

# TI FORTH INSTRUCTION MANUAL

This document originally prepared in 1983 by  
Leslie O'Hagan  
Leon Tietz  
John T. Yantis

—Edited by Lee Stewart (2012)



# Dedication

This diskette-based Forth Language system for the Texas Instruments TI-99/4A Home Computer was adapted by Leon Tietz and Leslie O'Hagan of the TI Corporate Engineering Center from Ed Ferguson's TMS9900 implementation of the Forth Interest Group (FIG) standard kernel. This system was placed in the public domain "as is" by Texas Instruments on December 21, 1983, by sending one copy of this *TI Forth Instruction Manual* and the TI Forth System diskette to each of the TI-recognized TI-99/4A Home Computer User Groups as of that date. There were no more copies made, and none are available from Texas Instruments. TI Forth had not undergone the testing and evaluation normally given a product which is intended for distribution at the time TI withdrew from the Home Computer market. Although both the diskette and this manual may contain errors and omissions, TI Forth for the TI-99/4A Home Computer ***will not be supported by*** TI in any way, shape, form or fashion. What is contained in this manual and on the accompanying TI Forth System diskette is all that exists of this system, and is its sole reference.

Texas Instruments Incorporated (hereinafter "TI") hereby relinquishes any and all proprietary claims to the software language known as "TI Forth" to the public for free use thereof, without reservations on the part of TI. It should be understood that the TI Forth software language is not subject to any warranties of fitness, either express or implied, by TI, and TI makes no representations as to the fitness of the TI Forth software language for any intended application by the user. Any use of the TI Forth software language is specifically at the discretion of the user who assumes the entire responsibility for such use.

# Table of Contents

Dedication.....	iii
1 Introduction.....	1
1.1 Editor's Note.....	2
1.2 Starting Forth.....	3
2 Getting Started.....	5
2.1 Stack Manipulation.....	6
2.2 Arithmetic and Logical Operations .....	6
2.3 Comparison Operations .....	7
2.4 Memory Access Operations .....	7
2.5 Control Structures .....	8
2.6 Input and Output to/from the Terminal .....	9
2.7 Numeric Formatting .....	10
2.8 Disk-Related Words.....	10
2.9 Defining Words.....	11
2.10 Miscellaneous Words.....	11
3 How to Use the Forth Editor.....	13
3.1 Forth Screen Layout Caveat.....	14
3.2 The Two TI Forth Editors.....	14
3.3 Editing Instructions.....	14
4 Memory Maps.....	17
4.1 VDP Memory Map.....	17
4.2 CPU Memory.....	18
4.3 CPU RAM Pad.....	19
4.4 Low Memory Expansion.....	20
4.5 High Memory Expansion.....	20
5 System Synonyms and Miscellaneous Utilities.....	21
5.1 System Synonyms.....	21
5.1.1 VDP RAM Read/Write.....	22
5.1.2 Extended Utilities: GPLLNK, XMLLNK AND DSRLNK.....	23
5.1.3 VDP Write-Only Registers.....	24
5.1.4 VDP RAM Single-Byte Logical Operations.....	24
5.2 Disk Utilities.....	24
5.2.1 Disk Formatting Utility.....	25
5.2.2 Disk and Screen Copying Utilities.....	25
5.3 Listing Utilities.....	27
5.4 Debugging.....	27
5.4.1 Dump Information to Terminal.....	27
5.4.2 Tracing Word Execution.....	28
5.4.3 Recursion.....	29
5.5 Random Numbers.....	29
5.6 Miscellaneous Instructions.....	30

6	An Introduction to Graphics.....	31
6.1	Graphics Modes.....	31
6.2	Forth Graphics Words.....	32
6.3	Color Changes.....	32
6.4	Placing Characters on the Screen.....	33
6.5	Defining New Characters.....	34
6.6	Sprites.....	35
6.6.1	Magnification.....	35
6.6.2	Sprite Initialization.....	36
6.6.3	Using Sprites in Bit-Map Mode.....	37
6.6.4	Creating Sprites.....	37
6.6.5	Sprite Automotion.....	39
6.6.6	Distance and Coincidences between Sprites.....	40
6.6.7	Deleting Sprites.....	41
6.7	Multicolor Graphics.....	42
6.8	Using Joysticks.....	42
6.9	Dot Graphics.....	44
6.10	Special Sounds.....	45
6.11	Constants and Variables Used in Graphics Programming.....	46
7	The Floating Point Support Package.....	47
7.1	Floating Point Stack Manipulation.....	47
7.2	Floating Point Fetch and Store.....	48
7.3	Floating Point Conversion Words.....	48
7.4	Floating Point Number Entry.....	48
7.5	Floating Point Arithmetic.....	48
7.6	Floating Point Comparison Words.....	49
7.7	Formatting and Printing Floating Point Numbers.....	49
7.8	Transcendental Functions.....	50
7.9	Interface to the Floating Point Routines.....	50
8	Access to File I/O Using TI-99/4A Device Service Routines.....	52
8.1	The Peripheral Access Block (PAB).....	52
8.2	File Setup and I/O Variables.....	53
8.3	File Attribute Words.....	54
8.4	Words that Perform File I/O.....	55
8.5	Alternate Input and Output.....	59
8.6	File I/O Example 1: Relative Disk File.....	59
8.7	File I/O Example 2: Sequential RS232 File.....	61
9	The TI Forth 9900 Assembler.....	62
9.1	TMS9900 Assembly Mnemonics.....	62
9.2	Forth's Workspace Registers.....	63
9.3	Workspace Register Addressing.....	64
9.4	Symbolic Memory Addressing.....	64
9.5	Workspace Register Indirect Addressing.....	65
9.6	Workspace Register Indirect Auto-increment Addressing.....	65

9.7 Indexed Memory Addressing.....	65
9.8 Addressing Mode Words for Special Registers.....	65
9.9 Handling the Forth Stacks.....	66
9.10 Structured Assembler Constructs.....	66
9.11 Assembler Jump Tokens.....	67
9.12 Assembly Example for Structured Constructs.....	67
10 Interrupt Service Routines (ISRs).....	69
10.1 Installing a Forth Language Interrupt Service Routine.....	69
10.2 An Example of an Interrupt Service Routine.....	70
10.3 Installing the ISR.....	70
10.4 Some Additional Thoughts Concerning the Use of ISRs.....	71
11 Potpourri.....	72
11.1 BSAVE and BLOAD.....	72
11.1.1 Customizing How TI Forth Boots Up.....	73
11.1.2 An Overlay System with BSAVE/BLOAD.....	74
11.1.3 An Easier Overlay System in Source Code.....	75
11.2 Conditional Loads.....	76
11.3 Memory Resident Messages.....	77
11.4 CRU Words.....	77
12 TI Forth Dictionary Entry Structure.....	78
12.1 Link Field.....	78
12.2 Name Field.....	78
12.3 Code Field.....	79
12.4 Parameter Field.....	80
Appendix A ASCII Keycodes (Sequential Order).....	81
Appendix B ASCII Keycodes (Keyboard Order).....	83
Appendix C Differences between Starting FORTH and TI Forth.....	85
Appendix D The TI Forth Glossary.....	92
D.1 Explanation of Some Terms and Abbreviations.....	92
D.2 TI Forth Word Descriptions.....	93
Appendix E User Variables in TI Forth.....	156
E.1 TI Forth User Variables (Address Offset Order).....	156
E.2 TI Forth User Variables (Variable Name Order).....	158
Appendix F TI Forth Load Option Directory.....	160
F.1 Option: -SYNONYMS.....	160
F.2 Option: -EDITOR (40-Column Editor).....	160
F.3 Option: -COPY.....	160
F.4 Option: -DUMP.....	161
F.5 Option: -TRACE.....	161
F.6 Option: -FLOAT.....	161
F.7 Option: -TEXT.....	162
F.8 Option: -GRAPH1.....	162
F.9 Option: -MULTI.....	162
F.10 Option: -GRAPH2.....	162

F.11 Option: -SPLIT.....	163
F.12 Option: -VDPMODES.....	163
F.13 Option: -GRAPH.....	163
F.14 Option: -FILE.....	164
F.15 Option: -PRINT.....	164
F.16 Option: -CODE.....	164
F.17 Option: -ASSEMBLER.....	165
F.18 Option: -64SUPPORT ( 64-Column Editor ).....	165
F.19 Option: -BSAVE.....	165
F.20 Option: -CRU.....	165
Appendix G Assembly Source for CODEd Words.....	166
Appendix H Error Messages.....	171
Appendix I Contents of the TI Forth Diskette.....	173
Appendix J TI Forth Bugs.....	198
Appendix K Diskette Format Details.....	199
K.1 Volume Information Block (VIB).....	199
K.2 File Descriptor Index Record (FDIR).....	200
K.3 File Descriptor Record (FDR).....	200
K.4 Comparison of TI Forth and TI File System Layouts on the Same Disk.....	201
K.4.1 TI Forth System Disk.....	202
K.4.2 TI Forth Work Disk.....	204
Appendix L TI Forth System for Larger Disks.....	205
L.1 Larger System Disk.....	205
L.2 Larger Work Disk.....	206
L.3 Updating Disk Utilities for Larger Disks.....	206





# 1 Introduction

The Forth language was invented in 1969 by Charles Moore and has continually gained acceptance. The last several years have shown a dramatic increase in this language's following due to the excellent compatibility between Forth and mini- and microcomputers. Forth is a threaded interpretive language that occupies little memory, yet, maintains an execution speed within a factor of two of assembly language for most applications. It has been used for such diverse applications as radio telescope control to the creation of word processing systems. The Forth Interest Group (FIG) is dedicated to the standardization and proliferation of the Forth language. TI Forth is an extension of the fig-Forth dialect of the language. The fig-Forth language is in the public domain. Nearly every currently available mini- and microcomputer has a Forth system available on it, although some of these are not similar to the FIG version of the language.

The address for the Forth Interest Group is:

Forth Interest Group  
P. O. BOX 1105  
San Carlos, CA 94070

This document will cover some of the fundamentals of Forth and then show how the language has been extended to provide easy access to the diverse features of the TI-99/4A Computer. The novice Forth programmer is advised to seek additional information from such publications as:

*Starting FORTH*  
by Leo Brodie  
published by Prentice Hall

*Using FORTH*  
by Forth Inc.

*Invitation to FORTH*  
by Katzan  
published by Petrocelli Books

In order to utilize all the capabilities of the TI-99/4A, it is necessary to understand its architecture. It is recommended that the user who wants to use Forth for graphics, music, access to Disk Manager functions or files have a working knowledge of this architecture. This information is available in the *Editor/Assembler Manual* accompanying the Editor/Assembler Command Module. All the capabilities addressed in that document are possible in Forth and most have been provided by easy-to-use Forth words that are documented in this manual.

Forth is designed around a virtual machine with a stack architecture. There are two stacks: The first is referred to variously as the data stack, parameter stack or stack. The second is the return stack. The act of programming in Forth is the act of defining procedures called "words", which are defined in terms of other more basic words. The Forth programmer continues to do this until a single word becomes the application desired. Since a Forth word must exist before it can be referenced, a bottom up programming discipline is enforced. The language is structured and contains no GOTO statements. Successful Forth programming is best achieved by designing top down and programming bottom up.

Bottom-up programming is inconvenient in most languages due to the difficulty in generating drivers to adequately test each of the routines as they are created. This difficulty is so severe that bottom-up programming is usually abandoned. In Forth, however, each routine can be tested interactively from the console and it will execute identically to the environment of being called by another routine. Words take their parameters from the stack and place the results on the stack. To test a word, the programmer can type numbers at the console. These are put on the stack by the Forth system. Typing the word to be tested causes it to be executed and when complete, the stack contents can be examined. By writing only relatively small routines (words) all the boundary conditions of the routine can easily be tested. Once the word is tested (debugged) it can be used confidently in subsequent word definitions.

The Forth stack is 16 bits wide. [*Editor's Note:* In Forth, a 16-bit value is known as a *cell*; hence, the stack is one cell wide.] When multi-precision values are stored on the stack they are always stored with the most significant part most accessible. The width of the return stack is implementation dependent as it must contain addresses so that words can be nested to many levels. The return stack in TI Forth is 16 bits wide.

Disk drives in TI Forth are numbered starting with 0 and are abbreviated with “DR” preceding the drive number: DR0, DR1, etc. Other TI languages (TI BASIC, TI Extended BASIC, TI Assembler, etc.) and software refer to disk drives starting with 1 and the abbreviation “DSK” preceding the disk (drive) number: DSK1, DSK2, etc. From this you can see that DR0 and DSK1 refer to the same disk drive. When referring to the disk drives by device names, they will always be DSK1, ..., such as part of a complete file reference, *e.g.*, DSK1.MYFILE.

Keyboard key names in this document will be offset with “<” and set in the italicized font of the following examples: <ENTER>, <CTRL+V>, <FCTN+4>, <BREAK> and <CLEAR>. Incidentally, the last three key names listed refer to the same key.

## 1.1 Editor's Note

The source for this document was a series of sixteen files named A, B, C, ..., P in TI-Writer format, which I had purchased from the MANNERS (Mid-Atlantic Ninety-NinERS) TI Users Group shortly after TI put TI Forth into the public domain. I do not know who deserves the credit for originating these files; but, it was always my understanding they came from TI and that the printed document we all received with the TI Forth system was prepared in and printed from TI Writer. However, the A – P files have differences from the printed document. I have attempted to correct those differences; but, I have also taken the liberty of elaborating on the original in an effort to make it easier to understand and to correct known bugs. I have added a new chapter, “12 TI Forth Dictionary Entry Structure” and three new appendices, “J TI Forth Bugs”, “K Diskette Format Details” and “L TI Forth System for Larger Disks”.

Though I have been careful with my additional coding, as with anything else in this document, you assume responsibility for any use you make of it. Please, feel free to contact me with comments and corrections at *lee@stewkitt.com*.

—Lee Stewart  
February, 2012  
Silver Run, MD

## 1.2 Starting Forth

To operate the TI Forth System, you must have the following equipment:

- TI-99/4A Console
- Monitor
- Memory Expansion
- Disk Controller
- 1 (or more) Disk Drives
- Editor/Assembler Module
- RS232 Interface (optional)
- Printer (optional)

See the manuals accompanying each item for proper assembly of the TI-99/4A system.

To begin, power up the system. The TI Color-Bar screen should appear on your monitor. (If it does not, power down and recheck all connections.) Press any key to continue. A new screen will appear displaying a choice between TI BASIC and the Editor/Assembler. To use Forth, select the Editor/Assembler.

On the next screen choose the **LOAD AND RUN** option. The computer will ask for a **FILE NAME**. After placing your TI Forth System disk in the first drive, type "DSK1.FORTH" and press **<ENTER>**.

The TI Forth welcome screen will display a list of load options (or elective blocks). Each option loads all routines necessary to perform a particular group of tasks:

Load Option	Loads Forth Words Necessary to:	Chapter
<b>-SYNONYMS</b>	Perform VDP reads and writes. Random number generators and the disk formatting routine are also loaded.	5
<b>-EDITOR</b>	Run the regular, 40-column TI Forth editor.	3
<b>-COPY</b>	Copy Forth screens <sup>1</sup> and Forth disks. String store routines are also loaded.	5
<b>-DUMP</b>	Execute <b>DUMP</b> and <b>VLIST</b> .	5
<b>-TRACE</b>	Trace the execution of Forth words.	5
<b>-FLOAT</b>	Use floating-point arithmetic.	7
<b>-VDPMODES</b>	Change display screen to any of the 6 available VDP modes.	6
<b>-TEXT</b>	Change display screen to Text mode.	6
<b>-GRAPH1</b>	Change display screen to Graphics mode.	6

<sup>1</sup> A Forth screen is also called a block and consists of 16 lines of 64 characters for a total of 1024 characters. When a Forth screen is loaded from disk, 1024 characters are copied from the disk into a VDP RAM buffer. This is explained in more detail later in this document.

Load Option	Loads Forth Words Necessary to:	Chapter
<b>-MULTI</b>	Change display screen to Multicolor mode.	6
<b>-GRAPH2</b>	Change display screen to Graphics2 (bit-map) mode.	6
<b>-SPLIT</b>	Change display screen to either of the two Split modes.	6
<b>-FILE</b>	Utilize the file I/O capabilities of the TI-99/4A.	8
<b>-PRINT</b>	Send output to an RS232 device.	8
<b>-64SUPPORT</b>	Run the 64-column TI Forth editor.	3
<b>-CODE</b>	Write assembly code in hexadecimal.	9
<b>-ASSEMBLER</b>	Write routines in TI Forth Assembler.	9
<b>-GRAPH</b>	Utilize the graphics capabilities of the TI-99/4A.	6
<b>-BSAVE</b>	Save dictionary overlays to diskette.	11
<b>-CRU</b>	Access the Forth equivalents of TI-Assembler mnemonics: LDCR, STCR, SBO, SBZ and TB.	11

To load a particular package, simply type its name exactly as it appears in the list. For example, to load the graphics package, type **-GRAPH** and press **<ENTER>**. You may load more than one package at a time.

The list of load options may be displayed at any time by typing the word **MENU** and pressing **<ENTER>**. See Appendix F for a detailed list of what each option loads.

## 2 Getting Started

This chapter will familiarize you with the most common words (instructions) in the Forth Interest Group version of Forth (fig-Forth). The purpose is to permit those users that have at least an elementary knowledge of some Forth dialect to easily begin to use TI Forth. Those with no Forth experience should begin by reading a book such as *Starting FORTH* by Leo Brodie. Appendix C is designed to be used with this particular text and lists differences between the Forth language described in the book (poly-Forth) and TI Forth.

A word in Forth is any sequence of characters delimited by blanks or a carriage return (**<ENTER>**). In this document, all Forth words will be set in a bold mono-spaced font that distinguishes the digit ‘0’ from the capital letter ‘O’ and will always be followed by a blank, even when punctuation such as a period or a comma follows. For example, **DUP** is such a Forth word and is shown also at the end of this sentence to demonstrate this practice: **DUP** . This obviously looks odd; but, this notation is necessary to avoid ambiguity when discussing Forth words because many of them either end in or, in fact, are such punctuation marks themselves. For example, the following, space-delimited character strings are all Forth words:

. : , ' ! ; C, C! ;CODE ? ." !"

The following convention will be used when referring to the stack in Forth:

(  $n_1$   $n_2$  ---  $n_3$  )

This diagram shows the stack contents before and after the execution of a word. In this case the stack contains two values,  $n_1$  and  $n_2$ , before execution of a word. The execution is denoted by “---” and the stack contents after execution is  $n_3$ . The most accessible stack element is always on the right. In this example,  $n_2$  is more accessible than  $n_1$ . There may be values on the stack that are less accessible than  $n_1$  but these are unaffected by the execution of the word in question.

The return stack may also be indicated beside the parameter stack (the stack) with a preceding “R:”, especially when both stacks are involved, as follows:

(  $n$  --- ) ( R: ---  $n$  )

In addition, the following symbols are used as operands for clarity:

### SOME SYMBOLS USED IN THIS DOCUMENT

$n, n_1, \dots$	16-bit signed numbers
$d, d_1, \dots$	32-bit signed double numbers
$u$	16-bit unsigned number
$ud$	32-bit unsigned double number
$addr, addr_1, \dots$	memory addresses
$b$	8-bit byte ( in right half of word)
$c$	7-bit character (in right end of word )
$flag$	Boolean flag ( 0 = false, non-0 = true )
	separates alternate results

## 2.1 Stack Manipulation

The following are the most common stack manipulation words:

<b>DUP</b>	( $n \text{ --- } n \ n$ )	Duplicate top of stack
<b>DROP</b>	( $n \text{ ---}$ )	Discard top of stack
<b>SWAP</b>	( $n_1 \ n_2 \text{ --- } n_2 \ n_1$ )	Exchange top two stack items
<b>OVER</b>	( $n_1 \ n_2 \text{ --- } n_1 \ n_2 \ n_1$ )	Make copy of second item on top
<b>ROT</b>	( $n_1 \ n_2 \ n_3 \text{ --- } n_2 \ n_3 \ n_1$ )	Rotate third item to top
<b>-DUP</b>	( $n \text{ --- } n \ n \mid n$ )	Duplicate only if non-zero
<b>&gt;R<sup>2</sup></b>	( $n \text{ ---}$ ) ( R: $\text{--- } n$ )	Move top item on stack to return stack
<b>R&gt;</b>	( $\text{--- } n$ ) ( R: $n \text{ ---}$ )	Move top item on return stack to stack
<b>R</b>	( $\text{--- } n$ ) ( R: $n \text{ --- } n$ )	Copy top item of return stack to stack

## 2.2 Arithmetic and Logical Operations

The following are the most common arithmetic and logical operations:

<b>+</b>	( $n_1 \ n_2 \text{ --- } n_3$ )	Add
<b>D+</b>	( $d_1 \ d_2 \text{ --- } d_3$ )	Add double precision numbers
<b>-</b>	( $n_1 \ n_2 \text{ --- } n_3$ )	Subtract ( $n_1 - n_2$ )
<b>1+</b>	( $n_1 \text{ --- } n_2$ )	Increment by 1
<b>2+</b>	( $n_1 \text{ --- } n_2$ )	Increment by 2
<b>1-</b>	( $n_1 \text{ --- } n_2$ )	Decrement by 1
<b>2-</b>	( $n_1 \text{ --- } n_2$ )	Decrement by 2
<b>*</b>	( $n_1 \ n_2 \text{ --- } n_3$ )	Multiply
<b>/</b>	( $n_1 \ n_2 \text{ --- } n_3$ )	Divide ( $n_1 / n_2$ )
<b>MOD</b>	( $n_1 \ n_2 \text{ --- } n_3$ )	Modulo ( remainder from $n_1 / n_2$ )
<b>/MOD</b>	( $n_1 \ n_2 \text{ --- } \text{rem quot}$ )	Divide giving remainder & quotient
<b>*/MOD</b>	( $n_1 \ n_2 \ n_3 \text{ --- } \text{rem quot}$ )	$n_1 * n_2 / n_3$ with 32 bit intermediate
<b>*/</b>	( $n_1 \ n_2 \ n_3 \text{ --- } n_4$ )	Like <b>*/MOD</b> but giving <i>quot</i> only

---

2 **>R** and **R>** must be used with caution as they may interfere with the normal address stacking mechanism of Forth. Make sure that each **>R** in your program has an **R>** to match it in the same word definition.

