff" 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 EOF 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.

DUMP s DUMP
: Dump memory from s to e on the terminal (or which device is
: selected via 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.



Review by Arne Henden

Stackwork's FORTH (SL5) is one of the more unusual and interesting implementations of FORTH that I've seen. It matches FORTH-77 (with some minor differences) instead of FORTH-79. While double precision integer arithmetic operations are not included, many normally optional features such as arrays and case statements are available. The SL5 copy that I have is over a year old so this review does not include any recent changes.

File Access

As opposed to Z80 FORTH, which is tied to CP/M but treats the disk just like traditional FORTH (as 250 1024-byte blocks, randomly accessed); SL5 uses sequential access, record oriented files exclusively and has no block structure.

This means that SL5 does not reserve 1024-byte memory blocks to buffer data from the disk, leaving more memory space for application software. One input and one output file can be open simultaneously with the basic system, and more files can easily be added if necessary. SL5 can perform character or buffered read/write operations. Rather than LOAD one application screen at a time, it loads the entire application file.

FORTHwords continued

The next couple of columns will discuss benchmark results for the five FORTHS mentioned in this column, a very accurate hardware timer using CTC#0, and discuss several screens of utility software. Also, there'll be a review of HART-FORTH.

Meanwhile, please send me your questions and comments about FORTH. I would very much like to help you use this unique programmer's language to its fullest advantage!



What you get

The SL5 disk comes with one of the best manuals I've seen. Its major entries include a tutorial (6 pages), reference (22 pages), the Z80 assembler (12 pages) and a glossary (20 pages).

The disk includes the kernel (both .ASM and .COM files), the debug package (DEBUG.SL5), and the assembler (ASSEM.SL5). SL5 itself required no modifications and worked the first time. One of the major advantages of SL5 is that all of the source code is included, making system modification simple.

Disk I/O

SL5's non-FORTHian file structure makes SL5 appear quite different from other versions of FORTH. No editor is included because all files can be entered with any standard editor such as ED. This means that debugging changes are not interactive.

Because you cannot examine source files easily while in SL5 (no LIST commands), errors during loading are hard to trace down. You cannot incrementally load your program. More disk activity is necessary since source code is brought in one sector at a time and no extensive buffering is included. At the same time, the ability to manipulate source files with standard editors and other language compilers is an advantage. Leaving out the screen buffers provides more application programming space.

The compiler and assembler

Another advantage of SL5 is that a cross-compiler is built-in. You can easily produce ROMable code for dedicated applications. SL5 does this by separating code and data areas from the symbol table. You can create headerless code by simply removing the symbol table while cross-compiling. Variables and arrays can be stored separately from code segments, easing the problem of creating ROMable FORTH.

New system configurations can be of two types: user applications loaded on top of SL5, creating a new

SL5.COM file; and creating entirely new systems using the cross compiler, perhaps modifying the kernel or burning a small application into ROM.

The assembler is excellent. Not only are Zilog mnemonics supported, but high level structures such as IF-ELSE-ENDIF and BEGIN-END are available. Stackworks is careful to point out system register usage to prevent loss of pointers by assembly code produced by the user.

Conclusion

My major complaint with SL5 is its file structure. When you lose the FORTH block/screen, you lose many of the advantages of FORTH. Interactive debugging is hampered, more disk I/O occurs, and less storage space is available on any given disk. SL5 is not easily upgradable to FORTH-79 because of the file structure.

Also, no double precision integer words are included, and floating point is not available as an option.

In all, I think SL5 is a good language that is well documented and comes with a lot of usually optional features. Its unique implementation of FORTH is interesting, but at the same time prevents me from recommending it to a user who wants to learn standard FORTH.



Name:	SL5
Authors:	The Stackworks Mike Brothers Larry Mongin Dave DeLauter
Type:	FORTH for the Z80
Distributor:	Supersoft Associates P. O. Box 1628 Champaign, IL 61820
Price:	\$175
Requires:	CP/M operating system Z80 processor 24K bytes RAM
Manual:	110 page 8 1/2 x 11" spiral bound