<b>U*</b>	$(u_1 u_2 \dots u_{d_1} \dots u_{d_2})$	Unsigned * with double product
<b>U/</b>	$(u_1 u_2 \dots u_{rem} u_{quot})$	Unsigned / with remainder
<b>MAX</b>	$(n_1 n_2 \dots n_1   n_2)$	Maximum
<b>MIN</b>	$(n_1 n_2 \dots n_1   n_2)$	Minimum
<b>ABS</b>	$(n \dots  n )$	Absolute value
<b>DABS</b>	$(d \dots  d )$	Absolute value of 32-bit number
<b>MINUS</b>	$(n_1 \dots n_2)$	Leave two's complement
<b>DMINUS</b>	$(d_1 \dots d_2)$	Leave two's complement of 32-bits
<b>AND</b>	$(n_1 n_2 \dots n_3)$	Bitwise logical AND $n_3$
<b>OR</b>	$(n_1 n_2 \dots n_3)$	Bitwise logical OR $n_3$
<b>XOR</b>	$(n_1 n_2 \dots n_3)$	Bitwise logical exclusive OR $n_3$
<b>SWPB</b>	$(n_1 \dots n_2)$	Swap the bytes of $n_1$ producing $n_2$
<b>SRC</b>	$(n_1 n_2 \dots n_3)$	Shift $n_1$ right circular $n_2$ bits giving $n_3$
<b>SRL</b>	$(n_1 n_2 \dots n_3)$	Shift $n_1$ right logical $n_2$ bits giving $n_3$
<b>SRA</b>	$(n_1 n_2 \dots n_3)$	Shift $n_1$ right arithmetic $n_2$ bits giving $n_3$
<b>SLA</b>	$(n_1 n_2 \dots n_3)$	Shift $n_1$ left arithmetic $n_2$ bits giving $n_3$

### 2.3 Comparison Operations

The following are the most common comparisons:

<b>&lt;</b>	$(n_1 n_2 \dots flag)$	True if $n_1$ less than $n_2$ (signed)
<b>=</b>	$(n_1 n_2 \dots flag)$	True if top two numbers are equal
<b>&gt;</b>	$(n_1 n_2 \dots flag)$	True if $n_1$ greater than $n_2$
<b>0&lt;</b>	$(n \dots flag)$	True if top number is negative
<b>0=</b>	$(n \dots flag)$	True if top number is 0 ( <i>i.e.</i> NOT)
<b>U&lt;</b>	$(u_1 u_2 \dots flag)$	Unsigned integer compare

### 2.4 Memory Access Operations

The following operations are used to inspect and modify memory locations anywhere in the computer:

<b>@</b>	$(addr \dots n)$	Replace word address by its contents
<b>!</b>	$(n addr \dots)$	Store $n$ at address (store a word)

<b>C@</b>	( <i>addr</i> --- <i>b</i> )	Fetch the byte at <i>addr</i>
<b>C!</b>	( <i>b</i> <i>addr</i> --- )	Store <i>b</i> at address (store a byte)
<b>?</b>	( <i>addr</i> --- )	Print the contents of address
<b>+!</b>	( <i>n</i> <i>addr</i> --- )	Add <i>n</i> to contents of address
<b>CMOVE</b>	( <i>from_addr</i> to_ <i>addr</i> <i>u</i> --- )	Block move <i>u</i> bytes.
<b>FILL</b>	( <i>addr</i> <i>u</i> <i>b</i> --- )	Fill <i>u</i> bytes with <i>b</i> beginning at <i>addr</i>
<b>ERASE</b>	( <i>addr</i> <i>u</i> --- )	Fill <i>u</i> bytes beginning at <i>addr</i> with 0s
<b>BLANKS</b>	( <i>addr</i> <i>u</i> --- )	Fill <i>u</i> bytes with blanks beginning at <i>addr</i>

## 2.5 Control Structures

The following sets of words are used to implement control structures in Forth. They are used to create all looping and conditional structures. These structures may be nested to any depth. If they are nested improperly an error message will be generated at compile time and the word definition will be aborted.

<b>DO ... LOOP</b>		<b>DO</b> sets up a loop with a loop counter. The stack contains the first and final values of the loop counter. The loop is executed at least once. <b>LOOP</b> causes a return to the word following <b>DO</b> unless termination is reached.
<b>DO</b>	( <i>end+1</i> <i>start</i> --- )	
<b>I</b>	( --- <i>n</i> )	Used between <b>DO</b> and <b>LOOP</b> . Places value of loop counter on stack.
<b>J</b>	( --- <i>n</i> )	Used when <b>DO LOOP</b> s are nested. Places value of next outer loop counter on the stack.
<b>LEAVE</b>	( --- )	Causes loop to terminate at next <b>LOOP</b> or <b>+LOOP</b> .
<b>DO ... +LOOP</b>		<b>DO</b> as above. <b>+LOOP</b> adds top stack value to loop counter (index)
<b>DO</b>	( <i>end+1</i> <i>start</i> --- )	
<b>+LOOP</b>	( <i>n</i> --- )	
<b>IF ... ENDIF</b>		<b>IF</b> tests the top of stack and if non-zero (true), the words between <b>IF</b> and <b>ENDIF</b> are executed. Otherwise, they are skipped and execution resumes after <b>ENDIF</b> .
<b>IF</b>	( <i>flag</i> --- )	
<b>IF ... ELSE ... ENDIF</b>		<b>IF</b> tests the top of stack and if non-zero (true), the words between <b>IF</b> and <b>ELSE</b> are executed. If the top of the stack is zero (false), the words between <b>ELSE</b> and <b>ENDIF</b> are executed. Execution then continues after <b>ENDIF</b> .
<b>IF</b>	( <i>flag</i> --- )	
<b>THEN</b>		May be used as a synonym for <b>ENDIF</b> .
<b>BEGIN ... UNTIL</b>		Loop which executes the words between <b>BEGIN</b> and



<b>UNTIL</b> ( <i>flag</i> ---)	<b>UNTIL</b> until the top of stack when tested by <b>UNTIL</b> is non-zero (true).
<b>END</b>	May be used as a synonym for <b>UNTIL</b> .
<b>BEGIN ... AGAIN</b>	Creates an infinite loop continually re-executing the words between <b>BEGIN</b> and <b>AGAIN</b> <sup>3</sup> .
<b>BEGIN ... WHILE ... REPEAT</b> <b>WHILE</b> ( <i>flag</i> ---)	Executes words between <b>BEGIN</b> and <b>WHILE</b> leaving <i>flag</i> which is tested by <b>WHILE</b> . If <i>flag</i> is non-zero (true), executes words between <b>WHILE</b> and <b>REPEAT</b> , then jumps back to <b>BEGIN</b> . If <i>flag</i> is zero (false), continues execution after the <b>REPEAT</b> .
<b>CASE</b> <i>n</i> <sub>1</sub> <b>OF ... ENDOF</b> <i>n</i> <sub>2</sub> <b>OF ... ENDOF</b> ... <i>n</i> <sub><i>m</i></sub> <b>OF ... ENDOF</b> ... <b>ENDCASE</b> <b>CASE</b> ( <i>n</i> ---)	Looks for a number ( <i>n</i> <sub>1</sub> , <i>n</i> <sub>2</sub> , ..., <i>n</i> <sub><i>m</i></sub> ) matching <i>n</i> . If there is a match, executes the code between the <b>OF ... ENDOF</b> set that immediately follows the matching number, proceeding then to the code following <b>ENDCASE</b> . If there is no match, the code after the last <b>ENDOF</b> is executed, with <b>ENDCASE</b> dropping <i>n</i> from the stack. Execution then continues after <b>ENDCASE</b> . Code after the last <b>ENDOF</b> may use <i>n</i> , which is still available; but, it must not consume <i>n</i> . Otherwise, <b>ENDCASE</b> will drop whatever was under <i>n</i> , adversely affecting program logic and possibly causing a stack underflow.

## 2.6 Input and Output to/from the Terminal

The most common type of terminal input is simply to enter a number at the terminal. This number will be placed on the stack. The number which is input will be converted according to the number base stored at **BASE** . **BASE** is also used during numeric output.

<b>DECIMAL</b>	( --- )	Sets the base to Decimal (Base 10)
<b>HEX</b>	( --- )	Sets the base to Hexadecimal (Base 16)
<b>BASE</b>	( --- <i>addr</i> )	System variable containing number base. To set some base ( <i>e.g.</i> , Octal) use the following sequence: <b>8 BASE !</b>
<b>.</b>	( <i>n</i> --- )	Print a signed number
<b>U.</b>	( <i>u</i> --- )	Print an unsigned number
<b>.R</b>	( <i>n</i> <sub>1</sub> <i>n</i> <sub>2</sub> --- )	Print <i>n</i> <sub>1</sub> right-justified in field of width <i>n</i> <sub>2</sub>
<b>D.</b>	( <i>d</i> --- )	Print double-precision number
<b>D.R</b>	( <i>d n</i> --- )	Print double-precision number right-justified in field of width <i>n</i>
<b>CR</b>	( --- )	Perform a Carriage Return/Line Feed

<sup>3</sup> This loop may be exited by executing **R> DROP** one level below.

<b>SPACE</b>	( --- )	Type 1 space
<b>SPACES</b>	( <i>n</i> --- )	Type <i>n</i> spaces
<b>."</b>	( --- )	Print a string terminated by "
<b>TYPE</b>	( <i>addr n</i> --- )	Type <i>n</i> characters from <i>addr</i> to terminal
<b>COUNT</b>	( <i>addr</i> --- <i>addr+1 n</i> )	Move string length from <i>addr</i> to stack
<b>?TERMINAL</b>	( --- <i>flag</i> )	Test if <b>&lt;BREAK&gt;</b> ( <b>&lt;CLEAR&gt;</b> on TI-99/4A) pressed
<b>?KEY</b>	( --- <i>n</i> )	Read keyboard. If no key pressed, <i>n</i> = 0 else <i>n</i> = ASCII keycode.
<b>KEY</b>	( --- <i>c</i> )	Wait for a keystroke and put its ASCII value on the stack.
<b>EMIT</b>	( <i>c</i> --- )	Type character from stack to terminal
<b>EXPECT</b>	( <i>addr n</i> --- )	Read <i>n</i> characters (or until <b>CR</b> ) from terminal to <i>addr</i>
<b>WORD</b>	( <i>c</i> --- )	Read one word from input stream delimited by <i>c</i>

## 2.7 Numeric Formatting

Advanced numeric formatting control is possible with the following words:

<b>NUMBER</b>	( <i>addr</i> --- <i>d</i> )	Convert string at <i>addr</i> to <i>d</i> number
<b>&lt;#</b>	( --- )	Start output string conversion
<b>#</b>	( <i>d</i> <sub>1</sub> --- <i>d</i> <sub>2</sub> )	Convert next, least-significant digit of <i>d</i> <sub>1</sub> leaving <i>d</i> <sub>2</sub>
<b>#S</b>	( <i>d</i> --- 0 0 )	Convert all significant digits from right to left
<b>SIGN</b>	( <i>n d</i> --- <i>d</i> )	Insert sign of <i>n</i> into number
<b>#&gt;</b>	( <i>d</i> --- <i>addr u</i> )	Terminate conversion, ready for <b>TYPE</b>
<b>HOLD</b>	( <i>c</i> --- )	Insert ASCII character <i>c</i> into string

## 2.8 Disk-Related Words

The following words assist in maintaining source code on disk as well as implementing the Forth virtual memory capability:

<b>LIST</b>	( <i>n</i> --- )	List screen <i>n</i> to terminal
<b>LOAD</b>	( <i>n</i> --- )	Compile or execute screen <i>n</i>
<b>BLOCK</b>	( <i>n</i> --- <i>addr</i> )	Leave address of block <i>n</i> , reading it from disk if necessary
<b>B/BUF</b>	( --- <i>n</i> )	Constant giving disk block size in bytes
<b>BLK</b>	( --- <i>addr</i> )	User variable containing current block number (contains 0 for terminal input)

<b>SCR</b>	( --- <i>addr</i> )	User variable containing screen number most recently referenced by <b>LIST</b> or <b>EDIT</b>
<b>UPDATE</b>	( --- )	Mark last buffer accessed as updated
<b>FLUSH</b>	( --- )	Write all updated buffers to disk
<b>EMPTY-BUFFERS</b>	( --- )	Erase all buffers

## 2.9 Defining Words

The following are defining words. They are used not only to create new Forth words but in the case of **<BUILDS ... DOES>** and **;CODE** to create new defining words.

<b>:</b>	<b>xxx</b>	( --- )	Begin colon definition of <b>xxx</b>
<b>;</b>		( --- )	End colon definition
<b>VARIABLE</b>	<b>xxx</b>	( <i>n</i> --- )	Create variable with initial value <i>n</i>
	<b>xxx</b>	( --- <i>addr</i> )	Returns address when executed
<b>CONSTANT</b>	<b>xxx</b>	( <i>n</i> --- )	Create constant with value <i>n</i>
	<b>xxx</b>	( --- <i>n</i> )	Returns <i>n</i> when executed
<b>CODE</b>	<b>xxx</b>	( --- )	Begin definition of assembly language primitive named <b>xxx</b>
<b>;CODE</b>		( --- )	Create new defining word with execution-time code routine
<b>&lt;BUILDS ... DOES&gt;</b>			Create new defining word using high level Forth.
	<b>DOES&gt;</b>	( --- <i>addr</i> )	

## 2.10 Miscellaneous Words

The following words are relatively common but don't fit well in any of the above categories:

<b>CONTEXT</b>	( --- <i>addr</i> )	Leave address of pointer to context vocabulary (searched first)
<b>CURRENT</b>	( --- <i>addr</i> )	Leave address of pointer to current vocabulary (new definitions placed there)
<b>FORTH</b>	( --- )	Set <b>CONTEXT</b> to main Forth vocabulary
<b>DEFINITIONS</b>	( --- )	Set <b>CURRENT</b> to <b>CONTEXT</b>
<b>VOCABULARY</b>	<b>xxx</b>	( --- ) Define new vocabulary
<b>(</b>	( --- )	Begin comment. Terminated by <b>)</b>
<b>FORGET</b>	<b>xxx</b>	( --- ) Forget all definitions back to and including <b>xxx</b>
<b>ABORT</b>	( --- )	Error termination

---

' <b>xxx</b>	( --- <i>addr</i> )	Leave address of <b>xxx</b> . If compiling compile address. (tick)
<b>HERE</b>	( --- <i>addr</i> )	Leaves address of next unused byte in the dictionary
<b>PAD</b>	( --- <i>addr</i> )	Leaves address of scratch area
<b>IN</b>	( --- <i>addr</i> )	User variable containing offset into input buffer
<b>SP@</b>	( --- <i>addr</i> )	Leaves address of top stack item
<b>ALLOT</b>	( <i>n</i> --- )	Leave <i>n</i> -byte gap in dictionary
<b>,</b>	( <i>n</i> --- )	Compile <i>n</i> into the dictionary (comma)

Several Forth screens on the TI Forth System disk serve special purposes. Forth screen 0 may not be modified because it is used by the disk Device Service Routine (DSR) to locate the object code of the Forth kernel. Forth screen 3 is the **BOOT** screen (see **BOOT** in Appendix D), and Forth screens 4 and 5 contain error messages used by several Forth words. Any disk placed in drive 0 (DR0) must contain a copy of Forth screens 4 and 5.

Many additional words are available in TI Forth. The user should consult the remaining chapters in this manual as well as the glossary (Appendix D) and Appendix F for a complete description. Most of these words are disk-resident and must be loaded by the user via the load options, which are viewable by typing **MENU** , before they become available.

## 3 How to Use the Forth Editor

Words introduced in this chapter:

<b>CLEAR</b>	<b>FLUSH</b>
<b>ED@</b>	<b>TEXT</b>
<b>EDIT</b>	<b>WHERE</b>

In the Forth language, programs are divided into screens or blocks. Each Forth screen is 16 lines of 64 characters and has a number associated with it. A TI-99/4A disk holds 90 Forth screens (numbered 0 – 89), however, Forth screen 0 is special and is usually not used. A program may occupy as many Forth screens as necessary.

You must read Chapter 5, “System Synonyms and Miscellaneous Utilities” and correctly format your data disk before using the editor. Disks initialized by the disk manager are acceptable. After loading Forth from the System disk, place the System disk in DR1 (2<sup>nd</sup> drive) and your Forth disk in DR0 (1<sup>st</sup> drive). It is necessary to copy Forth screens 4 and 5 from the Forth System disk onto your Forth disk. These screens contain the error messages. If you have a two-drive system, see the instructions for **SCOPY** and **SMOVE** in Chapter 5 for directions on how to do this.

If you have a one-drive system, however, this procedure is more complicated. The following diagram illustrates the process used to copy parts of a Forth disk or an entire Forth disk with a one drive system.

START: With original diskette in your drive and type:

**FLUSH**

LOOP: Type these lines:

*scr* **BLOCK DROP UPDATE**

•  
•  
•

*scr* **BLOCK DROP UPDATE**

}  
}  
}

up to 5 screens because the system  
has 5 disk buffers

Switch to backup diskette and type:

**FLUSH**

Go back to LOOP if you need to copy more screens.

Now you are ready to begin editing your Forth disk.

**CAUTION:** Do *not* edit your System disk. It is a hybrid disk containing both TI-99/4A files and Forth screens. Editing the disk may destroy its integrity!

### 3.1 Forth Screen Layout Caveat

As indicated above, Forth screens are laid out in 16 lines of 64 characters each. However, you should be aware that the lines have no actual delimiters, *i.e.*, there are no carriage-return or line-feed characters at the end of a Forth-screen line. This means that one line wraps around to the next line with no intervening white-space such that a word ending on one line will be concatenated with a word that starts on the next line if there is no intervening space. This will usually be nonsense to the system and generate an error message when the screen is loaded, indicating that the unintended word has not been defined. Worse, it can result in an unintended existing word such as **-DUP** instead of **- DUP** or **+LOOP** instead of **+ LOOP**.

### 3.2 The Two TI Forth Editors

There are two Forth editors available on the TI Forth System disk. The first, which is loaded by **-EDITOR**, operates in **TEXT** mode. It will be referred to as the 40-column editor<sup>4</sup>. Each Forth screen is displayed in two halves (left and right) in normal sized characters.

The second, which is loaded by **-64SUPPORT**, operates in bit map mode. It allows you to view an entire Forth screen at once; however, the characters are very small. It will be referred to as the 64-column editor.

Only one editor may be in memory at any time. Load whichever you prefer. Editing instructions are identical for each.

### 3.3 Editing Instructions

Initialization fills each Forth screen with non-printable characters. These characters appear as solid white squares on the terminal when you are using the 40-column editor and as unidentifiable characters in the 64-column editor. A Forth screen must be filled with blanks before it can be used. Typing a Forth screen number and **CLEAR** will fill a Forth screen with blanks.

#### 1 CLEAR

will prepare Forth screen 1 for use by the editor.

You may begin writing on Forth screen 1 or on any Forth screen you wish. To bring a Forth screen from the disk into the editor, type the Forth screen number followed by the word **EDIT**.

#### 1 EDIT

The above instruction will bring the contents of Forth screen 1 into view. If you did not **CLEAR** the screen before entering the editor, the screen will appear to be a block of undefined characters. You must exit the editor temporarily and clear the screen on the disk before you can write to it. To exit the editor, press the **<BACK>** (**<FCTN+9>**) function key on your keyboard. To clear the screen, type the screen number and the word **CLEAR**.

To re-enter the editor, You do *not* have to type **1 EDIT** again. A special Forth word,

**ED@**

<sup>4</sup> The 40-column Forth editor may only be used when the computer is in **TEXT** mode (see Chapter 6). For example, if the 40-column editor is loaded, don't type **EDIT** while you are in **SPLIT** or **SPLIT2** mode.

will return you to the last screen you were editing.

Upon entering the editor, the cursor is located in column 1 of line 0. It is customary to use line 0 for a comment describing the contents of that screen. Type a comment that says “**PRACTICE SCREEN**” or something to that effect. Do not forget that all comments must begin with a “( ”<sup>5</sup> and end with a “)”.

If you are using the 40-column editor, you have probably noticed that only 35 columns ( of the 64 available columns ) are visible on your terminal. To see the rest of the screen, type any characters on line 1 until you reach the right margin. Now type a few more characters. Notice that the screen is now displaying columns 30 – 64. Press **<ENTER>** to move to the beginning of the next line.

The function keys on your keyboard each perform a special editing function.

key	function
<b>&lt;FCTN+S&gt;</b> , (←)	moves the cursor one position to the left.
<b>&lt;FCTN+D&gt;</b> , (→)	moves the cursor one position to the right.
<b>&lt;FCTN+E&gt;</b> , (↑)	moves the cursor up one position.
<b>&lt;FCTN+X&gt;</b> , (↓)	moves the cursor down one position.
<b>&lt;DELETE&gt;</b> ( <b>&lt;FCTN+1&gt;</b> )	deletes the character on which the cursor is placed.
<b>&lt;INSERT&gt;</b> ( <b>&lt;FCTN+2&gt;</b> )	inserts a space to the left of the cursor moving the rest of the line right one space. Characters may be lost off the end of the line.
<b>&lt;AID&gt;</b> ( <b>&lt;FCTN+7&gt;</b> )	erases from the cursor to the end of a line and saves the erased characters in <b>PAD</b> . They may be placed at the beginning of a new line by pressing <b>&lt;REDO&gt;</b> . <b>&lt;REDO&gt;</b> inserts a line just above where the cursor is and places the contents of <b>PAD</b> there.
<b>&lt;BEGIN&gt;</b> ( <b>&lt;FCTN+5&gt;</b> )	<b>40-column editor:</b> moves the cursor 28 positions to the right if the cursor is on the left half of a Forth screen. Otherwise, it moves the cursor 28 positions to the left. This key can be used to toggle between the left and right half of a screen. <b>64-column editor:</b> places the cursor in the upper left corner
<b>&lt;ERASE&gt;</b> ( <b>&lt;FCTN+3&gt;</b> )	are used in combination to pick up lines and move them elsewhere on the screen. <b>&lt;ERASE&gt;</b> picks up one line while erasing it from view. <b>&lt;REDO&gt;</b> inserts this line just above the line on which the cursor is placed. Both <b>ERASE</b> and <b>&lt;REDO&gt;</b> may be used repeatedly to erase several lines from view or to insert multiple copies of a line.
<b>&lt;REDO&gt;</b> ( <b>&lt;FCTN+8&gt;</b> )	
<b>&lt;CTRL+8&gt;</b>	will insert a blank line just above the line the cursor is on.
<b>&lt;CTRL+V&gt;</b>	will tab forward by words.
<b>&lt;FCTN+V&gt;</b>	will tab backwards by words.

5 The left parenthesis *must* be followed by at least 1 space. Press **<ENTER>** to move to the next line.

Experiment with these features until you feel you understand each of their functions. Erase the line you typed from the screen and type a sample program for practice.

The Forth editor allows you to move forward or backward a screen without leaving the editor. Pressing **<CLEAR>** (**<FCTN+4>**) will read in the succeeding screen. Pressing **<PROCEED>** (**<FCTN+6>**) will read in the preceding screen.

If an error occurs during a **LOAD** command, typing the word **WHERE** will bring you back into the editor and place the cursor at the exact point the error occurred.

The word **FLUSH** is used to force the disk buffers that contain data no longer consistent with the copy on disk to be written to the disk. Use this word at the end of an editing session to be certain your changes are written to the disk.

One last note about Forth screens: Though your word definitions can span more than one screen, you should try to insure that any given word is defined on a single screen. This aids in clarity and the good Forth-programming practice of keeping word definitions short.



## 4 Memory Maps

The following diagrams illustrate the memory allocation in the TI-99/4A system. For more detailed information, see the *Editor/Assembler Manual*.<sup>6</sup>

The VDP memory can be configured in many ways by the user. The TI Forth system provides the ability to set up this memory for each of the VDP's 4 modes of operation ( Text, Graphics, Multicolor And Graphics2 ). The allocation of memory for these modes is shown on the VDP Memory Map. The first three modes are shown on the left half of the figure, the Graphics2 mode on the right half. The area at **03C0h** is used by the transcendental functions in all modes for a rollout area. If transcendentials are used during Graphics2 (bit-map) mode, this portion of the color table must be saved by the user before using the transcendental function and restored afterward. Note that the VDP RAM is accessed from the 9900 only through a memory mapped port and is not directly in the processor's address space.

The only CPU RAM on a true 16-bit data bus is in the console at **8300h**. Because this is the fastest RAM in the system, the Forth Workspace and the most frequently executed code of the interpreter are placed in this area to maximize the speed of the TI Forth system. The use of the remainder of the RAM in this area is dictated by the TI-99/4A's resident operating system.

The 32K byte memory expansion is divided into an 8K piece at **2000h** and a 24K piece at **A000h**. The small piece contains BIOS and utility support for TI Forth as well as 5K of disk buffers, the Return Stack, and the User Variable area. The large piece of this RAM contains the dictionary, the Parameter Stack and the Terminal Input Buffer.

### 4.1 VDP Memory Map

Address				Address
<b>0000h</b>	Graphics & Multicolor Screen Image Table	Text Mode Screen Image Table	Bit Map Color Table	<b>1800h</b>
<b>02FFh</b>	<b>300h</b> bytes			<b>0000h</b>
<b>0300h</b>	Sprite Attribute List			
<b>037Fh</b>	<b>80h</b>			
<b>0380h</b>	Color Table			
<b>039Fh</b>	<b>20h</b>			
<b>03A0h</b>	Unused			
<b>03BFh</b>	<b>20h</b>			
<b>03C0h</b>	VDP Rollout Area			
<b>03DFh</b>	<b>20h</b>			
<b>03E0h</b>	Stack for VSPTR			
<b>045Fh</b>	<b>80h</b>			
<b>0460h</b>	PABS etc.			
<b>077Fh</b>	<b>320h</b>			

<sup>6</sup> Hexadecimal (base 16) notation for integers in this manual is indicated when a string of 1 – 4 hexadecimal digits (**0** – **9**, **A** – **F**) is followed by 'h'. For example, **2F0Eh** is a hexadecimal integer equivalent in value to decimal integer 12046 and **Ah** is decimal 10. The 'h' is never typed into the Forth terminal or on Forth screens. It is used in this manual only to avoid confusion. The notation used in the *Editor/Assembler Manual* (use of a preceding '>' instead of a trailing 'h') is only used in Chapter 9 for the conventional assembler examples, where it is required as input to the Editor/Assembler module.

Address		Address
<b>0780h</b>	Sprite Motion Table <b>80h</b>	
<b>07FFh</b>		
<b>0800h</b>	Pattern Descriptor Table	
<b>0BFFh</b>	Sprite Descriptor Table	
	0 – 127 <b>400h</b>	
<b>0C00h</b>	128 – 255 <b>400h</b>	
<b>0FFFh</b>		
<b>1000h</b>	Forth's Disk Buffer	
<b>13FFh</b>	(4 sectors) <b>400h</b>	
<b>1400h</b>	Unused <b>21D8h</b>	<b>17FFh</b>
		<b>1800h</b>
		<b>1AFFh</b>
		<b>1B00h</b>
		<b>1BFFh</b>
		<b>1C00h</b>
		<b>1FFFh</b>
<b>35D7h</b>		<b>2000h</b>
<b>35D8h</b>	Disk Buffering Region for 3 Simultaneous Disk Files <b>A28h</b>	<b>37FFh</b>
		<b>3800h</b>
		<b>387Fh</b>
		<b>3880h</b>
		<b>39D9h</b>
		<b>39DAh</b>
<b>3FFFh</b>		<b>3FFFh</b>

## 4.2 CPU Memory

Address	
<b>0000h</b>	Console ROM
<b>1FFFh</b>	
<b>2000h</b>	Low Memory Expansion
<b>3FFFh</b>	Loader, Your Program, REF/DEF Table
<b>4000h</b>	Peripheral ROMs for DSRs
<b>5FFFh</b>	
<b>6000h</b>	Unavailable—ROM in Command Modules
<b>7FFFh</b>	
<b>8000h</b>	Memory-mapped Devices for VDP, GROM, SOUND, SPEECH. CPU RAM at
<b>9FFFh</b>	<b>8300h – 83FFh</b>
<b>A000h</b>	High Memory Expansion
	Your Program
<b>FFFFh</b>	

### 4.3 CPU RAM Pad

Address <sup>7</sup>	
<b>8300h</b> <b>831Fh</b>	Forth's Workspace
<b>832Eh</b> <b>8347h</b>	Forth's Inner Interpreter, etc.
<b>834Ah</b> <b>8351h</b>	FAC (Floating Point Accumulator)
<b>8356h</b> <b>8357h</b>	Subroutine Pointer for DSRs
<b>835Ch</b> <b>8363h</b>	ARG (Floating Point Argument Register)
<b>8370h</b> <b>8371h</b>	Highest Available Address of VDP RAM
<b>8372h</b>	Least Significant Byte of Data Stack Pointer
<b>8373h</b>	Least Significant Byte of Subroutine Stack Pointer
<b>8374h</b>	Keyboard Number to be Scanned
<b>8375h</b>	ASCII Keycode Detected by Scan Routine
<b>8376h</b>	Joystick Y-status
<b>8377h</b>	Joystick X-status
<b>8379h</b>	VDP Interrupt Timer
<b>837Ah</b>	Number of Sprites that can be in Automotion
<b>837Bh</b>	VDP Status Byte    Bit 0 <sup>8</sup> On during VDP Interrupt Bit 1    On when 5 Sprites on a Line Bit 2    On when Sprite Coincidence Bits 3-7   Number of 5 <sup>th</sup> Sprite on a Line
<b>837Ch</b>	GPL Status Byte    Bit 0    High Bit Bit 1    Greater than Bit Bit 2    On when Keystroke Detected (COND) Bit 3    Carry Bit Bit 4    Overflow Bit
<b>8380h</b>	Default Subroutine Stack Address
<b>83A0h</b>	Default Data Stack Address
<b>83C0h</b> <b>83C2h</b>	Random Number Seed (Begin Interrupt Workspace) Flag    Bit 0    Disable All of the Following Bit 1    Disable Sprite Motion Bit 2    Disable Auto Sound Bit 3    Disable System Reset Key (Quit)
<b>83C4h</b>	Link to DSR Hook
<b>83D4h</b>	Contents of VDP Register 1
<b>83E0h</b> <b>83FFh</b>	Begin GPL Workspace

<sup>7</sup> Locations omitted are not used by Forth, but may be used by system routines.

<sup>8</sup> Bit 0 = high order bit.

### 4.4 Low Memory Expansion

<b>2000h</b> <b>200Fh</b>	XML Vectors	<b>0010h</b> bytes
<b>2010h</b> <b>3423h</b>	Disk Buffers	<b>1414h</b>
<b>3424h</b> <b>397Fh</b>	99/4 Support for Forth	<b>055Ch</b>
<b>3980h</b> <b>39FFh</b>	User Variable Area	<b>0080h</b>
<b>3A00h</b> <b>3CD9h</b>	Assembler Support	<b>020Ah</b>
<b>3CDAh</b> <b>3FFFh</b>	<div style="text-align: center;"> ↑  ↑  Return Stack </div>	<b>0326h</b>

### 4.5 High Memory Expansion

<b>A000h</b> <b>BC7Fh</b>	Resident Forth Vocabulary	<b>1C80h</b>
<b>BC80h</b>	User Dictionary Space <div style="text-align: center;"> ↓  ↓  ↑  ↑ </div>	<b>4320h</b>
<b>FF9Fh</b>	Parameter Stack	
<b>FFA0h</b> <b>FFF1h</b>	Terminal Input Buffer	<b>0052h</b>

## 5 System Synonyms and Miscellaneous Utilities

Words introduced in this chapter:

<b>!"</b>	<b>MYSELF</b>	<b>UNTRACE</b>
<b>.S</b>	<b>RANDOMIZE</b>	<b>VAND</b>
<b>: (traceable)</b>	<b>RND</b>	<b>VFILL</b>
<b>CLS</b>	<b>RNDW</b>	<b>VLIST</b>
<b>DISK-HEAD</b>	<b>SCOPY</b>	<b>VMBR</b>
<b>DSRLNK</b>	<b>SEED</b>	<b>VMBW</b>
<b>DTEST</b>	<b>SMOVE</b>	<b>VOR</b>
<b>DUMP</b>	<b>TRACE</b>	<b>VSBR</b>
<b>FORMAT-DISK</b>	<b>TRIAD</b>	<b>VSBW</b>
<b>FORTH-COPY</b>	<b>TRIADS</b>	<b>VWTR</b>
<b>GPLLNK</b>	<b>TROFF</b>	<b>VXOR</b>
<b>INDEX</b>	<b>TRON</b>	<b>VMLLNK</b>

Several utilities are available to give you simple access to many resources of the TI-99/4A Home Computer. These are defined as system synonyms.

Also included in this chapter are several disk utilities, special trace routines, random number generators and a special routine that allows recursion.

The descriptions that follow in tabular form include the abbreviation “instr” for “instruction”.

### 5.1 System Synonyms

The system synonyms are loaded by typing the TI Forth **MENU** option, **-SYNONYMS**. These utilities allow you to

- change the display;
- access the Device Service Routines for peripheral devices such as RS232 interfaces and disk drives;
- link your program to GPL and Assembler routines; and
- perform operations on VDP memory locations.

### 5.1.1 VDP RAM Read/Write

The first group of instructions enables you to read from and write to VDP RAM. Each of the following Forth words implements the Editor/Assembler utility with the same name.

**VSBW** (*b vaddr ---*)

Writes a single byte to VDP RAM. It requires 2 parameters on the stack: a byte *b* to be written and a VDP address *vaddr*.

	base	byte	vaddr	instr
HEX	A3	380	VSBW	

The above line, when interpreted will change the base to hexadecimal, push **A3h** and **380h** onto the stack and, when **VSBW** executes, places the value **A3h** into VDP address **380h**.

**VMBW** (*addr vaddr count ---*)

Writes multiple bytes to VDP RAM. You must first place on the stack a source address at which the bytes to be written are located. This must be followed by a VDP address ( or destination ) and the number of bytes to be written.

	base	addr	vaddr	count	instr
HEX	PAD	808	4	VMBW	

reads 4 bytes from PAD and writes them into VDP RAM beginning at **808h**.

**VSBR** (*vaddr --- byte*)

Reads a single byte from VDP RAM and places it on the stack. A VDP address is the only parameter required.

	base	vaddr	instr
HEX	781	VSBR	

places the contents of VDP address **781h** on the stack.

**VMBR** (*vaddr addr count ---*)

Reads multiple bytes from VDP and places them at a specified address. You must specify the VDP source address, a destination address and a byte count.

	base	vaddr	addr	count	instr
HEX	300	PAD	20	VMBR	

reads 32 bytes beginning at **300h** and stores them into PAD.

**VFILL** (*vaddr count byte ---*)

If you wish to fill a group of consecutive VDP memory locations with a particular byte, a **VFILL** instruction is available. You must specify a beginning VDP address, a count and the byte you wish to write into each location.

base	vaddr	count	byte	instr
HEX	300	20	0	VFILL

fills 32 (20h) locations, starting at 300h, with zeroes.

### 5.1.2 Extended Utilities: GPLLNK, XMLLNK AND DSRLNK

The next group of instructions allows you to implement the Editor/Assembler instructions GPLLNK, XMLLNK and DSRLNK. To assist the user, the Forth instructions have the same names as the Editor/Assembler utilities. Consult the *Editor/Assembler Manual* for more details.

**GPLLNK** ( addr --- )

Allows you to link your program to Graphics Programming Language (GPL) routines. You must place on the stack the address of the GPL routine to which you wish to link.

base	addr	instr
HEX	16	GPLLNK

branches to the GPL routine located at 16h which loads the standard character set into VDP RAM. It then returns to your program.

**XMLLNK** ( addr --- )

Allows you to link a Forth program to a routine in ROM or to branch to a routine located in the Memory Expansion unit. The instruction expects to find a ROM address on the stack.

base	addr	instr
HEX	800	XMLLNK

accesses the Floating Point multiplication routine, located in ROM at 800h, and returns to your program.

**DSRLNK** ( --- )

Links a Forth program to any Device Service Routine (DSR) in ROM. Before this instruction is used, a Peripheral Access Block (PAB) must be set up in VDP RAM. A PAB contains information about the file to be accessed. See the *Editor/Assembler Manual* and Chapter 8 of this manual for additional setup information. **DSRLNK** needs no parameters on the stack.

The Editor/Assembler version of DSRLNK also allows linkage with a subroutine, but the TI Forth version does not. If you need this functionality, you might define the following word in decimal mode (**BASE** contains Ah):

**: DSRLNK-SP 10 14 SYSTEM ;**

See the *Editor/Assembler Manual* for details on this form of the call to the DSRLNK utility. You will also need to consult the DSR's specifications because this form of access is at a lower level, with each subroutine often requiring information that differs from the PAB set up for **DSRLNK**.

### 5.1.3 VDP Write-Only Registers

The VDP contains 8 special write-only registers. In the Editor/Assembler, a **VWTR** instruction is used to write values into these registers. The Forth word **VWTR** implements this instruction.

**VWTR**            ( *b n ---* )

**VWTR** requires 2 parameters; a byte *b* to be written and a VDP register number *n*.

base	<i>b</i>	<i>n</i>	instr
HEX	F5	7	<b>VWTR</b>

The above instruction writes **F5h** into VDP write only register number 7. This particular register controls the foreground and background colors in text mode. Executing the above instruction will change the foreground color to white and the background color to light blue.

### 5.1.4 VDP RAM Single-Byte Logical Operations

**VAND** , **VOR** and **VXOR**            ( *b vaddr ---* )

The Forth instructions **VAND** , **VOR** and **VXOR** greatly simplify the task of performing a logical operation on a single byte in VDP RAM. Normally, 3 programming steps would be required: a read from VDP RAM, an operation, and a write back into VDP RAM. The above instructions get the job done in a single step. Each of these words requires 2 parameters, a byte *b* to be used as the second operand and the VDP address *vaddr* at which to perform the operation. The result of the operation is placed back into *vaddr*.

base	<i>b</i>	<i>vaddr</i>	instr
HEX	F0	804	<b>VAND</b>
HEX	F0	804	<b>VOR</b>
HEX	F0	804	<b>VXOR</b>

Each of the above instructions reads the byte stored at **804h** in VDP RAM, performs an AND, OR or XOR on that byte and **F0h**, and places the result back into VDP RAM at **804h**.

## 5.2 Disk Utilities

The TI Forth system was designed to be used with 90 screens per disk, *i.e.*, with 90 KB, single-sided, single-density (SSSD) disks. The system easily scales up to other disk formats<sup>9</sup>, except for some of the disk utilities in this section: **FORTH-COPY** , **DTEST** , **DISK-HEAD** and **FORMAT-DISK** are hardwired to use 90 KB disks. **FORTH-COPY** and **DTEST** require minor changes in the word

<sup>9</sup> See Appendix K for a detailed discussion of disk format.



definitions to change the 90-screen limit per disk (See Forth screen 39). Changing **DISK-HEAD** to work is a lot more complicated! It requires low-level knowledge of the format of TI disks to modify its definition (See Forth screen 40). **FORMAT-DISK** is part of the resident TI Forth vocabulary, making it wiser to use other means of formatting disks rather than attempting to re-write the definition for this word, but see Appendix L.

### 5.2.1 Disk Formatting Utility

**FORMAT-DISK**            ( *n* --- )

**FORMAT-DISK** is one of the system utilities loaded by the **-SYNONYMS** option. Any disk that you wish to use with the Forth system must first be properly formatted. Place the disk in a disk drive and place the number of that disk drive on the stack. TI Forth numbers disk drives beginning with 0, therefore, if the new disk is in the first drive, put a 0 on the stack, *etc.* Next, type **FORMAT-DISK**.

#### 0 **FORMAT-DISK**

will initialize the disk in DR0, thus preparing it for use by the Forth system. Disks initialized by the TI Disk Manager are properly formatted and may be used. **FORMAT-DISK** assumes 90 KB, SSSD TI disks.

### 5.2.2 Disk and Screen Copying Utilities

The disk and screen copying utilities are loaded by the **-COPY** option.

**DISK-HEAD**            ( --- )

The TI Forth System disk, or any disk which contains a copy of Forth screens 0 thru 19 of the System disk, may be copied with the TI Disk Manager. Any other disk may be copied with the TI Disk Manager only after a special header has been written on it by the TI Forth word **DISK-HEAD**. Please note that you *must* reset the value of the user variable **DISK\_LO** to zero *before* using **DISK-HEAD**. This word writes the volume name "FORTH" on the disk and creates a single file named "SCREENS" of type "DIS/FIX128", *i.e.*, display-type, 128-byte, fixed-length records. The file is set up to fill all available space on the disk.

Any Forth disk (system or screens-only), which can be copied by the TI Disk Manager, can also be accessed from TI BASIC. If you access a Forth disk that contains the Forth kernel, the only file you should access is named "SYS-SCRNS" and record 0 of the file will be located on line 4 of screen 19. Records of length = 128 bytes will proceed thru record 565, which is located on line 14 of screen 89. Record 566 then wraps to line 4 of screen 1. The file ends with record 623 located on line 6 of screen 8.

A Forth disk which does not contain the kernel may also be accessed by TI BASIC, but the location of the records will be different. The file created by **DISK-HEAD** above, named "SCREENS", will begin on line 8 of screen 8 and continue thru record 651 located on line 14 of screen 89. Record 652 begins on line 12 of screen 0 and the file ends with record 713 on line 6 of screen 8.

**FORTH-COPY** ( --- )

To copy an entire 90 KB, SSSD Forth disk without using the TI Disk Manager, you must place the new disk in DR0 and the source disk in DR1. Typing **FORTH-COPY** will copy the entire contents of the disk in DR1 onto the disk in DR0. Please note that you *must* reset the value of the user variable **DISK\_LO** to zero *before* using **FORTH-COPY**. This will allow you to copy screen 0. This is accomplished by executing the following instruction:

**0 DISK\_LO !**

Using **FORTH-COPY** to copy Forth disks that have higher capacity than 90 KB, *e.g.*, 180 KB or 360 KB, requires rewriting the definition of **FORTH-COPY**, as well as changing **DISK\_SIZE** and **DISK\_HI** to accommodate the new disk sizes (see Appendix L).

**SCOPY** ( *scr<sub>1</sub> scr<sub>2</sub>* --- )

You can copy the contents of a single Forth screen from one screen location to another without destroying the original copy by using the **SCOPY** instruction. A source screen number *scr<sub>1</sub>* and a destination screen number *scr<sub>2</sub>* must be specified.

base	<i>scr<sub>1</sub></i>	<i>scr<sub>2</sub></i>	instr
DECIMAL 5		17	<b>SCOPY</b>

will write the contents of screen 5 over the contents of screen 17 without erasing screen 5. The old contents of screen 17 will be destroyed.

**SMOVE** ( *scr<sub>1</sub> scr<sub>2</sub> count* --- )

The **SMOVE** instruction acts as a multiple **SCOPY**. It allows you to copy a group of Forth screens with a single instruction. You must designate a beginning source screen, a beginning destination screen, and the number of screens you wish to copy. When using **SMOVE**, overlapping screen ranges may be used without user concern. The order of the copy is adjusted so that the entire group of screens is moved intact.

base	<i>scr<sub>1</sub></i>	<i>scr<sub>2</sub></i>	<i>count</i>	instr
DECIMAL 11		36	7	<b>SMOVE</b>

will copy screens 11 - 17 over screens 36 - 42 without erasing screens 11 - 17.

Both the **SCOPY** and **SMOVE** instructions can be used to copy screens from one disk drive to another. Assuming that **DISK\_SIZE** ( a user variable which contains the number of screens per disk ) is at its default value of 90, screens 0 - 89 are contained on the disk in DR0, screens 90 -179 are located on the disk in DR1, *etc.* **Note:** To copy screens from one disk drive to another, you must reset the user variable **DISK\_HI**. If you are using two disk drives, its value must be 180 (2 · 90). This is accomplished by executing the following instruction:

**180 DISK\_HI !**

Therefore, to copy screen 6 on DR0 to screen 20 on DR1, you would type:

base	<i>scr<sub>1</sub></i>	<i>scr<sub>2</sub></i>	instr
DECIMAL 6		110	<b>SCOPY</b>

The **SMOVE** instruction is handled in the same manner. Simply use an offset of **DISK\_SIZE** to specify which disk drives you wish to copy to and from.

#### **DTEST** ( --- )

If you have reason to suspect that a 90 KB, SSSD disk has a bad sector or is in some way damaged, a non-destructive disk test is available. The **DTEST** instruction will attempt to read each screen from the disk in DR0. Please note that you *must* reset the value of the user variable **DISK\_LO** to zero *before* using **DTEST**. A screen number will be displayed on your monitor as each screen is read. If execution stops before screen 89 is reached, the problem lies in the last screen displayed. To correct the problem, **CLEAR** that screen and write to it again. This correction will work if the disk surface is intact and if the formatting information has not been damaged. **DTEST** can be rewritten to accommodate more capacious disks (see **FORTH-COPY** above and Appendix L).

### 5.3 Listing Utilities

There are three words on the TI Forth System disk (loaded by the **-PRINT** option) which make listing information from a Forth disk very simple.

#### **TRIAD** ( scr --- )

The first, called **TRIAD**, requires a Forth screen number on the stack. When executed, it will print to an RS232 device the three screens which contain the specified screen, beginning with a screen number evenly divisible by three. Screens that contain non-printable information will be skipped. If your RS232 printer is not on Port 1 and set at 9600 Baud, you must modify the word **SWCH** on your System disk.

#### **TRIADS** ( scr<sub>1</sub> scr<sub>2</sub> --- )

The second instruction, called **TRIADS**, may be thought of as a multiple **TRIAD**. It expects a beginning and an ending screen number on the stack. **TRIADS** performs as many **TRIADS** as necessary to cover the specified range of screens.

#### **INDEX** ( scr<sub>1</sub> scr<sub>2</sub> --- )

The **INDEX** instruction allows you to list to your terminal line 0 (the comment line) of each of a specified range of screens. **INDEX** expects a beginning and an ending screen number on the stack. If you wish to temporarily stop the flow of output in order to read it before it scrolls off the screen, simply press any key. Press any key to start up again. Press **<BREAK>** (**<CLEAR>** or **<FCTN+4>**) to exit execution prematurely.

### 5.4 Debugging

#### 5.4.1 Dump Information to Terminal

Choosing the **-DUMP** option loads three useful TI Forth words for getting information for debugging purposes.

**VLIST** ( --- )

The Forth word **VLIST** lists to your terminal the names of all words currently defined in the **CONTEXT** vocabulary. This instruction requires no parameters and may be halted and started again by pressing any key as with **INDEX** in the previous section.

**DUMP** ( *addr count* --- )

The **DUMP** instruction allows you to list portions of memory to your terminal. **DUMP** requires two parameters: an address and a byte count. For example,

base	<i>addr</i>	<i>count</i>	instr
HEX	2F26	100	DUMP

will list 256 (**100h**) bytes of memory beginning at address **2F26h** to your terminal. Press any key to temporarily stop execution in order to read the information before it scrolls off the screen. Press any key to continue. To exit this routine permanently, press **<BREAK>**.

**.S** ( --- )

The Forth word **.S** allows you to view the parameter stack contents. It may be placed inside a colon definition or executed directly from the keyboard. The word **SP!** should be typed before executing a routine that contains **.S**. This will clear any garbage from the stack. The | symbol is printed to represent the bottom of the stack. The number appearing farthest from the | is the most accessible stack element.

### 5.4.2 Tracing Word Execution

A special set of instructions allows you to trace the execution of any colon definition. Executing the **TRACE** instruction will cause all following colon definitions to be compiled in such a way that they can be traced. In other words, the Forth word **:** takes on a new meaning. To stop compiling under the **TRACE** option, type **UNTRACE**. When you have finished debugging, recompile the routine under the **UNTRACE** option.

After instructions have been compiled under the **TRACE** option, you can trace their execution by typing the word **TRON** before using the instruction. **TRON** activates the trace. If you wish to execute the same instruction without the trace, type **TROFF** before using the instruction.

The actual trace will print the word being traced, along with the stack contents, each time the word is encountered. This shows you what numbers are on the stack just before the traced word is executed. The | symbol is used to represent the bottom of the stack. The number printed closest to the | is the least accessible while the number farthest from the | is the most accessible number on the stack. Here is a sample **TRACE** session:

```

DECIMAL
TRACE ok           (compile next definition with TRACE option)
: CUBE DUP DUP * * ; (routine to be traced)
UNTRACE OK         (don't compile next definition with TRACE option)
: TEST CUBE ROT CUBE ROT CUBE ; ok
TRON ok            (want to execute with a TRACE)
5 6 7 TEST         (put parameters on stack and execute TEST)
CUBE               (TRACE begins)
| 5 6 7            (stack contents upon entering CUBE)

```

```

CUBE
| 6 343 5           (stack contents upon entering CUBE)
CUBE
| 343 125 6 ok

```

### 5.4.3 Recursion

Normally, a Forth word cannot call itself before the definition has been compiled through to a **;** because the **SMUDGE** bit is set. To allow recursion, TI Forth includes the special word **MYSELF**.

**MYSELF** ( --- )

The **MYSELF** instruction places the CFA of the word currently being compiled into its own definition thus allowing a word to call itself.

The following, more complex, **TRACE** example uses a recursive factorial routine for illustration:

```

DECIMAL ok
TRACE ok           (compile following definition under TRACE option)
: FACT DUP 1 > DUP 1 - MYSELF * ENDIF ; ok
UNTRACE ok
TRON ok
5 FACT             (put parameter on stack and execute FACT)
FACT              (TRACE begins)
| 5
FACT
| 5 4
FACT
| 5 4 3
FACT
| 5 4 3 2
FACT
| 5 4 3 2 1 ok
.S                (check final stack contents)
| 120 ok

```

Each time the traced **FACT** routine calls itself, a **TRACE** is executed.

## 5.5 Random Numbers

Two different random number functions are available in TI Forth.

**RND** (  $n_1$  ---  $n_2$  )

The first, **RND**, generates a positive random integer between 0 and a specified range  $n_1$ .

base	$n_1$	instr
<hr/>		
DECIMAL	13	RND

will place on the stack an integer greater than or equal to 0 and less than 13.

**RNDW** ( --- n )

The second random number function, **RNDW** , generates a random word (2 bytes). No range is specified for **RNDW** .

**RNDW**

will place on the stack a number from 0 to FFFFh.

**RANDOMIZE** ( --- )

To guarantee a different sequence of random numbers each time a program is run, the **RANDOMIZE** instruction must be used. **RANDOMIZE** places an unknown seed into the random number generator.

**SEED** ( n --- )

To place a known seed into the random number generator, the **SEED** instruction is used. You must specify the seed value.

**4 SEED**

will place the value 4 into the random number generator seed location.

## 5.6 Miscellaneous Instructions

**!"** ( addr --- )

This word is loaded by the **-COPY** option to be used by **DISK-HEAD** , but is available for your use. It stores a string at a specified address, but does not store the character count, which you would need to use **TYPE** . **!"** expects to find an address on the stack and must be followed by a string terminated with a " . The following instruction places the string "HOW ARE YOU?" at address **PAD** :

**PAD !" HOW ARE YOU?"**

**CLS** ( --- )

**CLS** is loaded by the **-SYNONYMS** option. Use this word to clear the display screen. **CLS** clears the display screen by filling the screen image table with blanks. The screen image table runs from **SCRN\_START** to **SCRN\_END** . **CLS** may be used inside a colon definition or directly from the keyboard. **CLS** will not clear bit-map displays or sprites.

## 6 An Introduction to Graphics

Words introduced in this chapter:

<b>#MOTION</b>	<b>GRAPHICS</b>	<b>SPLIT2</b>
<b>BEEP</b>	<b>GRAPHICS2</b>	<b>SPRCOL</b>
<b>CHAR</b>	<b>HCHAR</b>	<b>SPRDIST</b>
<b>CHARPAT</b>	<b>HONK</b>	<b>SPRDISTXY</b>
<b>COINC</b>	<b>JOYST</b>	<b>SPRGET</b>
<b>COINCALL</b>	<b>LINE</b>	<b>SPRITE</b>
<b>COINCXY</b>	<b>MAGNIFY</b>	<b>SPRPAT</b>
<b>COLOR</b>	<b>MCHAR</b>	<b>SPRPUT</b>
<b>DELALL</b>	<b>MINIT</b>	<b>SSDT</b>
<b>DELSPR</b>	<b>MOTION</b>	<b>TEXT</b>
<b>DOT</b>	<b>MULTI</b>	<b>UNDRAW</b>
<b>DRAW</b>	<b>SCREEN</b>	<b>VCHAR</b>
<b>DTOG</b>	<b>SPCHAR</b>	
<b>GCHAR</b>	<b>SPLIT</b>	

### 6.1 Graphics Modes

The TI Home Computer possesses a broad range of graphics capabilities. Four screen modes are available to the user:

- 1) **Text Mode**—Standard ASCII characters are available, and new characters may be defined. All characters have the same foreground and background color. The screen is 40 columns by 24 lines. Text mode is used by the Forth 40-column screen editor.
- 2) **Graphics Mode**—Standard ASCII characters are available, and new characters may be defined. Each character set may have its own foreground and background color.
- 3) **Multicolor Mode**—The screen is 64 columns by 48 rows. Each standard character position is now 4 smaller boxes which can each have a different color. ASCII characters are not available and new characters cannot be defined.
- 4) **Bit-Map Mode (Graphics2)**—This mode is available only on the TI-99/4A. Bit-map mode allows you to set any pixel on the screen and to change its color within the limits permitted by the 9918a. The screen is 256 columns by 192 rows. Graphics2 mode is used by the 64-column editor.

Sprites (moving graphics) are available in all modes except text. The sprite automotion feature is not available in graphics2, split, or split2 modes.

Two unique graphics modes have been created by using graphics2 mode in a non-standard way. Split and split2 modes allow you to display text while creating bit-map graphics. Split mode sets the top two thirds of the screen in graphics2 mode and places text on the last third. Split2 sets the

top one sixth of the screen as a text window and the rest in graphics2 mode. These modes provide an interactive bit map graphics setting. That is, you can type bit map instructions and watch them execute without changing modes.

You may place the computer in the above modes by executing one of the following instructions:

```
TEXT      ( --- )
GRAPHICS  ( --- )
MULTI     ( --- )
GRAPHICS2 ( --- )
SPLIT     ( --- )
SPLIT2    ( --- )
```

## 6.2 Forth Graphics Words

Many Forth words have been defined to make graphics handling much easier for the user. As many words are mentioned, an annotation will appear underneath them denoting which of the modes they may be used in (T G M B). These denote text, graphics, multicolor and bit-mapped (graphics2, split, split2) modes, respectively.

In several instruction examples, a base ( **HEX** or **DECIMAL** ) is specified. This does not mean that you must be in a particular base in order to use the instruction. It merely illustrates that some instructions are more easily written in hexadecimal than in decimal. It also avoids ambiguity.

## 6.3 Color Changes

The simplest graphics operations involve altering the color of the screen and of character sets. There are 32 character sets ( 0 – 31 ), each containing 8 characters. For example, character set 0 consists of characters 0 – 7, character set 1 consists of characters 8 – 15, etc. Sixteen colors are available on the TI Home Computer.

<b>Color</b>	<b>Hex Value</b>	<b>Color</b>	<b>Hex Value</b>
transparent	0	medium red	8
black	1	light red	9
medium green	2	dark yellow	A
light green	3	light yellow	B
dark blue	4	dark green	C
light blue	5	magenta	D
dark red	6	gray	E
cyan	7	white	F



**SCREEN** (*color ---*)

The Forth word **SCREEN** following one of the above table values will change the screen color to that value. The following example changes the screen to light yellow:

base	color	instr	
<b>HEX</b>	<b>B</b>	<b>SCREEN</b>	or
<b>DECIMAL</b>	<b>11</b>	<b>SCREEN</b>	

(G)

For text mode, the color of the foreground also needs to be set and should be different from the background color so that text is visible. The foreground color must be in the leftmost 4 bits of the byte passed to **SCREEN**. It is easier to compose the byte in hexadecimal than decimal because each half of the byte is one hexadecimal digit. To set the foreground to black (**1**) and the background to light yellow (**Bh**), the following sequence will do the trick:

**HEX 1B SCREEN**

**COLOR** (*fg bg charset ---*)

The foreground and background colors of a character set may also be easily changed:

base	fg	bg	charset	instr	
<b>HEX</b>	<b>4</b>	<b>D</b>	<b>1A</b>	<b>COLOR</b>	or
<b>DECIMAL</b>	<b>4</b>	<b>13</b>	<b>26</b>	<b>COLOR</b>	

(G)

The above instruction will change character set 26 ( characters 208 – 215 ) to have a foreground color of dark blue and a background color of magenta.

## 6.4 Placing Characters on the Screen

**HCHAR** (*col row count char ---*)

To print a character anywhere on the screen and optionally repeat it horizontally, the **HCHAR** instruction is used. You must specify a starting column and row position as well as the number of repetitions and the ASCII code of the character you wish to print.

*Keep in mind that both rows and columns are numbered from zero !!!*

For example,

base	col	row	count	char	instr	
<b>HEX</b>	<b>A</b>	<b>11</b>	<b>5B</b>	<b>2A</b>	<b>HCHAR</b>	or
<b>DECIMAL</b>	<b>10</b>	<b>17</b>	<b>91</b>	<b>42</b>	<b>HCHAR</b>	

(T G)



this character is defined:

	<b>3C66h</b>	<b>DBE7h</b>	<b>E7DBh</b>	<b>663Ch</b>
<b>Rows</b>	0 – 1	2 – 3	4 – 5	6 – 7

**CHAR**      (*n<sub>1</sub> n<sub>2</sub> n<sub>3</sub> n<sub>4</sub> char ---*)

The Forth word **CHAR** is used to create new characters. To assign the above pattern to character number 123, you would type

base	<i>n<sub>1</sub></i>	<i>n<sub>2</sub></i>	<i>n<sub>3</sub></i>	<i>n<sub>4</sub></i>	<i>char</i>	instr	
<b>HEX</b>	<b>3C66</b>	<b>DBE7</b>	<b>E7DB</b>	<b>663C</b>	<b>7B</b>	<b>CHAR</b>	or
<b>DECIMAL</b>	<b>15426</b>	<b>56295</b>	<b>59355</b>	<b>26172</b>	<b>123</b>	<b>CHAR</b>	

(T G)

As you can see, it is more natural to use this instruction in **HEX** than in **DECIMAL**.

**CHARPAT**      (*char --- n<sub>1</sub> n<sub>2</sub> n<sub>3</sub> n<sub>4</sub>*)

To define another character to look like character 65 (“A”), for example, you must first find out what the pattern code for “A” is. To accomplish this, use the **CHARPAT** instruction. This instruction leaves the character definition on the stack in the proper order for a **CHAR** instruction. Study this line of code:

<b>HEX</b>	<b>41</b>	<b>CHARPAT</b>	<b>7E</b>	<b>CHAR</b>	or
<b>DECIMAL</b>	<b>65</b>	<b>CHARPAT</b>	<b>126</b>	<b>CHAR</b>	

(T G)

The above instructions place on the stack the character pattern for “A” and assigns the pattern to character 126. Now both character 65 and 126 have the same shape.

## 6.6 Sprites

Sprites are moving graphics that can be displayed on the screen independently and/or on top of other characters. Thirty-two sprites are available.

### 6.6.1 Magnification

Sprites may be defined in 4 different sizes or magnifications:

Magnification Factor	Description
0	Causes all sprites to be single size and unmagnified. Each sprite is defined only by the character specified and occupies one character position on the screen.

Magnification Factor	Description
1	Causes all sprites to be single size and magnified. Each sprite is defined only by the character specified, but this character expands to fill 4 screen positions.
2	Causes all sprites to be double size and unmagnified. Each sprite is defined by the character specified along with the next 3 characters. The first character number must be divisible by 4. This character becomes the upper left quarter of the sprite, the next characters are the lower left, upper right, lower right respectively. The sprite fills 4 screen positions.
3	Causes all sprites to be double size and magnified. Each sprite is defined by 4 characters as above, but each character is expanded to occupy 4 screen positions. The sprite fills 16 positions.

The default magnification is 0.

**MAGNIFY**      ( *n* --- )

To alter sprite magnification, use the Forth word **MAGNIFY** .

<i>n</i>	instr
<b>2</b>	<b>MAGNIFY</b>
	(G M B)

will change all sprites to double size and unmagnified.

### 6.6.2 Sprite Initialization

**SSDT**      ( *vaddr* --- )

Before you begin defining sprites, you must execute the Forth word **SSDT** which roughly translates, “set Sprite Descriptor Table.” Before executing this instruction, the computer must be set into the VDP mode you wish to use with sprites. Recall that *sprites are not available in text mode*.

You have a choice of overlapping your sprite character definitions with the standard characters in the Pattern Descriptor Table (see VDP Memory Map in Chapter 4) or moving the Sprite Descriptor Table elsewhere in memory. This move is highly recommended to avoid confusion. **2000h** is usually a good location, but any available 2K (**800h**) boundary will do.

base	vaddr	instr	
<b>HEX</b>	<b>2000</b>	<b>SSDT</b>	or
<b>DECIMAL</b>	<b>8192</b>	<b>SSDT</b>	
		(G M B)	

will move the Sprite Descriptor Table to **2000h**. Use the value **800h** with the SSDT instruction if you do not want to move the Descriptor Table.

*Note:* Whether or not you choose to move the table, you must execute this instruction before you can use sprites in your program!!!

### 6.6.3 Using Sprites in Bit-Map Mode

**SATR** ( --- vaddr )

When using sprites in any of the bit-map modes (graphics2, split, split2), a little extra work is required. After entering the desired VDP mode, the location of the Sprite Attribute List must be changed to **3800h** as follows.

**HEX 3800 ' SATR !<sup>10</sup>**

The base address of the Sprite Descriptor Table must also be changed using the **SSDT** instruction. It will be based at the same address as the Sprite Attribute List (**3800h**), but only a few character numbers will be available for sprite patterns. **SPCHAR** may only be used to define patterns 16 – 58. (See following section for information on **SPCHAR**.)

<b>3800h</b>	Sprite Attribute List <b>0080h</b>
<b>3880h</b>	Sprite Patterns 16-58 ( based at <b>3800h</b> )
<b>39D9h</b>	<b>015Ah</b>

### 6.6.4 Creating Sprites

The first task involved in creating sprites is to define the characters you will use to make them. These definitions will be stored in the Sprite Descriptor Table mentioned in the above section.

**SPCHAR** (  $n_1 n_2 n_3 n_4$  char --- )

A word identical in format to **CHAR** is used to store sprite character patterns. If you are using a magnification factor of 2 or 3, do not forget that you must define 4 consecutive characters for *each* sprite. In this case, the character # of the first character must be a multiple of 4.

<sup>10</sup> Bug fix: See Appendix J.

base	$n_1$	$n_2$	$n_3$	$n_4$	$char$	instr	
<b>HEX</b>	<b>0F0F</b>	<b>2424</b>	<b>F0F0</b>	<b>4242</b>	<b>0</b>	<b>SPCHAR</b>	or
<b>DECIMAL</b>	<b>3855</b>	<b>9252</b>	<b>61680</b>	<b>8770</b>	<b>0</b>	<b>SPCHAR</b>	

(G M B)

defines character 0 in the Sprite Descriptor Table. If your Pattern and Sprite Descriptor Tables overlap, use character numbers below 127 with caution.

**SPRITE** (*dotrow dotcol color char spr ---*)

To define a sprite, you must specify the dot column and dot row at which its upper left corner will be located, its color, a character number and a sprite number ( 0 - 31 ).

base	$dotcol$	$dotrow$	$color$	$char$	$spr$	instr	
<b>HEX</b>	<b>6B</b>	<b>4C</b>	<b>5</b>	<b>10</b>	<b>1</b>	<b>SPRITE</b>	or
<b>DECIMAL</b>	<b>107</b>	<b>76</b>	<b>5</b>	<b>16</b>	<b>1</b>	<b>SPRITE</b>	

(G M B)

defines sprite #1 to be located at column 107 and row 76, to be light blue and to begin with character 16. Its size will depend on the magnification factor.

Once a sprite has been created, changing its pattern, color or location is trivial.

**SPRPAT** (*char spr ---*)

base	$char$	$spr$	instr	
<b>HEX</b>	<b>14</b>	<b>1</b>	<b>SPRPAT</b>	or
<b>DECIMAL</b>	<b>20</b>	<b>1</b>	<b>SPRPAT</b>	

(G M B)

will change the pattern of sprite #1 to character number 20.

**SPRCOL** (*color spr ---*)

base	$color$	$spr$	instr	
<b>HEX</b>	<b>C</b>	<b>2</b>	<b>SPRCOL</b>	or
<b>DECIMAL</b>	<b>12</b>	<b>2</b>	<b>SPRCOL</b>	

(G M B)

will change the color of sprite #2 to dark green.

**SPRPUT** (*dotcol dotrow spr ---*)

base	<i>dotcol</i>	<i>dotrow</i>	<i>spr</i>	instr	
<b>HEX</b>	<b>28</b>	<b>4F</b>	<b>1</b>	<b>SPRPUT</b>	or
<b>DECIMAL</b>	<b>40</b>	<b>79</b>	<b>1</b>	<b>SPRPUT</b>	

(G M B)

will place sprite #1 at column 40 and row 79.

### 6.6.5 Sprite Automotion

In graphics or multicolor mode, sprites may be set in automotion. That is, having assigned them horizontal and vertical velocities and set them in motion, they will continue moving with no further instruction. Sprite automotion is only available in graphics and multicolor modes.

Velocities from 0 to **7Fh** are positive velocities (down for vertical and right for horizontal), and from **FFh** to **80h** are taken as two's complement negative velocities.

**MOTION** (*xvel yvel spr ---*)

base	<i>xvel</i>	<i>yvel</i>	<i>spr</i>	instr	
<b>HEX</b>	<b>FC</b>	<b>6</b>	<b>1</b>	<b>MOTION</b>	or
<b>DECIMAL</b>	<b>-4</b>	<b>6</b>	<b>1</b>	<b>MOTION</b>	

(G M)

will assign sprite #1 a horizontal velocity of -4 and a vertical velocity of 6, but will not actually set them into motion.

**#MOTION** (*n ---*)

After you assign each sprite you want to use a velocity, you must execute the word **#MOTION** to set the sprites in motion. **#MOTION** expects to find on the stack the highest sprite number you are using + 1.

<i>n</i>	instr
<b>6</b>	<b>#MOTION</b>

(G M)

will set sprites #0 - #5 in motion.

<i>n</i>	instr
<b>0</b>	<b>#MOTION</b>

will stop all sprite automotion, but motion will resume when another **#MOTION** instruction is executed.

**SPRGET** (*spr --- dotcol dotrow*)

Once a sprite is in motion, you may wish to find out its horizontal and vertical position on the screen at a given time.

<i>spr</i>	<i>instr</i>
<b>2</b>	<b>SPRGET</b>

(G M B)

will return on the stack the horizontal position of sprite #2 underneath the vertical position. The sprite does *not* have to be in automotion to use this instruction.

### 6.6.6 Distance and Coincidences between Sprites

It is possible to determine the distance  $d$  between two sprites or between a sprite and a point on the screen. This capability comes in handy when writing game programs. The actual value returned by each of the Forth words, **SPRDIST** and **SPRDISTXY**, is  $d^2$ . Distance  $d$  is the hypotenuse of the right triangle formed by joining the line segments,  $d$ ,  $x_2 - x_1$  (the horizontal  $x$ -distance difference in dot columns) and  $y_2 - y_1$  (the vertical  $y$ -distance difference in dot rows). The squared distance between the two sprites or the sprite and screen point is calculated by squaring the  $x$ -distance difference and adding that to the square of the  $y$ -distance difference, *i.e.*,  $d^2 = (x_2 - x_1)^2 + (y_2 - y_1)^2$ .

**SPRDIST**      (*spr<sub>1</sub> spr<sub>2</sub> --- n*)

<i>spr<sub>1</sub></i>	<i>spr<sub>2</sub></i>	<i>instr</i>
<b>2</b>	<b>4</b>	<b>SPRDIST</b>

(G M B)

returns on the stack the square of the distance between sprite #2 and sprite #4.

**SPRDISTXY**      (*dotcol dotrow spr --- n*)

<i>base</i>	<i>dotcol</i>	<i>dotrow</i>	<i>spr</i>	<i>instr</i>
<b>DECIMAL</b>	<b>65</b>	<b>21</b>	<b>5</b>	<b>SPRDISTXY</b>

(G M B)

returns the square of the distance between sprite #5 and the point (65,21).

A coincidence occurs when two sprites become positioned directly on top of one another. That is, their upper left corners reside at the same point. Because this condition rarely occurs when sprites are in automotion you can set a tolerance limit for coincidence detection. For example, a tolerance of 3 would report a coincidence whenever the two sprites upper left corners came within 3 dot positions of each other.

**COINC**      (*spr<sub>1</sub> spr<sub>2</sub> tol --- flag*)

To find a coincidence between two sprites, the Forth word **COINC** is used.

<i>spr<sub>1</sub></i>	<i>spr<sub>2</sub></i>	<i>tol</i>	<i>instr</i>
<b>7</b>	<b>9</b>	<b>2</b>	<b>COINC</b>

(G M B)



will detect a coincidence between sprites #7 and #9 if their upper left corners passed within 2 dot positions of each other. If a coincidence is found, a true flag is left on the stack. If not, a false flag is left.

**COINCXY** ( *dotcol dotrow spr tol --- flag* )

Detecting a coincidence between a sprite and a point is similar.

base	<i>dotcol</i>	<i>dotrow</i>	<i>spr</i>	<i>tol</i>	instr
<b>DECIMAL</b>	<b>63</b>	<b>29</b>	<b>8</b>	<b>3</b>	<b>COINCXY</b>

(G M B)

will detect a coincidence between sprite #8 and the point ( 63,29 ) with a tolerance of 3. A true or false flag will again be left on the stack.

Both of the above instructions will detect a coincidence between non-visible parts of the sprites. That is, you may not be able to *see* the coincidence.

**COINCALL** ( *--- flag* )

Another instruction is used to detect only *visible* coincidences. It, however, will not detect coincidences between a select two sprites, but will return a true flag when any two sprites collide. This instruction is **COINCALL** , and requires no arguments.

### 6.6.7 Deleting Sprites

As you might have noticed, sprites do not go away when you clear the rest of the screen with CLS. Special instructions must be used to remove sprites from the display,

**DELSPR** ( *spr ---* )

<i>spr</i>	instr
<b>2</b>	<b>DELSPR</b>

(G M B)

will remove sprite #2 from the screen by altering its description in the sprite Attribute List (see VDP Memory Map in Chapter 4). It does not remove the velocity of sprite #2 from the Sprite Motion Table, nor does it alter the number of sprites the computer thinks it is dealing with. In other words, if you were to redefine sprite #2, it would immediately begin moving with whatever speed the old sprite #2 had.

**DELALL** ( *---* )

**DELALL**

(G M B)

on the other hand, will remove all sprites from the screen, and from memory. **DELALL** needs no parameters. Only the Sprite Descriptor Table will remain intact after this instruction is executed.

## 6.7 Multicolor Graphics

Multicolor mode allows you to display kaleidoscopic graphics. Each character position on the screen consists of 4 smaller squares which can each be a different color. A cluster of these characters produces a kaleidoscope when the colors are changed rapidly.

**MINIT** ( --- )

After entering multicolor mode, it is necessary to initialize the screen. The **MINIT** instruction will accomplish this. It needs no parameters.

When in multicolor mode, the columns are numbered 0 – 63 and rows are numbered 0 – 47. A multicolor character is  $\frac{1}{4}$  the size of a standard character; therefore more of them fit across and down the screen.

**MCHAR** ( *color col row ---* )

To define a multicolor character, you must specify a color and a position (column, row ) and then execute the word **MCHAR** :

base	<i>color</i>	<i>col</i>	<i>row</i>	instr	
<b>HEX</b>	<b>B</b>	<b>1A</b>	<b>2C</b>	<b>MCHAR</b>	or
<b>DECIMAL</b>	<b>11</b>	<b>26</b>	<b>44</b>	<b>MCHAR</b>	

The above instruction will place a light yellow square at (26,44).

To change a character's color, simply define a different color **MCHAR** with the same position. In other words, cover the existing character.

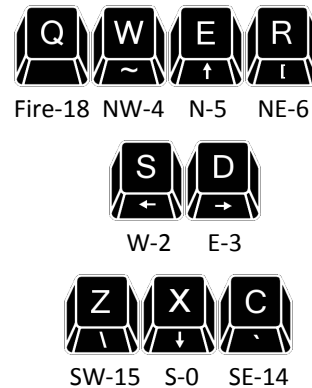
## 6.8 Using Joysticks

**JOYST** (  $n_1$  --- *char*  $n_2$   $n_3$  )

The **JOYST** instruction allows you to use joysticks in your Forth program. **JOYST** requires only one parameter, *viz.*, a keyboard number  $n_1$ . The keyboard number tells the computer which joystick or which side of the keyboard to scan for input. When keyboard #1 is specified ( $n_1 = 1$ ), both joystick #1 and the left side of the keyboard are scanned. When keyboard #2 is specified ( $n_1 = 2$ ), joystick #2 and the right side of the keyboard are scanned. A “Key Pad” exists on each side of the keyboard and may be used in place of joysticks. Map directions (N, S, E, W, NE, etc.) are used on the diagrams below to indicate the corresponding display-screen directions (up, down, right, left, diagonally-up-and-right, *etc.*) The following diagrams show which keys have which function.

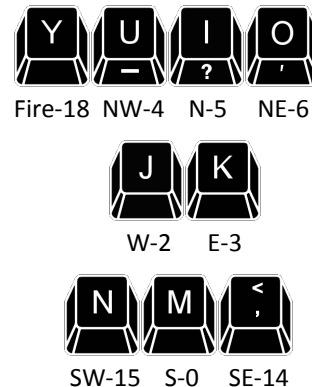
When Joystick #1 is specified, these keys on the left side of the keyboard are valid ■■■→

The function of each key is indicated below the key and is followed by the character code returned as *char* on the stack.



When Joystick #2 is specified, these keys on the right side of the keyboard are valid ■■■→

The function of each key is indicated below the key and is followed by the character code returned as *char* on the stack.



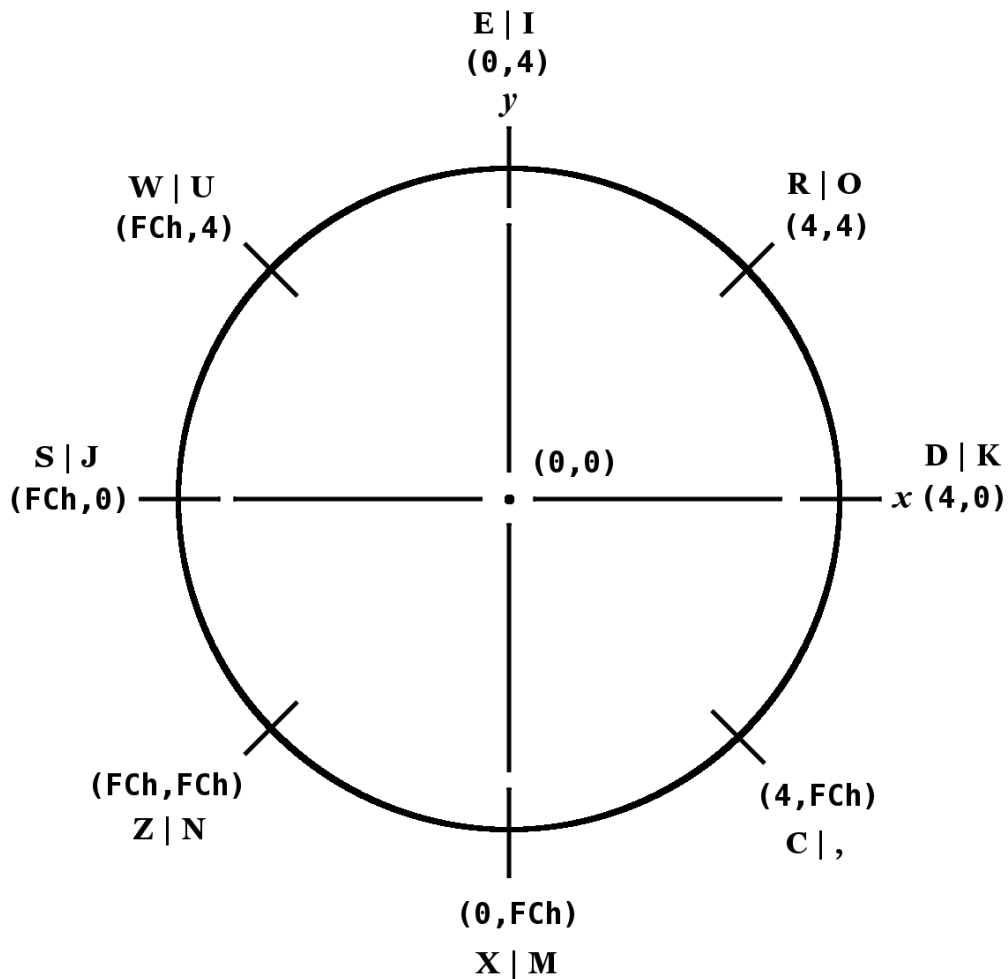
The **JOYST** instruction returns 3 numbers on the stack: a character code *char* on the bottom of the stack, an *x*-joystick status  $n_2$  and a *y*-joystick status  $n_3$  on top of the stack. The joystick positions are illustrated in the diagram that follows.

**FCh** equals decimal 252. The capital letters and ‘,’ separated by ‘|’ indicate which keys on the left and right side of the keyboard return these values. *Note:* The character value of all fire buttons is 18 (**12h**).

If no key is pressed, the returned values will be a character code of 255 (**FFh**), and the current *x*- and *y*-joystick positions. If a valid key is pressed, the character code of that key will be returned along with its translated directional meaning (see diagram).

If an illegal key is pressed, three zeroes will be returned. If the fire button is pressed, a character code of 18 (**12h**) along with two zeroes will be returned.

If you are using **JOYST** in a loop, do not forget to **DROP** or otherwise use the three numbers left on the stack before calling **JOYST** again. A stack overflow will likely result if you do not.



*Joystick positions and values*

## 6.9 Dot Graphics

High resolution (dot) graphics are available in graphics2, split and split2 modes. In graphics2 mode, it is possible to independently define each of the 49152 pixels on the screen. Split and split2 modes allow you to define the upper two thirds or the lower five sixths of the pixels.

Three dot drawing modes are available:

**DRAW** ( --- )

plots dots in the 'on' state.

**UNDRAW** ( --- )

plots dots in the 'off' state.

**DTOG** ( --- )

toggles dots between the 'on' and 'off' state. If the dot is 'on', **DTOG** will turn it 'off' and vice versa.

**DMODE** ( --- *addr* )

The value of a variable called **DMODE** controls which drawing mode you are in. If **DMODE** = 0, you are in **DRAW** mode. If **DMODE** = 1, you are in **UNDRAW** mode, and if **DMODE** = 2, you are in **DTOG** mode.

**DOT** ( *dotcol dotrow* --- )

To actually plot a dot on the screen, the **DOT** instruction is used. You must specify the dot column and dot row of the pixel you wish to plot:

base	<i>dotcol</i>	<i>dotrow</i>	instr
<b>DECIMAL</b>	<b>34</b>	<b>12</b>	<b>DOT</b>

will plot or unplot, depending on the value of **DMODE** , a dot at position ( 34,12 ).

**DCOLOR** ( --- *addr* )

The default color for dots is white on transparent. The screen color default is black. To alter the foreground and background color of the dots you plot, you must modify the value of the variable **DCOLOR** . The value of **DCOLOR** should be two hexadecimal digits where the first digit specifies the foreground color and the second specifies a background color. Why do you need a background color for a dot? There is a simple explanation. Each dot represents one bit of a byte in memory. Any bit in the byte that is turned 'on' displays the foreground color while the others take on the background color. Usually, you would specify the background color to be transparent.

**LINE** ( *dotcol<sub>1</sub> dotrow<sub>1</sub> dotcol<sub>2</sub> dotrow<sub>2</sub>* --- )

The Forth instruction **LINE** allows you to easily plot a line between *any* two points on the bit-map portion of the screen. You must specify a dot column and a dot row for each of the two points.

base	<i>dotcol<sub>1</sub></i>	<i>dotrow<sub>1</sub></i>	<i>dotcol<sub>2</sub></i>	<i>dotrow<sub>2</sub></i>	instr
<b>DECIMAL</b>	<b>23</b>	<b>12</b>	<b>56</b>	<b>78</b>	<b>LINE</b>

The above instruction will plot a line from left to right between ( 23,12 ) and ( 56,78 ). The line instruction calls **DOT** to plot each point therefore, you must preset **DMODE** and **DCOLOR** before using **LINE** .

## 6.10 Special Sounds

Two special sounds can be used to enhance your graphics application. To use these noises in your program, simply type the name of the sound you want to hear. No parameters are needed.

**BEEP** ( --- )

The first is called **BEEP** and produces a pleasant high pitched sound.

**HONK** ( --- )

The other, called **HONK** , produces a less pleasant low tone.

### 6.11 *Constants and Variables Used in Graphics Programming*

The following constants and variables are defined in the graphics routines. The value of **COLTAB** , **PDT** , **SATR** , **SMTN** and **SPDTAB** must be changed if you are operating in graphics2, split or split2 mode. See the VDP Memory Map in Chapter 4.

name	type	description	default
<b>COLTAB</b>	constant	VDP address of Color Table	<b>380h</b>
<b>DMODE</b>	variable	Dot graphics drawing mode	<b>0</b>
<b>PDT</b>	constant	VDP address of Pattern Descriptor Table	<b>800h</b>
<b>SATR</b>	constant	VDP address of Sprite Attribute Table	<b>300h</b>
<b>SMTN</b>	constant	VDP address of Sprite Motion Table	<b>780h</b>
<b>SPDTAB</b>	constant	VDP address of Sprite Descriptor Table	<b>800h</b>

## 7 The Floating Point Support Package

Words introduced in this chapter:

<b>&gt;ARG</b>	<b>F0&lt;</b>	<b>FUML</b>
<b>&gt;F</b>	<b>F0=</b>	<b>FOVER</b>
<b>&gt;FAC</b>	<b>F&lt;</b>	<b>FSUB</b>
<b>?FLERR</b>	<b>F=</b>	<b>FSWAP</b>
<b>ATN</b>	<b>F&gt;</b>	<b>INT</b>
<b>COS</b>	<b>F@</b>	<b>LOG</b>
<b>EXP</b>	<b>FAC-&gt;S</b>	<b>PI</b>
<b>F!</b>	<b>FAC&gt;</b>	<b>S-&gt;F</b>
<b>F*</b>	<b>FAC&gt;ARG</b>	<b>S-&gt;FAC</b>
<b>F+</b>	<b>FADD</b>	<b>SETFL</b>
<b>F-</b>	<b>FDIV</b>	<b>SIN</b>
<b>F-&gt;S</b>	<b>FDUP</b>	<b>SQR</b>
<b>F.</b>	<b>FF.</b>	<b>TAN</b>
<b>F.R</b>	<b>FF.R</b>	<b>VAL</b>
<b>F/</b>	<b>FLERR</b>	

The floating point package is designed to make it easy to use the Radix 100 floating point package available in ROM in the TI-99/4A console. Normal use of these routines does not require the user to understand the implementation. For those users desiring to improve the efficiency of these operations by optimizing the code for this implementation, the details are given in the latter portion of this chapter.

### 7.1 Floating Point Stack Manipulation

The floating point numbers in the TI-99/4A occupy 4 16-bit words<sup>11</sup> (8 bytes) each. In order to simplify stack manipulations with these numbers, the following stack manipulation words are presented:

<b>FDUP</b>	$(f \text{---} f \ f)$
<b>FDROP</b>	$(f \text{---} )$
<b>FOVER</b>	$(f_1 \ f_2 \text{---} f_1 \ f_2 \ f_1)$
<b>FSWAP</b>	$(f_1 \ f_2 \text{---} f_2 \ f_1)$

<sup>11</sup> This use of the term “word” here is different from a Forth word. It refers to the largest memory unit the TMS9900 CPU can address. It is equal to 2 bytes or 16 bits.

## 7.2 Floating Point Fetch and Store

Floating point numbers can be stored and fetched by using

**F!** (*f addr ---*)

**F@** (*addr --- f*)

The user must ensure that adequate storage is allocated for these numbers (*e.g.*, **0 VARIABLE nnnn 6 ALL0T** could be used. **VARIABLE** allots 2 bytes.)

## 7.3 Floating Point Conversion Words

The following words put floating point numbers on the stack so that the above operations can be used:

**S->F** (*n --- f*)

A 16-bit number can be converted to floating point by using **S->F**. It functions by replacing the 16-bit number on the stack by a floating point number of equal value.

**F->S** (*f --- n*)

This is the inverse of **S->F**. It starts with a floating point number on the stack and leaves a 16-bit integer.

## 7.4 Floating Point Number Entry

In addition, the word

**>F** (*--- f*)

can be used from the console or in a colon definition to convert a string of characters to a floating point number. Note that **>F** is independent of the current value of **BASE**.

The string is always terminated by a blank or carriage return. The following are examples:

```
>F 123          or 123 S->F
>F 123.46
>F -123.46
>F 1.23E-006
>F 9.88E+091
>F 0            or 0 S->F
```

## 7.5 Floating Point Arithmetic

Floating point arithmetic can now be performed on the stack just as it is with integers. The four arithmetic operators are:

**F+** (*f<sub>1</sub> f<sub>2</sub> --- f<sub>3</sub>*)



<b>F-</b>	$(f_1 f_2 \text{ --- } f_3)$	Puts on the stack the result $(f_3)$ of $f_1 - f_2$ .
<b>F*</b>	$(f_1 f_2 \text{ --- } f_3)$	Puts on the stack the result $(f_3)$ of $f_1 \times f_2$ .
<b>F/</b>	$(f_1 f_2 \text{ --- } f_3)$	Puts on the stack the result $(f_3)$ of $f_1 / f_2$ .
<b>PI</b>	$(\text{--- } f)$	

The word **PI** is a constant available to place 3.141592653590 on the stack.

## 7.6 Floating Point Comparison Words

Comparisons between floating point numbers and testing against zero are provided by the following words. They are used just like their 16-bit counterparts except that the numbers tested are floating point.

<b>F0&lt;</b>	$(f \text{ --- } flag)$	<i>flag</i> is true if <i>f</i> on stack is negative
<b>F0=</b>	$(f \text{ --- } flag)$	<i>flag</i> is true if <i>f</i> on stack is zero
<b>F&gt;</b>	$(f_1 f_2 \text{ --- } flag)$	<i>flag</i> is true if $f_1 > f_2$
<b>F=</b>	$(f_1 f_2 \text{ --- } flag)$	<i>flag</i> is true if $f_1 = f_2$
<b>F&lt;</b>	$(f_1 f_2 \text{ --- } flag)$	<i>flag</i> is true if $f_1 < f_2$

## 7.7 Formatting and Printing Floating Point Numbers

**F.**  $(f \text{ --- })$

The word **F.** is used to print the floating point number on the top of the stack to the terminal. The format used is identical to that used by BASIC:

- 1) Integers representable exactly are printed without a trailing decimal,
- 2) Fixed point format is used for numbers in range and
- 3) Exponential format (scientific notation) is used for very large or very small numbers.

**F.R**  $(f n \text{ --- })$

If the floating point numbers are to be output in a table the word **F.R** can be used to right justify it in a field of width *n* where *n* is a 16-bit word added to the top of the stack for this purpose.

Two additional words are used for more specific formatting:

**FF.**  $(f n_1 n_2 \text{ --- })$

**FF.** requires two integers on the stack above the floating point number *f*. They control the maximum number of digits (*n*<sub>1</sub>) to convert and the number of digits (*n*<sub>2</sub>) following the decimal point.

**FF.R**  $(f n_1 n_2 n_3 \text{ --- })$

**FF.R** adds the printing field width (*n*<sub>3</sub>), in which the output is right justified. As for **FF.**, *n*<sub>1</sub> is the maximum number of digits to convert and *n*<sub>2</sub> is the number of digits following the decimal point.

## 7.8 Transcendental Functions

The following transcendental functions are also available:

<b>INT</b>	$(f_1 \text{ --- } f_2)$	Returns largest integer not larger than input
<b>^</b>	$(f_1 \ f_2 \text{ --- } f_3)$	$f_3$ is $f_1$ raised to the $f_2$ power
<b>SQR</b>	$(f_1 \text{ --- } f_2)$	$f_2$ is the square root of $f_1$
<b>EXP</b>	$(f_1 \text{ --- } f_2)$	$f_2$ is e (2.71828...) raised to the $f_1$ power
<b>LOG</b>	$(f_1 \text{ --- } f_2)$	$f_2$ is the natural log of $f_1$
<b>COS</b>	$(f_1 \text{ --- } f_2)$	$f_2$ is the cosine of $f_1$ (in radians)
<b>SIN</b>	$(f_1 \text{ --- } f_2)$	$f_2$ is the sin of $f_1$ (in radians)
<b>TAN</b>	$(f_1 \text{ --- } f_2)$	$f_2$ is the tangent of $f_1$ (in radians)
<b>ATN</b>	$(f_1 \text{ --- } f_2)$	$f_2$ is the arctangent (in radians) of $f_1$

*Caution!* A conflict exists when using transcendentals and floating point prints while in bit-map mode. The contents of the VDP Rollout Area (**3C0h – 3DFh**) must be saved before transcendentals or floating point prints are executed and restored upon completion.

*Note:* The transcendentals also use the area known as the stack for the value stack pointer, VSPTR (See VDP Memory Map in Chapter 4). This area is pointed to by **836Eh** (VSPTR).

## 7.9 Interface to the Floating Point Routines

The remainder of this chapter will address the interface to the floating point routines in the console in greater detail and is not necessary for most floating point operations.

The floating point routines use two memory locations in the console CPU RAM as floating point registers. They are called FAC (for floating point accumulator) and ARG (for argument register). Forth has two constants with these same names that can be used to access these locations directly:

<b>FAC</b>	$( \text{--- } addr )$	constant that puts the address of FAC on the stack.
<b>ARG</b>	$( \text{--- } addr )$	constant that puts the address of ARG on the stack.

The words **>FAC** and **>ARG** move floating point data from the stack to these locations.

<b>&gt;FAC</b>	$( f \text{ --- } )$	moves $f$ to FAC.
<b>&gt;ARG</b>	$( f \text{ --- } )$	moves $f$ to ARG.
<b>FAC&gt;</b>	$( \text{--- } f )$	is used to move data from FAC to the stack.
<b>SETFL</b>	$( f_1 \ f_2 \text{ --- } )$	

Each of the binary floating point operations requires that two numbers be moved from the stack to FAC and ARG. **SETFL** does this by calling **>FAC** and **>ARG** to place  $f_2$  in FAC and  $f_1$  in ARG.

The words **FADD**, **FSUB**, **FMUL** and **FDIV** each use the values in FAC and ARG and leave the result in FAC as they perform the floating point arithmetic functions.

**FADD** ( --- )

**FSUB** ( --- )

**FMUL** ( --- )

**FDIV** ( --- )

When conversion from 16-bit integer to floating point is performed by **S->F**, it is done in the FAC. If the user does not desire the result to be copied from FAC to the stack, the word **S->FAC** can be used instead:

**S->FAC** ( *n* --- )

**S->FAC** moves a 16-bit integer (*n*) to the FAC, where it converts it to a floating point number.

Several miscellaneous words include:

**FAC->S** ( --- *n* ) converts the contents of FAC to a 16-bit integer on the stack.

**FAC>ARG** ( --- ) copies the contents of FAC to ARG.

**VAL** ( --- )

**VAL** converts a string at PAD to a floating point number in FAC. **VAL** expects the first byte at PAD to be the character count. There must not be any leading spaces in the string.

**FLERR** ( --- *n* )

**FLERR** is used to fetch the contents of the floating point error register (**8354h**) to the stack. See the *Editor/Assembler Manual* for more information.

**?FLERR** ( --- )

**?FLERR** issues an appropriate error message if the last floating point operation resulted in an error.

## 8 Access to File I/O Using TI-99/4A Device Service Routines

Words introduced in this chapter:

APPND	I/OMD	REC-NO
CHAR-CNT!	INPT	RLTV
CHAR-CNT@	INTRNL	RSTR
CHK-STAT	LD	SCRTCH <sup>12</sup>
CLR-STAT	N-LEN!	SET-PAB
CLSE	OPN	SQNTL
DLT	OUTPT	STAT
DOI/O	PAB-ADDR	SV
DSPLY	PAB-BUF	SWCH
F-D"	PAB-VBUF	UNSWCH
FILE	PUT-FLAG	UPDT
FXD	RD	VRBL
GET-FLAG	REC-LEN	WRT

This chapter will explain the means by which different types of data files native to the TI-99/4A are accessed with TI Forth. To further illustrate the material, two commented examples have been included in this chapter. The first (§ 8.6 ) demonstrates the use of a relative disk file and the second (§ 8.7 ) a sequential RS232 file.

A group of Forth words has been included in this version of TI Forth to permit a Forth program to reference common data with BASIC or Assembly Language programs. These words implement the file system described in the *TI BASIC Manual* and the *Editor/Assembler Manual*. Note that the diskette on which you received your TI Forth system is **not** a standard diskette and that you should perform file I/O to/from disks only if they are initialized by the Disk Manager and do **not** contain Forth screens.

### 8.1 The Peripheral Access Block (PAB)

Before any file access can be achieved, a Peripheral Access Block (PAB) must be set up that describes the device and file to be accessed. Most of the words in this chapter are designed to make manipulation of the PAB as easy as possible.

A PAB consists of 10 bytes of VDP RAM plus as many bytes as the device name to be accessed. An area of VDP RAM has been reserved for this purpose (consult the VDP Memory Map in Chapter 4). The user variable PABS points to the beginning of this region. **Do not** use the first 2 bytes of this area as they are used by Forth in its Forth-style disk access. Adequate space is provided for many PABs in this area. More information on the details of a PAB are available in

<sup>12</sup> **SCRTCH** , though defined in TI Forth, was never implemented in any DSR for the TI-99/4A. Its use will result in a file I/O error.

the *Editor/Assembler Manual*, page 293ff. The following diagram illustrates the structure of a PAB:

Byte 0 I/O Opcode	Byte 1 Flag/Status
Bytes 2 & 3 Data Buffer Address in VDP	
Byte 4 Logical Record Length	Byte 5 Character Count
Bytes 6 & 7 Record Number	
Byte 8 Screen Offset	Byte 9 Name Length
Byte 10+ File Descriptor • • •	

## 8.2 File Setup and I/O Variables

All Device Service Routines (DSRs) on the TI-99/4A expect to perform data transfers to/from the VDP RAM. Since Forth is using CPU RAM it means that the data will be moved twice in the process of reading or writing a file. Three variables are defined in the file I/O words to keep track of these memory areas.

**PAB-ADDR** ( --- *vaddr* )

Points into VDP RAM to first byte of the PAB.

**PAB-BUF** ( --- *addr* )

Points into CPU RAM to first byte in Forth's memory where allocation has been made for this buffer.

**PAB-VBUF** ( --- *vaddr* )

Points into VDP RAM to the first byte of a region of adequate length to store data temporally while it is transferred between the file and Forth. The area of VDP RAM which is used for this purpose is labeled "Unused" on the VDP Memory Map in Chapter 4. If working in bit-map mode, be cautious where **PAB-VBUF** is placed.

**FILE** ( *vaddr*<sub>1</sub> *addr* *vaddr*<sub>2</sub> --- )

The word **FILE** is a defining word and permits you to create a word which is the name by which the file will be known. A decision must be made as to the location of each of the buffers before the word **FILE** may be used. The values to be used for those locations are

contained in the above variables and are placed on the stack in the above order followed by **FILE** and the file name (not necessarily the device name). For example:

#### Using The Defining Word, **FILE**

<b>0 VARIABLE MY-BUF 78 ALLOT</b>	(Create 80 character buffer)
<b>PABS @ 10 +</b>	(PAB starts 10 bytes into region for PABS: <b>PAB-ADDR</b> )
<b>MY-BUF</b>	(Location of <b>PAB-BUF</b> )
<b>6000</b>	(A free area for <b>PAB-VBUF</b> )
<b>FILE JOE</b>	(Whenever the word <b>JOE</b> is executed, the file I/O variables will be set as defined here.)
<b>JOE</b>	(Use the word before using any other file I/O words)

#### **SET-PAB** ( --- )

The word that creates the PAB skeleton is **SET-PAB** . It creates a PAB at the address shown in **PAB-ADDR** and zeroes it except for the buffer address slot. Into this it places the contents of the variable **PAB-VBUF** .

### 8.3 File Attribute Words

Files on the TI-99/4A have various characteristics that are indicated by keywords. The following table describes the available options. The example in the back of the chapter will be helpful in that it shows at what time in the procedure these words are used. Use only the attributes which apply to your file and ignore the others. Remember, if you are using multiple files, then the file referenced is the file whose name word was most recently executed.

Attribute Type	Options From		Description
	BASIC	Forth	
File Type	SEQUENTIAL	<b>SQNTL*</b>	Records may only be accessed in sequential order
	RELATIVE	<b>RLTV</b>	Accessed in sequential or random order. Records must be of fixed length
Record Type	FIXED	<b>FXD*</b>	All records in the file are the same length
	VARIABLE	<b>VRBL</b>	Records in the same file may have different lengths
Data Type	DISPLAY	<b>DSPLY*</b>	File contains printable or displayable characters
	INTERNAL	<b>INTRNL</b>	File contains data in machine or binary format

Attribute Type	Options From		Description
	BASIC	Forth	
Mode of Operation	INPUT	<b>INPT</b>	File contents can be read from but not written to
	OUTPUT	<b>OUTPT</b>	File contents can be written to but not read from
	UPDATE	<b>UPDT*</b>	File contents can be written to and read from
	APPEND	<b>APPND</b>	Data may be added to end of file but cannot be read

\* Default if attribute is not specified

#### **REC-LEN** ( *b* --- )

To specify the record length for a file, the desired length byte *b* should be on the stack when the word **REC-LEN** is executed. The length will be placed in the current PAB.

#### **F-D"** ( --- )

Every file must have a name to specify the device and file to be accessed. This is performed with the **F-D"** word which enters the File Description in the PAB. **F-D"** must be followed by a string describing the file and terminated by a " mark. Here are a few examples of the use of **F-D"** :

**F-D" RS232.BA=9600"**

**F-D" DSK2.FILE-ABC"**

### **8.4 Words that Perform File I/O**

The actual I/O operations are performed by the following words. The table gives the usual BASIC keyword associated with the corresponding Forth word. Here, as in the previous table, the Forth words are spelled differently than the BASIC words to avoid conflict with one or more existing Forth words.

From BASIC	From Forth	DSR Opcode
OPEN	<b>OPN</b>	0
CLOSE	<b>CLSE</b>	1
READ	<b>RD</b>	2
WRITE	<b>WRT</b>	3
RESTORE	<b>RSTR</b>	4
LOAD	<b>LD</b>	5
SAVE	<b>SV</b>	6

From BASIC	From Forth	DSR Opcode
DELETE	<b>DLT</b>	7
SCRATCH	<b>SCRTCH</b> <sup>13</sup>	8
STATUS	<b>STAT</b>	9

**OPN** ( --- )

opens the file specified by the currently selected PAB, which is pointed to by **PAB-ADDR** .

**CLSE** ( --- )

closes the file whose PAB is pointed to by **PAB-ADDR** .

**REC-NO** ( *n* --- )

Before using the **RD** , **WRT** and **SCRTCH**<sup>14</sup> instructions with a relative file, you must place the desired record number *n* into the PAB. To do this, place the record number *n* on the stack and execute the word **REC-NO** . If your file is Sequential, you need not do this.

**RD** ( --- *n* )

The **RD** instruction will transfer the contents of the next record from the current file into your **PAB-BUF** and leave a character count *n* on the stack.

**WRT** ( *n* --- )

takes a character count *n* from the stack and moves that number of characters from the **PAB-BUF** to the current file.

**RSTR** ( *n* --- )

takes a record number *n* from the stack and repositions (restores) a relative file to that record for the next access.

**SCRTCH**<sup>15</sup> ( *n* --- )

is used to remove a relative record. It requires a record number *n* on the stack.

**LD** ( *n* --- )

used to load a program file of maximum *n* bytes into VDP RAM at the address specified in **PAB-VBUF** . **OPN** and **CLSE** need not be used.

**SV** ( *n* --- )

used to save *n* bytes of a program file from VDP RAM at the address specified in **PAB-VBUF** . **OPN** and **CLSE** need not be used.

**DLT** ( --- )

is used to delete the file whose PAB is pointed to by **PAB-ADDR** .

---

<sup>13</sup> See footnote 12, page 52.

<sup>14</sup> *idem*

<sup>15</sup> *idem*



**STAT** ( --- *b* )

returns the status byte *b* (labeled “screen offset”) of the current device/file from the PAB pointed to by **PAB-ADDR**. The table below, excerpted from the *Editor/Assembler Manual*, p. 298, shows the meaning of each bit of the status byte:

Bit	Status Byte Information When Value is	
	1	0
0	File does not exist.	File exists. If device is a printer or similar, always 0.
1	Protected file.	Unprotected file.
2		Reserved for future use. Always 0.
3	INTERNAL data type.	DISPLAY data type or program file.
4	Program file.	Data file.
5	VARIABLE record length.	FIXED record length.
6	At physical end of peripheral. No more data can be written.	Not at physical end of peripheral. Always 0 when file not open.
7	End of file (EOF). Can be written if open in APPEND, OUTPUT or UPDATE modes. Reading will cause an error.	Not EOF. Always 0 when file not open.

The words that follow are available for the advanced user and their utility can be worked out by examining their definitions on Forth screen 68ff. They are lower-level words that are used in the definitions of the above file I/O words.

**GET-FLAG** ( --- *b* )

retrieves to the stack the flag/status byte *b* from the current PAB. See table below for the meaning of each bit:

**Flag/Status Byte of PAB (Byte 1)**

Bits	Contents	Meaning
0–2	Error Code	0 = no error. Error codes are decoded in table below.
3	Record Type	0 = fixed-length records; 1 = variable-length records.
4	Data Type	0 = DISPLAY; 1 = INTERNAL.
5–6	Mode of Operation	0 = UPDATE; 1 = OUTPUT; 2 = INPUT; 3 = APPEND.
7	File Type	0 = sequential file; 1 = relative file.

### Error Codes in Bits 0–2 of Flag/Status Byte of PAB

Error Code	Meaning
0	No error unless bit 2 of status byte at address <b>837Ch</b> is set ( then, bad device name).
1	Device is write protected.
2	Bad OPEN attribute such as incorrect file type, incorrect record length, incorrect I/O mode or no records in a relative record file.
3	Illegal operation; <i>i.e.</i> , an operation not supported on the peripheral or a conflict with the OPEN attributes.
4	Out of table or buffer space on the device.
5	Attempt to read past the end of file. When this error occurs, the file is closed. Also given for non-extant records in a relative record file.
6	Device error. Covers all hard device errors such as parity and bad medium errors.
7	File error such as program/data file mismatch, non-existing file opened in INPUT mode, <i>etc.</i>

**PUT-FLAG** ( *b* --- )

writes the flag/status byte *b* on the stack to the current PAB. See table after **GET-FLAG** for the meaning of each bit.

**CLR-STAT** ( --- )

clears the error code in bits 0–2 of the flag/status byte of the current PAB.

**CHK-STAT** ( --- )

checks the error code in bits 0–2 of the flag/status byte of the current PAB. If it is not 0, an appropriate error message is printed.

**I/OMD** ( --- *b* )

gets the flag/status byte *b* of the current PAB, clears the I/O mode bits (5 & 6) and leaves it on the stack in preparation for setting the I/O mode with an I/O word.

**CHAR-CNT!** ( *n* --- )

stores the character count *n* in the current PAB prior to a write operation. **CHAR-CNT!** is used by **WRT**.

**CHAR-CNT@** ( --- *n* )

retrieves the character count *n* from the current PAB of the last read operation. It is used by **RD**.

**N-LEN!** ( *b* --- )

stores in the current PAB the length byte *b* of the file descriptor associated with the current PAB. For “DSK1.MYFILE”, this would be 11.

**DOI/O** ( *n* --- )

executes the **DSRLNK** word with the I/O opcode *n* on the stack. The current PAB must be updated with the information required by opcode *n* before executing **DOI/O**. See Section 18.2.1 of the *Editor/Assembler Manual* for details or consult the definitions on Forth screen 68ff. of the I/O words, **OPN**, **CLSE**, **RD**, **WRT**, **RSTR**, **SCRATCH**<sup>16</sup>, **LD**, **SV**, **DLT** and **STAT**, all of which use this low-level word in their definitions.

Examples of file I/O in use are available on Forth screen 72ff., which defines the Alternate I/O capabilities for printing to the RS232 interface.

## 8.5 Alternate Input and Output

When using alternate input or output devices, the 1-byte buffer in VDP memory must be the byte immediately preceding the PAB for **ALTIN** or **ALTOUT**.

The words

**SWCH** ( --- ) and

**UNSWCH** ( --- )

make it possible to send output that would normally go to the monitor to an RS232 printer. For example, the **LIST** instruction normally outputs to the monitor. By typing

**SWCH 45 LIST UNSWCH**

you can list Forth screen 45 to the printer. If your RS232 printer is not on port 1 and set at 9600 baud, you must modify the word **SWCH** on your system disk.

The user variables

**ALTIN** ( --- *vaddr* ) and

**ALTOUT** ( --- *vaddr* )

contain values which point to the current input and output devices. The value of **ALTIN** is 0 if input is coming from the keyboard, else its value is a pointer to the VDP address where the PAB for the alternate input device is located. The value of **ALTOUT** is 0 if the output is going to the monitor. Otherwise, it contains a pointer to the PAB of the alternate output device.

## 8.6 File I/O Example 1: Relative Disk File

Instruction	Comment
<b>HEX</b>	Change number base to Hexadecimal
<b>0 VARIABLE BUFR 3E ALLOT</b>	Create space for a 64 byte buffer which will be the <b>PAB-BUF</b>
<b>PABS @ A +</b>	PAB starts 10 bytes into <b>PABS</b> . This will be the <b>PAB-ADDR</b>
<b>BUFR 1700</b>	Place the <b>PAB-BUF</b> and <b>PAB-VBUF</b> on stack in preparation

<sup>16</sup> See footnote 12, page 52.

Instruction	Comment
	for <b>FILE</b>
<b>FILE TESTFIL</b>	Associates the name <b>TESTFIL</b> with these three parameters
<b>TESTFIL</b>	File name must be executed before using any other File I/O words
<b>SET-PAB</b>	Create PAB skeleton
<b>RLTV</b>	Make <b>TESTFIL</b> a relative file
<b>DSPLY</b>	Records will contain printable information
<b>40 REC-LEN</b>	Record length is 64 ( <b>40h</b> ) bytes
<b>F-D" DSK2.TEST"</b>	Will create the file descriptor "DSK2.TEST" in the PAB for <b>TESTFIL</b> .
<b>OPN</b>	Open the file. This will create the file on disk unless it already exists.
. . . . .	
To write more than one record to the file, it is necessary to write a procedure. This routine may be composed on a Forth screen beforehand and loaded at this time.	
<b>: FIL-WRT TESTDATA</b>	<b>TESTDATA</b> is assumed to be the beginning memory address of the information to be written to the file
<b>10 0 DO</b>	Want to write 16 ( <b>10h</b> ) records
<b>DUP</b>	Duplicate address
<b>BUFR 40 CMOVE</b>	Move 64 bytes of information into the <b>PAB-BUF</b>
<b>I REC-NO</b>	Place record number into PAB
<b>40 WRT</b>	Write one 64-byte record to the disk
<b>40 +</b>	Increment address for next record
<b>LOOP DROP</b>	Clear stack
<b>;</b>	End definition
. . . . .	
<b>FIL-WRT</b>	Execute writing procedure
<b>4 REC-NO RD</b>	Choose a record number to read (4 is chosen here) to verify correct output. A byte count will be left on the stack and the read information will be in <b>BUFR</b>
<b>BUFR 40 DUMP</b>	Print out the read information to the monitor. ( <b>DUMP</b> routines must be loaded)
<b>CLSE</b>	Close the file

**8.7 File I/O Example 2: Sequential RS232 File**

Instruction	Comment
HEX	Change number base to Hexadecimal
0 VARIABLE MY-BUF 4E ALL0T	Create a 80 character <b>PAB-BUF</b>
PABS @ 30 +	Skip all previous PAB. This will be the <b>PAB-ADDR</b>
MY-BUF 1900	Place the <b>PAB-BUF</b> and <b>PAB-VBUF</b> on stack in preparation for <b>FILE</b>
FILE PRNTR	Associates the name <b>PRNTR</b> with these three parameters
PRNTR	File name must be executed before using any other File I/O words
SET-PAB	Create a PAB skeleton
DSPLY	<b>PRNTR</b> will contain printable information
SQNTL	<b>PRNTR</b> may be accessed only in sequential order
VRBL	Records may have variable lengths
50 REC-LEN	Maximum record length is 80 char.
F-D" RS232.BA=9600"	<b>PRNTR</b> will be an RS232 file. Baud rate = 9600.
OPN	Open the file
. . . . .	
A procedure is necessary to write more than one record to a file. A file-write routine may be composed on a Forth screen beforehand and loaded at this time. The following is a simple example:	
: PRNT FILE-INFO	<b>FILE-INFO</b> is assumed to be the beginning memory address of the information to be sent to the printer
20 0 D0	Will write 32 records
DUP	Duplicate address
MYBUF 50 CMOVE	Move 80 characters from <b>FILE-INFO</b> to <b>MY-BUF</b>
50 WRT	Write one record to printer
50 +	Increment address on stack
LOOP DROP	Clear stack
;	End definition
. . . . .	
PRNT	Execute write program
CLSE	Close the file called <b>PRNTR</b>

## 9 The TI Forth 9900 Assembler

The assembler supplied with your TI Forth system is typical of assemblers supplied with fig-Forth systems. It provides the capability of using all of the opcodes of the TMS9900 as well as the ability to use structured assembly instructions. It uses no labels. The complete Forth language is available to the user to assist in macro type assembly, if desired. The assembler uses the standard Forth convention of Reverse Polish Notation for each instruction. For example the instruction to add register 1 to register 2 is:

```
1 2 A,
```

As can be seen in the above example, the ‘add’ instruction mnemonic is followed by a comma. Every opcode in this Forth assembler is followed by a comma. The significance is that when the opcode is reached during the assembly process, the instruction is compiled into the dictionary. The comma is a reminder of this compile operation. It also serves to assist in differentiating assembler words from the rest of the words in the TI Forth language. A complete list of Forth-style instruction mnemonics is given in the next section.


### 9.1 TMS9900 Assembly Mnemonics

A,	JEQ,	RSET,
AB,	JGT,	RTWP,
ABS,	JH,	S,
AI,	JHE,	SB,
ANDI,	JL,	SB0,
B,	JLE,	SBZ,
BL,	JLT,	SET0,
BLWP,	JMP,	SLA,
C,	JNC,	SOC,
CB,	JNE,	SOCB,
CI,	JNO,	SRA,
CKOF,	JOC,	SRC,
CKON,	JOP,	SRL,
CLR,	LDCR,	STCR,
COC,	LI,	STST,
CZC,	LIMI,	STWP,
DEC,	LREX,	SWPB,
DECT,	LWPI,	SZC,
DIV,	MOV,	SZCB,
IDLE,	MOVB,	TB,
INC,	MPY,	X,
INCT,	NEG,	XOP,
INV,	ORI,	XOR,

These words are available when the assembler is loaded. Only the word **C**, conflicts with the existing Forth vocabulary.

Most assembly code in Forth will probably use Forth's workspace registers. The following table describes the register allocation. The user may use registers 0 through 7 for any purpose. They are used as temporary registers only within Forth words which are themselves written in TMS9900 assembly code.

## 9.2 Forth's Workspace Registers

Register Name	Usage
0	 These registers are available. They are used only within Forth words written in <b>CODE</b> .
1	
2	
3	
4	
5	
6	
7	
UP	Points to base of User Variable area
SP	Parameter Stack Pointer
W	Inner Interpreter current Word pointer
11	Linkage for subroutines in <b>CODE</b> routines
12	Used for CRU instructions
IP	Interpretive Pointer
RP	Return Stack Pointer
NEXT	Points to the next instruction fetch routine

When the assembler is loaded, it is loaded into the **ASSEMBLER** vocabulary. To use the assembler, type **ASSEMBLER** to make it the context vocabulary. Assembly definitions begin with either the word **CODE** or **;CODE**. These are used in the following way:

### **ASSEMBLER CODE EXAMPLE**

This begins the definition of a code routine named **EXAMPLE**. The above words would be followed by assembly mnemonics as desired. **;CODE** is used as very much like the word **DOES>**:

### **ASSEMBLER : DEF-WRD**

- 
- an existing defining word must be included

. here to create the dictionary header.

**;CODE**

assembly mnemonics

Later when the newly created defining word **DEF-WRD** is executed in the following form, a new word is defined:

**DEF-WRD TEST**

This will create the word **TEST** which has as its execution procedure the code following **;CODE**.

We will now introduce those words that permit this assembler to perform the various addressing modes of which the TMS9900 is capable. Each of the remaining examples will show both the Forth assembler code for various instructions and the more conventional method of coding the same instructions.

The word **NEXT**, is defined as (see § 9.8 for definition of **\*NEXT**)

**: NEXT, \*NEXT B, ;**

and is equivalent to the following assembly code:

**B        \*R15**

### 9.3 Workspace Register Addressing

The registers in the Forth code below are referenced directly by number:

Forth	Conventional Assembler
<b>CODE EX1</b>	<b>DEF    EX1</b>
<b>1 2 A,</b>	<b>EX1    A    R1,R2</b>
<b>3 INC,</b>	<b>INC    R3</b>
<b>3 FFFC ANDI,</b>	<b>ANDI   R3,&gt;FFFC</b>
<b>NEXT,</b>	<b>B       *R15</b>

### 9.4 Symbolic Memory Addressing

Symbolic addressing is done with the **@()** word. It is used after the address.

Forth	Conventional Assembler
<b>0 VARIABLE VAR1</b>	<b>VAR1    BSS    2</b>
<b>5 VARIABLE VAR2</b>	<b>VAR2    DATA   5</b>
<b>CODE EX2</b>	<b>DEF    EX2</b>
<b>VAR2 @() 1 MOV,</b>	<b>EX2    MOV    @VAR2,R1</b>
<b>1 2 SRC,</b>	<b>SRC    R1,2</b>
<b>1 VAR1 @() S,</b>	<b>S       R1,@VAR1</b>
<b>VAR2 @() VAR1 @() SOC,</b>	<b>SOC    @VAR2,@VAR1</b>
<b>NEXT,</b>	<b>B       *R15</b>



### 9.5 Workspace Register Indirect Addressing

Workspace Register Indirect addressing is done with the **\*?** word. It is used after the register number to which it pertains.

Forth	Conventional Assembler		
2000 CONSTANT XRAM	XRAM	EQU	>2000
CODE EX3		DEF	EX3
1 XRAM LI,	EX3	LI	R1,XRAM
1 *? 2 MOV,		MOV	*R1,R2
NEXT,		B	*R15

### 9.6 Workspace Register Indirect Auto-increment Addressing

Workspace Register Indirect Auto-increment addressing is done with the **\*?+** word. It is also used after the register to which it pertains.

Forth	Conventional Assembler		
2000 CONSTANT XRAM	XRAM	EQU	>2000
CODE EX4		DEF	EX4
1 XRAM LI,	EX4	LI	R1,XRAM
1 *?+ 2 MOV,		MOV	*R1+,R2
NEXT,		B	*R15

### 9.7 Indexed Memory Addressing

The final addressing type is Indexed Memory addressing. This is performed with the **@(?)** word used after the Index and register as shown below:

Forth	Conventional Assembler		
2000 CONSTANT XRAM	XRAM	EQU	>2000
CODE EX5		DEF	EX5
XRAM 1 @(?) 2 MOV,	EX5	MOV	@XRAM(R1),R2
XRAM 22 + 2 @(?)		MOV	XRAM+22@ (2) ,XRAM+26@ (2)
XRAM 26 + 2 @(?) MOV,			
NEXT,		B	*R15

### 9.8 Addressing Mode Words for Special Registers

In order to make addressing modes easier for the **W**, **RP**, **IP**, **SP**, **UP** and **NEXT** registers, the following words are available and eliminate the need to enter the register name separately:

Register Address	Indirect	Indexed	Indirect Auto-increment
<b>W</b>	<b>*W</b>	<b>@(W)</b>	<b>*W+</b>
<b>RP</b>	<b>*RP</b>	<b>@(RP)</b>	<b>*RP+</b>
<b>IP</b>	<b>*IP</b>	<b>@(IP)</b>	<b>*IP+</b>
<b>SP</b>	<b>*SP</b>	<b>@(SP)</b>	<b>*SP+</b>
<b>UP</b>	<b>*UP</b>	<b>@(UP)</b>	<b>*UP+</b>
<b>NEXT</b>	<b>*NEXT</b>	<b>@(NEXT)</b>	<b>*NEXT+</b>

## 9.9 Handling the Forth Stacks

Both the parameter stack and the return stack grow downward in memory. This means that removing a cell from the top of either stack requires *incrementing* the stack pointer after consuming the cell's value. Conversely, adding a cell requires *decrementing* the stack pointer. The Forth Assembler word **\*SP+** references the contents of the top cell of the parameter stack and then increments the stack pointer **SP** to reduce the size of the stack by one cell. The following code copies the contents of the stack's top cell to register 0 and reduces the stack by one cell:

```
*SP+ 0 MOV,
```

The following code adds a cell to the top of the stack and copies the contents of register 1 to the new cell:

```
SP DECT,  
1 *SP MOV,
```

The same procedures obtain for the return stack using **\*RP+**, **RP** and **\*RP**; but, be very careful if you must manipulate the return stack.

## 9.10 Structured Assembler Constructs

This assembler also permits the user to write structured code, *i.e.*, code that does not use labels. This is done in a manner very similar to the way that Forth implements conditional constructs. The major difference is that rather than taking a value from the stack and using it as a true/false flag, the processor's condition register is used to determine whether or not to jump. The following structured constructs are implemented:

```
IF, ... ENDIF,  
IF, ... ELSE, ... ENDIF,  
BEGIN, ... UNTIL,  
BEGIN, ... AGAIN,  
BEGIN, ... WHILE, ... REPEAT,
```

The three conditional words in the previous list (**IF**, **UNTIL**, and **WHILE**, ) must each be preceded by one of the jump tokens in the next section.

### 9.11 Assembler Jump Tokens

Token	Comment	Conventional Assembler Used	Machine Code Generated
<b>EQ</b>	True if =	<b>JNE</b>	<b>1600h</b>
<b>GT</b>	True if signed >	<b>JGT \$+1 JMP</b>	<b>1501h 1000h</b>
<b>GTE</b>	True if signed > or =	<b>JLT</b>	<b>1100h</b>
<b>H</b>	True if unsigned >	<b>JLE</b>	<b>1200h</b>
<b>HE</b>	True if unsigned > or =	<b>JL</b>	<b>1A00h</b>
<b>L</b>	True if unsigned <	<b>JHE</b>	<b>1400h</b>
<b>LE</b>	True if unsigned < or =	<b>JH</b>	<b>1B00h</b>
<b>LT</b>	True if signed <	<b>JLT \$+1 JMP</b>	<b>1100h 1000h</b>
<b>LTE</b>	True if signed < or =	<b>JGT</b>	<b>1500h</b>
<b>NC</b>	True if No Carry	<b>JOC</b>	<b>1800h</b>
<b>NE</b>	True if equal bit not set	<b>JEQ</b>	<b>1300h</b>
<b>NO</b>	True if No overflow	<b>JNO \$+1 JMP</b>	<b>1901h 1000h</b>
<b>NP</b>	True if Not odd Parity	<b>JOP</b>	<b>1C00h</b>
<b>OC</b>	True if Carry bit is set	<b>JNC</b>	<b>1700h</b>
<b>OO</b>	True if Overflow	<b>JNO</b>	<b>1900h</b>
<b>OP</b>	True if Odd Parity	<b>JOP \$+1 JMP</b>	<b>1C00h 1000h</b>

### 9.12 Assembly Example for Structured Constructs

The following example is designed to show how these jump tokens and structured constructs are used:

Forth	Conventional Assembler	
<b>( GENERALIZED SHIFTER )</b>	<b>* GENERALIZED SHIFTER</b>	
<b>CODE SHIFT</b>	<b>DEF</b>	<b>SHIFT</b>
<b>*SP+ 0 MOV,</b>	<b>SHIFT</b>	<b>MOV *SP+,R0</b>
<b>NE IF,</b>		<b>JEQ L3</b>
<b>*SP 1 MOV,</b>		<b>MOV *SP,R1</b>
<b>0 ABS,</b>		<b>ABS R0</b>
<b>GTE IF,</b>		<b>JLT L1</b>
<b>1 0 SLA,</b>		<b>SLA R1,0</b>
<b>ELSE,</b>		<b>JMP L2</b>
<b>1 0 SRL</b>	<b>L1</b>	<b>SRL R1,0</b>
<b>ENDIF,</b>		
<b>1 *SP MOV,</b>	<b>L2</b>	<b>MOV R1,*SP</b>
<b>ENDIF,</b>		
<b>NEXT,</b>	<b>L3</b>	<b>B *R15</b>

One word of caution is in order. The structured constructs shown above do not check to ensure that the jump target is within range ( +127,-128 words ). This will be a problem only with very large assembly language definitions and will violate the Forth philosophy of small, easily understood words.

## 10 Interrupt Service Routines (ISRs)

The TI-99/4A has the built-in ability to execute an interrupt routine every 1/60 second. This facility has been extended by the TI Forth system so that the routine to be executed at each interrupt period may be written in Forth rather than in assembly language. This is an advanced programming concept and its use depends on the user's knowledge of the TI-99/4A.

The user Variables **ISR** and **INTLNK** are provided to assist the user in using ISRs. Initially, they each contain the address of the link to the Forth ISR handler. To correctly use User Variable **ISR** the following steps should be followed:

### 10.1 *Installing a Forth Language Interrupt Service Routine*

- 1) Create and test a Forth routine to perform the function.
- 2) Determine the Code Field Address (**CFA**) of the routine in (1).
- 3) Write the **CFA** from (2) into **ISR**.
- 4) Write the contents of **INTLNK** into **83C4h (33732)**.

The ISR linkage mechanism is designed so that your interrupt service routine will be allowed to execute immediately after each time the Forth system executes the "NEXT" instruction (as it does at the end of each code word). In addition, the **KEY** routine has been coded so that it also executes "NEXT" after every keyscan whether or not a key has been pressed. The "NEXT" instruction is actually coded in TI Assembler as "**B \*NEXT**" or "**B \*R15**" because workspace register 15 (**R15** or **NEXT**) contains the address of the next instruction to be executed. This executes the same procedure as the TI Forth Assembler word **NEXT**, (see Chapter 9).

Before installing an ISR you should have some idea of how long it takes to execute, keeping in mind that for normal behavior it should execute in less than 16 milliseconds. ISRs that take longer than that may cause erratic sprite motion and sound because of missed interrupts. In addition it is possible to bring the Forth system to a slow crawl by using about 99% of the processor's time for the ISR.

The ISR capability has obvious applications in game software as well as for playing background music or for spooling screens from disk to printer while other activities are taking place. This final application will require that disk buffers and user variables for the spool task be separate from the main Forth task or a very undesirable cross-fertilization of buffers may result. In addition it should be mentioned that disk activity causes all interrupt service activity to halt.

ISRs in Forth can be written as either colon definitions or as **CODE** definitions. The former permits very easy routine creation, and the latter permits the same speed capabilities as routines created by the Editor/Assembler. Both types can be used in a single routine to gain the advantages of both.

## 10.2 An Example of an Interrupt Service Routine

An example of a simple ISR is given below. This example also illustrates some of the problems associated with ISRs and how they can be circumvented. The problems are:

- 1) A contention for PAD between a normal Forth command and the ISR routine.
- 2) Long execution time for the ISR routine. (Even simple routines, especially if they include output conversion routines or other words that nest Forth routines very deeply, will not complete execution in 1/60 second.)

These problems are overcome by moving PAD in the interrupt routine to eliminate the interference between the foreground and the background task. The built-in number formatting routines are quite general and hence pay a performance penalty. This example performs this conversion rather crudely, but fast enough that there is adequate time remaining in each 1/60 second to do meaningful computing.

<b>0 VARIABLE TIMER</b>	( <b>TIMER</b> will hold the current count)
<b>: UP 100 ALLOT ;</b>	(move <b>HERE</b> and thus <b>PAD</b> up 100 bytes)
<b>: DOWN -100 ALLOT DROP<sup>17</sup> ;</b>	(restore <b>PAD</b> to its original location)
<b>: DEMO UP</b>	(move <b>PAD</b> to avoid conflict)
<b>1 TIMER +! TIMER @</b>	(increment <b>TIMER</b> , leave on stack)
<b>PAD DUP 5 +</b>	(ready to loop from <b>PAD</b> + 5 down to <b>PAD</b> + 1)
<b>DO</b>	
<b>0 10 U/</b>	(make positive double, get 1 <sup>st</sup> digit)
<b>SWAP 48 +</b>	(generate ASCII digit)
<b>I C!</b>	(store to <b>PAD</b> )
<b>-1 +LOOP</b>	(decrement loop counter)
<b>PAD 1+ SCRN_START @ 5 VMBW</b>	(write to screen)
<b>DOWN ;</b>	(restore <b>PAD</b> location)

## 10.3 Installing the ISR

To install this ISR the following code may be executed:

<b>INTLNK @</b>	(get the ISR 'hook' to the stack)
<b>' DEMO CFA</b>	(get <b>CFA</b> of the word to be installed as ISR)
<b>ISR !</b>	(place it in user variable <b>ISR</b> )
<b>HEX 83C4 !</b>	(put ISR 'hook' into console interrupt service routine)
	(Note: the CFA must be in user variable <b>ISR</b> before writing to <b>83C4h</b> )

<sup>17</sup> Bug Fix: See Appendix J for the source of the fix. It might be clearer why **DROP** is necessary if it were placed after **+LOOP** instead of in the definition of **DOWN** : After the first pass through the loop in **DEMO** , the remainder from **U/** is consumed, but the quotient is left for the next pass through the loop and, of course, remains on the stack when the loop exits. **DROP** cleans up the stack.

To reverse the installation of the ISR one can either write a 0 to **83C4h** or place the **CFA** of **NOP** (a do-nothing instruction) in user variable **ISR**.

### 10.4 *Some Additional Thoughts Concerning the Use of ISRs*

ISRs are uninterruptible. Interrupts are disabled by the code that branches to your ISR routine and they are not enabled until just before branching back to the foreground routine. *Do not enable interrupts in your interrupt routine.*

- 1) Caution must be exercised when using PABs, changing user variables or using disk buffers in an ISR, as these activities will likely interfere with the foreground task unless duplicate copies are used in the two processes.
- 2) An ISR must never expect nor leave anything on the stacks. It may however use them in the normal manner during execution.
- 3) Disk activity disables interrupts as do most of the other DSRs in the TI-99/4A. An ISR that is installed will not execute during the time interval in which disk data transfer is active. It will resume after the disk is finished. Note that it is possible to **LOAD** from disk while the ISR is active. It will wait for about a second each time the disk is accessed. The dictionary will grow with the resultant movement of **PAD** without difficulty.

# 11 Potpourri

Your TI Forth system has a number of additional features that will be discussed in this chapter. These include a facility to save and load binary images of the dictionary so that applications need not be recompiled each time they are used. Also available are a group of CRU (Communications Register Unit) instructions and a version of **MESSAGE** that does not require a disk to display the standard error messages.

## 11.1 **BSAVE and BLOAD**

**BSAVE**            (*addr scr<sub>1</sub> --- scr<sub>2</sub>*)

The word **BSAVE** is used to save binary images of the dictionary. **BSAVE** requires two entries on the stack:

- 1) The lowest memory address *addr* in the dictionary image to be saved to disk.
- 2) The Forth screen number *scr<sub>1</sub>* to which the saved image will be written.

**BSAVE** will use as many Forth screens as necessary to save the dictionary contents from the address given on the stack to **HERE**. These are saved with 1000 bytes per Forth screen until the entire image is saved. **BSAVE** returns on the stack the number *scr<sub>2</sub>* of the first available Forth screen after the image.

Each Forth screen of the saved image has the following format:

Byte #	Contents
0-1	Address at which the first image byte of this Forth screen will be placed.
2-3	DP for this memory image.
4-5	Contents of <b>CURRENT</b> .
6-7	Contents of <b>CURRENT @</b> .
8-9	Contents of <b>CONTEXT</b> .
10-11	Contents of <b>CONTEXT @</b> .
12-13	Contents of <b>VOC-LINK</b> .
14	The letter 't'.
15	The letter 'i'.
16-23	Not used.
24-1023	Up to 1000 bytes of the memory image.

**BLOAD**            (*scr --- flag*)

**BLOAD** is part of your TI Forth kernel and does not have to be loaded before you can use it. It reverses the **BSAVE** process and makes it possible to bring in an entire application in seconds. **BLOAD** expects a Forth screen number *scr* on the stack. Before performing the **BLOAD** function the 14<sup>th</sup> and 15<sup>th</sup> bytes are checked to see that they contain the letters "ti".



If they do, the load proceeds and **BLOAD** returns a flag of 0 on the stack signifying a successful load. If the letters “ti” are not found, then the **BLOAD** is not performed and a flag of 1 is returned. This facility permits a conditional binary load to be performed and if it fails (wrong disk, *etc.*), other actions can be performed.

Because the **BLOAD** and **BSAVE** facility is designed to start the save (and hence the load) at a user-supplied address, a complete overlay structure can be implemented. *Very important:* The user must ensure that when part of the dictionary is brought in, the remainder of the dictionary (older part) is identical to that which existed when the image was saved.

### 11.1.1 Customizing How TI Forth Boots Up

You may find that you use the same **MENU** choices frequently and would like to load them automatically and quickly each time you boot TI Forth. You can do this by using the Forth word **TASK** as a reference point for **BSAVE**. A no-operation word or null definition, **TASK** is the last word defined in the resident Forth vocabulary of TI Forth and the last word that *cannot* be forgotten using **FORGET**. Its definition is simply

```
: TASK ;
```

Its address can be used to **BSAVE** a personalized TI Forth system disk by using ' **TASK** as the address on the stack for **BSAVE**. If part of your personalized system includes the 64-column editor, you can use the 9 screens starting with screen 21 to save your system image:

```
' TASK 21 BSAVE .
```

(*Be sure to back up the original disk before trying this!*). It is important that you ensure that this procedure does not compromise Forth system screens you may need for your new personalized system. The . after **BSAVE** will report the next available screen from the value left on the stack. Subtracting 21 from that number will tell you how many screens it took to save the binary image in the above **BSAVE** line.

You now need to add the code to load what you have just saved the next time you boot your system. You can also do a little housecleaning by erasing superfluous material from screens 3 and 20:

On Forth screen 3:

- Erase lines 3 – 11. These definitions will be redundant.
- Replace **20 LOAD** on line 2 with **21 BLOAD** to load the rest of the system from **TASK** forward the next time you boot up TI Forth.

On Forth screen 20:

- Erase lines 0 – 8.
- On line 0, put something like: ( **MENU CHOICES** ), to indicate the purpose of lines 9 – 15. You need to keep those lines because **MENU** will list them to the screen regardless of how they read.

### 11.1.2 An Overlay System with BSAVE/BLOAD

As mentioned above, you can implement a complete overlay structure using **BSAVE** and **BLOAD**. It can be a bit tedious to set up, however, because you must ensure that the dictionary structure older than what you load with **BLOAD** is identical to what it was when the binary image was saved with **BSAVE**. If your application always uses **TASK** as the reference point, as in the previous section, for saving and loading all overlays you set up for your application, the situation is actually pretty simple. If, on the other hand, you wish to have the most efficiently running application possible with minimum load/reload times, you will want to load as overlays only those parts of your application that can be considered mutually exclusive or, at least, not redundant functions.

Such an application might be set up as follows:

1. Anticipate screens where overlays will be saved with **BSAVE**.
2. Set up storage (variables, arrays, ...) that is common to two or more overlays.
3. Set up the overlay-loading mechanism in your application to use **BLOAD** to load them. The following example illustrates such a mechanism using the **CASE ... ENDCASE** construct:

```

0 VARIABLE OVLY ( track current ovly# )
: OVLY_LD ( ovly# --- )
  DUP
  CASE
    1 OF 120 BLOAD ENDOF
    2 OF 130 BLOAD ENDOF
    3 OF 140 BLOAD ENDOF
    ( no overlay change if we get here! )
    -1 SWAP ( ENDCASE will DROP top number )
  ENDCASE
  ( 2 cells to here unless fell thru. Top cell: -1|0|1 )
  CASE
    -1 OF ." No choice for overlay " . CR ENDOF
    0 OF OVLY ! ENDOF ( Success! Save new # )
    1 OF ." Failed to load overlay " . CR ENDOF
  ENDCASE ;

```

4. Program a method for determining which overlay is needed for a particular function or set of functions and use **OVLY** to determine whether that overlay needs to be loaded.
5. As the last word of your application before any overlays, define **OVERLAYS** as a null definition to be a reference point for **BSAVE** and make it unforgettable:

```

: OVERLAYS ;
' OVERLAYS NFA FENCE !

```

6. Begin each overlay with the following null definition as a **FORGET** reference point for loading the next overlay source screen prior to saving its binary image with **BSAVE**:

```

: OVLY_STRT ;

```

7. After the successful load (with **BLOAD** ) of an overlay, set **OVLY** to its number as in the example in (3) above.

After programming and debugging the application, save the application and its overlays as follows:

1. Remove all system components from the dictionary that are not required by your application and that are newer than **TASK** . To start with a dictionary with only resident words:
  - a) Execute **-DUMP** to load the definition for **VLIST** .
  - b) Execute **VLIST** to get the name of the word immediately following **TASK** . Remember that **VLIST** lists the dictionary from **HERE** back to older words.
  - c) **FORGET** that word to leave only the resident dictionary. If the word following **TASK** , *i.e.*, listed just before **TASK** by **VLIST** , is **XXX** , then execute **FORGET XXX** .
2. Load all system components required to run your application. At the very least, you will need to load **-BSAVE** to use **BSAVE** to save the binary images for your application and its overlays, even though your application will never need it.
3. Load application.
4. Load first overlay.
5. **BSAVE** application using the address of **TASK** to a free Forth screen:
 

```
' TASK 110 BSAVE .
```
6. **BSAVE** first overlay using the address of **OVERLAYS** to a free Forth screen:
 

```
' OVERLAYS 120 BSAVE .
```
7. For each overlay following the first do the following:
  - a) **FORGET OVLY\_STRT**
  - b) **100 LOAD** (100 should be where the Forth screen for next overlay resides.)
  - c) 

```
' OVERLAYS 130 BSAVE .
```

 (Obviously, 130 should be a different screen for each additional overlay.)

### 11.1.3 An Easier Overlay System in Source Code

The above **BSAVE/BLOAD** method for setting up an overlay system can be very difficult to maintain because of the unforgiving nature of **BLOAD** . Any changes in the application other than the overlay section will almost certainly necessitate re-saving *all* of the overlays. An easier method to maintain is one such as described in *Starting FORTH*, p. 80ff. It will be necessarily slower to load overlays because it uses source screens. You can still save a binary image of the application as above with the first, presumably most used, overlay to minimize load time; but, it still may be better for software changes to **BSAVE** the application without an overlay.

Because you are not using **BSAVE** to save the overlays, you can dispense with one of the null definitions. Let us say you are using **OVERLAYS** , as the word to **FORGET** each time another overlay is loaded. **OVERLAYS** will now separate the main application from the current overlay and should, of course, be the last word of the main application. **OVERLAYS** should obviously not be made unforgettable! The first Forth screen of each overlay should begin with

```
FORGET OVERLAYS      : OVERLAYS ;
```

You can use the same mechanism ( **OVLY\_LD** ) as in the previous section for loading the overlays; but, you will need to change all instances of **BLOAD** to **LOAD** and, of course, the screens will be text screens, not binary images. You will also need to change the code that expects a flag on the stack from **BLOAD** because **LOAD** does not leave a flag.

## 11.2 Conditional Loads

**CLOAD** ( *scr ---* )

The word **CLOAD** has been included in your system to assist in easily managing the process of loading the proper support routines for an application without compiling duplicates of support routines into the dictionary.

**CLOAD** calls the words **<CLOAD>** , **WLITERAL** , and **SLIT** . Their functions are described briefly as follows:

**<CLOAD>** ( *---* )

performs the primary **CLOAD** function and is executed or compiled by **CLOAD** depending on **STATE** .

**SLIT** ( *--- addr* )

is a word designed to handle string literals during execution. Its purpose is to put the address of the string on the stack and step the Forth Instruction Pointer over it.

**WLITERAL** ( *---* )

is used to compile **SLIT** and the desired character string into the current dictionary definition. See the TI Forth Glossary ( Appendix D ) for more detail.

To use **CLOAD** , there must always be a Forth screen number on the stack. The word **CLOAD** must be followed by the word whose conditional presence in the dictionary will determine whether or not the Forth screen number on the stack is loaded.

### 27 CLOAD F00

This instruction, for example, will load Forth screen 27 only if a dictionary search, ( **FIND** ) , fails to find **F00** . **F00** should be the last word loaded by the command **27 LOAD** .

It is also possible to use **CLOAD** to abort the loading of a Forth screen. This is done by using the command:

### 0 CLOAD TESTWORD

If this line of code were located on Forth screen 50, and the word **TESTWORD** was in the present dictionary, the load would abort just as if a **;S** had been encountered.

Caution must be exercised when using **BASE->R** and **R->BASE** with **CLOAD** as these will cause the return stack to be polluted if a **LOAD** is aborted and the **BASE->R** is not balanced by a **R->BASE** at execution time.

### 11.3 *Memory Resident Messages*

**message** ( --- )

If the user desires, he may elect to use a version of **MESSAGE** which is provided on the system disk (Forth screen 84). This version is spelled with lower case **message**. The purpose of this version is to avoid having to place the messages on the diskette in DR0. The code to install this version is supplied on the same Forth screens with the routines. Installing **message** will remove the 5<sup>th</sup> disk buffer from the system and use that memory for storing the error messages. It will then place a patch in the old version of message to cause it to branch to the new routine. Caution must be exercised if **COLD** is executed with the new version in place, as **COLD** will restore the 5th buffer but will not unpatch the old version of **MESSAGE**. After performing the **COLD**, you must reinstall the new **message** or unpatch the old version of **MESSAGE** prior to the system using the word **MESSAGE**. Failure to do this will cause a crash. To repatch **MESSAGE**, the first two words in the parameter field must be restored to be the CFAs of **WARNING** and **@**.

### 11.4 *CRU Words*

**LDCR** (  $n_1$   $n_2$  *addr* --- )

**STCR** (  $n_1$  *addr* ---  $n_2$  )

**TB** ( *addr* --- *flag* )

**SB0** ( *addr* --- )

**SBZ** ( *addr* --- )

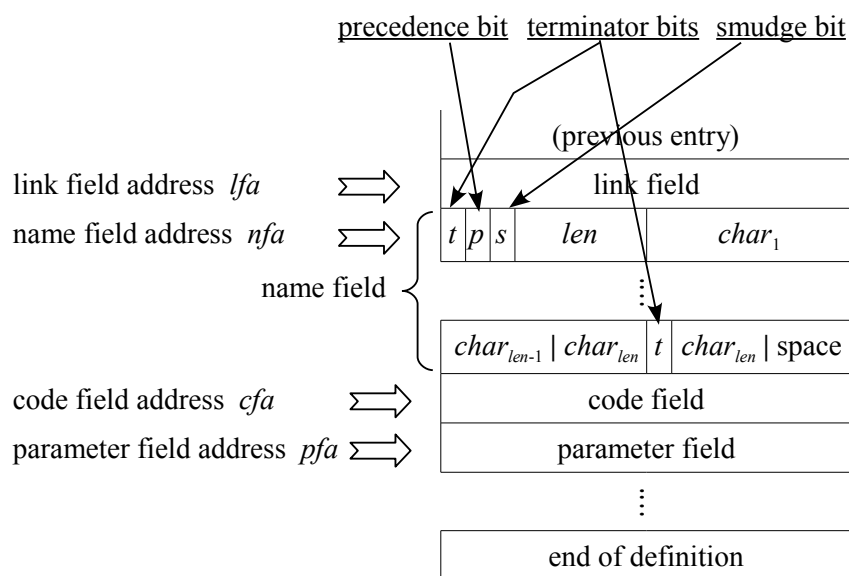
The above five words have been included to assist in performing CRU (Communications Register Unit) related functions. They allow the Forth programmer to perform the **LDCR**, **STCR**, **TB**, **SB0** and **SBZ** operations of the TMS9900 without using the Assembler. The functions of these words will be apparent when someone familiar with these instructions on the TMS9900 examines their definitions in the Glossary ( Appendix D ). Also, see the *Editor/Assembler Manual* for greater detail.

## 12 TI Forth Dictionary Entry Structure

[Editor's Note: As with several of the appendices in this document, this chapter was added by the editor.]

The structure of an entry (a Forth *word*) in the TI Forth dictionary is briefly described in this chapter to give the reader a better understanding of TI Forth and how its dictionary may differ from other Forth implementations.

The dictionary entries are shown here schematically as a stack of single cells of 16 bits each:



At the least, each entry contains a link field (1 cell), a name field (1 – 16 cells), a code field (1 cell) and a parameter field ( $n \geq 1$  cells).

### 12.1 Link Field

The link field is the first field in a definition. It contains the address of the name field of the immediately preceding word in the list to which the its word belongs in the dictionary. The address of this field is termed the link field address *lfa* and may be retrieved by pushing the *pfa* (see § 12.4 ) onto the stack and executing **LFA** .

### 12.2 Name Field

The name field follows the link field and may be as long as 16 cells (32 bytes). The name field address *nfa* points to this field and may be retrieved by pushing the *pfa* (see § 12.4 ) onto the stack and executing **NFA** .

The first byte is the length byte. The three highest bits of the length byte are the beginning terminator bit (**80h**), the precedence bit (**40h**) and the smudge bit (**20h**). These are shown in the above figure as *t*, *p* and *s*, respectively. That leaves 5 bits for the character-length *len* of the name, which is the reason that TI Forth words have a maximum length of 31 characters. The name field in TI Forth always occupies an even number of bytes, *i.e.*, it begins and ends on a cell boundary. The last byte of the name field will be either the last character of the name or a space and will have the highest bit (**80h**) set as the ending terminator bit.

To clarify the above diagram a bit, when the name is only one character long, the first character is obviously the last character and the ending terminator bit will be set in that byte, which results in a name field occupying just one cell.

The terminator bits are flags used by **TRAVERSE** (*q.v.*) to find the beginning or end of the name field, given the address of one end and the direction (+1|-1) to search.

The precedence bit is used to indicate that a word should be executed rather than compiled during compilation. It is set by **IMMEDIATE**, which sets the precedence bit for the most recently completed definition.

The smudge bit is used to hide/unhide a word from a dictionary search during compilation. If the smudge bit is set (**20h**), ' , **-FIND** and (**FIND**) will not find the word. During compilation, the smudge bit is toggled by **SMUDGE** or similar code and toggled again by ; or similar termination code.

### 12.3 Code Field

The code field immediately follows the last cell of the name field. The code field address *cfa* points to this field and may be retrieved by pushing the *pfa* (see § 12.4 ) onto the stack and executing **CFA**. The code field contains the address of the machine-code routine that TI Forth will run when it executes this word and depends on the nature of the word's definition. The following table shows common situations:

Word Defined by	Code Field Contains Address of	What the Runtime Code Does
<b>VARIABLE</b>	Runtime code of <b>VARIABLE</b>	Pushes word's <i>pfa</i> onto stack
<b>CONSTANT</b>	Runtime code of <b>CONSTANT</b>	Pushes contents of word's <i>pfa</i> onto stack
<b>:</b>	Runtime code of <b>:</b>	Executes the list of previously defined words, the addresses of which are stored beginning at this word's <i>pfa</i>
<b>CODE</b>	<i>pfa</i> of word	Executes machine code stored beginning at this word's <i>pfa</i>

## 12.4 *Parameter Field*

The parameter field follows the code field. The parameter field address *pfa* points to this address, which can be retrieved from the *nfa* by executing

```
' cccc PFA
```

The string **cccc** is the name of the desired Forth word and **PFA** is the Forth word that gets the *pfa* from the *nfa*, which **' cccc** gets for you. The contents of the parameter field depend on the type of word defined. The following table shows common situations:

Word Defined by	Parameter Field Contains
<b>VARIABLE</b>	Value of variable
<b>CONSTANT</b>	Value of constant
<b>:</b>	Mostly a list of the addresses (usually their <i>cfas</i> ) of previously defined words that comprise this word's definition
<b>CODE</b>	Machine code comprising this word's runtime code



## Appendix A ASCII Keycodes (Sequential Order)

Character		ASCII Code		Character		ASCII Code	
		hex	decimal			hex	decimal
NUL	<CTRL+,>	00h	0	SP		20h	32
SOH	<CTRL+A> <FCTN+7>	01h	1	!		21h	33
STX	<CTRL+B> <FCTN+4>	02h	2	"	<FCTN+P>	22h	34
ETX	<CTRL+C> <FCTN+1>	03h	3	#		23h	35
EOT	<CTRL+D> <FCTN+2>	04h	4	\$		24h	36
ENQ	<CTRL+E> <FCTN+=>	05h	5	%		25h	37
ACK	<CTRL+F> <FCTN+8>	06h	6	&		26h	38
BEL	<CTRL+G> <FCTN+3>	07h	7	'	<FCTN+O>	27h	39
BS	<CTRL+H> <FCTN+S>	08h	8	(		28h	40
HT	<CTRL+I> <FCTN+D>	09h	9	)		29h	41
LF	<CTRL+J> <FCTN+X>	0Ah	10	*		2Ah	42
VT	<CTRL+K> <FCTN+E>	0Bh	11	+		2Bh	43
FF	<CTRL+L> <FCTN+6>	0Ch	12	,		2Ch	44
CR	<CTRL+M>	0Dh	13	-		2Dh	45
SO	<CTRL+N> <FCTN+5>	0Eh	14	.		2Eh	46
SI	<CTRL+O> <FCTN+9>	0Fh	15	/		2Fh	47
DLE	<CTRL+P>	10h	16	0	<CTRL+0>	30h	48
DC1	<CTRL+Q>	11h	17	1	<CTRL+1>	31h	49
DC2	<CTRL+R>	12h	18	2	<CTRL+2>	32h	50
DC3	<CTRL+S>	13h	19	3	<CTRL+3>	33h	51
DC4	<CTRL+T>	14h	20	4	<CTRL+4>	34h	52
NAK	<CTRL+U>	15h	21	5	<CTRL+5>	35h	53
SYN	<CTRL+V>	16h	22	6	<CTRL+6>	36h	54
ETB	<CTRL+W>	17h	23	7	<CTRL+7>	37h	55
CAN	<CTRL+X>	18h	24	8		38h	56
EM	<CTRL+Y>	19h	25	9	<FCTN+Q> <FCTN+.>	39h	57
SUB	<CTRL+Z>	1Ah	26	:	<FCTN+/>	3Ah	58
ESC	<CTRL+,>	1Bh	27	;	<CTRL+/>	3Bh	59
FS	<CTRL+;>	1Ch	28	<	<FCTN+0>	3Ch	60
GS	<CTRL+=>	1Dh	29	=	<FCTN+;>	3Dh	61
RS	<CTRL+8>	1Eh	30	>	<FCTN+B>	3Eh	62
US	<CTRL+9>	1Fh	31	?	<FCTN+H>	3Fh	63

...continued from previous page—

Character	ASCII Code		Character	ASCII Code	
	hex	decimal		hex	decimal
@ <i>&lt;FCTN+J&gt;</i>	40h	64	`	60h	96
A <i>&lt;FCTN+K&gt;</i>	41h	65	a	61h	97
B <i>&lt;FCTN+L&gt;</i>	42h	66	b	62h	98
C <i>&lt;FCTN+M&gt;</i>	43h	67	c	63h	99
D <i>&lt;FCTN+N&gt;</i>	44h	68	d	64h	100
E	45h	69	e	65h	101
F <i>&lt;FCTN+Y&gt;</i>	46h	70	f	66h	102
G	47h	71	g	67h	103
H	48h	72	h	68h	104
I	49h	73	i	69h	105
J	4Ah	74	j	6Ah	106
K	4Bh	75	k	6Bh	107
L	4Ch	76	l	6Ch	108
M	4Dh	77	m	6Dh	109
N	4Eh	78	n	6Eh	110
O	4Fh	79	o	6Fh	111
P	50h	80	p	70h	112
Q	51h	81	q	71h	113
R	52h	82	r	72h	114
S	53h	83	s	73h	115
T	54h	84	t	74h	116
U	55h	85	u	75h	117
V	56h	86	v	76h	118
W	57h	87	w	77h	119
X	58h	88	x	78h	120
Y	59h	89	y	79h	121
Z	5Ah	90	z	7Ah	122
[	5Bh	91	{	7Bh	123
\	5Ch	92		7Ch	124
]	5Dh	93	}	7Dh	125
^	5Eh	94	~	7Eh	126
_	5Fh	95	DEL	7Fh	127

## Appendix B ASCII Keycodes (Keyboard Order)

Control Key	ASCII Code		Function Key	ASCII Code	
	hex	decimal		hex	decimal
<CTRL+1>	31h	49	<FCTN+1>	03h	3
<CTRL+2>	32h	50	<FCTN+2>	04h	4
<CTRL+3>	33h	51	<FCTN+3>	07h	7
<CTRL+4>	34h	52	<FCTN+4>	02h	2
<CTRL+5>	35h	53	<FCTN+5>	0Eh	14
<CTRL+6>	36h	54	<FCTN+6>	0Ch	12
<CTRL+7>	37h	55	<FCTN+7>	01h	1
<CTRL+8>	1Eh	30	<FCTN+8>	06h	6
<CTRL+9>	1Fh	31	<FCTN+9>	0Fh	15
<CTRL+0>	30h	48	<FCTN+0>	3Ch	60
<CTRL+=>	1Dh	29	<FCTN+=>	05h	5
<CTRL+Q>	11h	11	<FCTN+Q>	39h	57
<CTRL+W>	17h	23	<FCTN+W>	7Eh	126
<CTRL+E>	05h	5	<FCTN+E>	0Bh	11
<CTRL+R>	12h	18	<FCTN+R>	5Bh	91
<CTRL+T>	14h	20	<FCTN+T>	5Dh	93
<CTRL+Y>	19h	25	<FCTN+Y>	46h	70
<CTRL+U>	15h	21	<FCTN+U>	5Fh	95
<CTRL+I>	09h	9	<FCTN+I>	3Fh	63
<CTRL+O>	0Fh	15	<FCTN+O>	27h	39
<CTRL+P>	10h	16	<FCTN+P>	22h	34
<CTRL+>/>	3Bh	59	<FCTN+>/>	3Ah	58

...continued from previous page—

Control Key	ASCII Code		Function Key	ASCII Code	
	hex	decimal		hex	decimal
<CTRL+A>	01h	1	<FCTN+A>	7Ch	124
<CTRL+S>	13h	19	<FCTN+S>	08h	8
<CTRL+D>	04h	4	<FCTN+D>	09h	9
<CTRL+F>	06h	6	<FCTN+F>	7Bh	123
<CTRL+G>	07h	7	<FCTN+G>	7Dh	125
<CTRL+H>	08h	8	<FCTN+H>	3Fh	63
<CTRL+J>	0Ah	10	<FCTN+J>	40h	64
<CTRL+K>	0Bh	11	<FCTN+K>	41h	65
<CTRL+L>	0Ch	12	<FCTN+L>	42h	66
<CTRL+;,>	1Ch	28	<FCTN+;,>	3Dh	61
<CTRL+Z>	1Ah	26	<FCTN+Z>	5Ch	92
<CTRL+X>	18h	24	<FCTN+X>	0Ah	10
<CTRL+C>	03h	3	<FCTN+C>	60h	96
<CTRL+V>	16h	22	<FCTN+V>	7Fh	127
<CTRL+B>	02h	2	<FCTN+B>	3Eh	62
<CTRL+N>	0Eh	14	<FCTN+N>	44h	68
<CTRL+M>	0Dh	13	<FCTN+M>	43h	67
<CTRL+,>	00h	0	<FCTN+,>	38h	56
<CTRL+.>	1Bh	27	<FCTN+.>	39h	57

## Appendix C Differences between *Starting FORTH* and TI Forth

Page	Word	Changes Required
10	<b>BACKSPACE</b>	<b>&lt;FCTN+S&gt;</b> produces a backspace on the TI 99/4A.
10	<b>OK</b>	TI Forth automatically prints a space before “ok”.
16		The TI Forth dictionary can store names up to 31 characters in length.
18	<b>^</b>	Not a special character in TI Forth.
18	<b>."</b>	Will execute inside or outside a colon definition in TI Forth.
42	<b>/MOD</b>	Uses signed numbers in TI Forth. Remainder has sign of dividend.
42	<b>MOD</b>	Uses signed numbers in TI Forth. Remainder has sign of dividend.
50	<b>.S</b>	This word is available on the TI Forth disk. The TI Forth version prints a vertical bar (   ) followed by the stack contents. The stack contents will be printed as unsigned numbers. To use the definition shown you must make the following change because of vocabulary differences: in place of <b>'S</b> use <b>SP@ 2- :</b> <b>: .S CR SP@ 2- S@ @ . -2 +LOOP ;</b>
52	<b>2SWAP</b>	This word is not in TI Forth but can be created with the following definition: <b>: 2SWAP ROT &gt;R ROT R&gt; ;</b>
52	<b>2DUP</b>	This word is not in TI Forth but can be created with the following definition: <b>: 2DUP OVER OVER ;</b>
52	<b>2OVER</b>	This word is not in TI Forth but can be created with the following definition: <b>: 2OVER SP@ 6 + @ SP@ 6 + @ ;</b>
52	<b>2DROP</b>	This word is not in TI Forth but can be created with the following definition: <b>: 2DROP DROP DROP ;</b>
57		When you redefine a word that is already in the dictionary, TI Forth will issue a message saying “WORD isn’t unique”. In the example, a message saying “GREET isn’t unique” would appear.
60		TI Forth supports 90 screens per disk, Numbered 0-89.
63-82		The TI Forth Editor is different (much better) than the editor described in this section. Read the section of this <i>TI Forth Instruction Manual</i> describing the Editor.

Page	Word	Changes Required
83	<b>DEPTH</b>	See comments for page 50.
84	<b>COPY</b>	TI Forth has a disk based word <b>SCOPY</b> (screen copy) which is exactly like <b>COPY</b> , <i>e.g.</i> , <b>: COPY SCOPY ;</b>
84-5		Ignore Editor words.
89ff	<b>THEN</b>	<b>THEN</b> is in the TI Forth vocabulary and is a synonym for the word <b>ENDIF</b> . Many people find <b>ENDIF</b> less confusing than <b>THEN</b> .
91	<b>0&gt;</b>	This word is not in TI Forth but can be created with the following definition: <b>: 0&gt; 0 &gt; ;</b>
91	<b>NOT</b>	This word is not in TI Forth, but can be created with the following definition: <b>: NOT 0= ;</b>
101	<b>?DUP</b>	This word is identical to <b>-DUP</b> in TI Forth. Use the following definition if necessary: <b>: ?DUP -DUP ;</b>
101ff	<b>ABORT"</b>	As with the Forth-79 Standard, TI Forth provides <b>ABORT</b> instead of <b>ABORT"</b> .
102	<b>?STACK</b>	In TI Forth this word automatically calls <b>ABORT</b> and prints the appropriate error message.
107	<b>2*</b>	This word is not in TI Forth, but can be created with the following definition: <b>: 2* DUP + ;</b>
107	<b>2/</b>	This word is not in TI Forth, but can be created with the following definition: <b>: 2/ 1 SRA ;</b>
108	<b>NEGATE</b>	This word is not in TI Forth, but can be created with the following definition: <b>: NEGATE MINUS ;</b>
110	<b>I</b>	This word exists in TI Forth but also has a duplicate definition, <b>R</b> . <b>I</b> and <b>R</b> are identical in function.
110	<b>I'</b>	This word is not in TI Forth, but can be created with the following definition: ( <i>Note: R is a synonym for I.</i> ) <b>: I' R&gt; R SWAP &gt;R ;</b>
112		If you will notice, there is a . (print) missing in the <b>QUADRATIC</b> definition. You must add a . after the last + to make <b>QUADRATIC</b> work

Page	Word	Changes Required
		correctly.
112		Ignore the last two paragraphs. They do not apply.
131		Just a reminder! You must define <b>2DUP</b> and <b>2DROP</b> before the <b>COMPOUND</b> example may be used.
132		There is a mistake in the second definition of <b>TABLE</b> . It should look like this:  <pre> : TABLE CR 11 1 DO       11 1 DO I J * 5 U.R LOOP CR LOOP ; </pre>
134		When you execute the <b>DOUBLING</b> example, an extra number will be printed after 16384. This is because <b>+LOOP</b> behaves a little differently in TI Forth.
136		In the definition of <b>COMPOUND</b> , the <b>CR</b> should precede <b>SWAP</b> instead of <b>LOOP</b> .
137	<b>XX</b>	When an error is detected in TI Forth, the stack is cleared but then the contents of <b>BLK</b> and <b>IN</b> are saved on the stack to assist in locating the error. The stack may be completely cleared with the word <b>SP!</b> .
142	<b>PAGE</b>	This word is not in TI Forth, but can be created with the following definition:  <pre> : PAGE CLS 0 0 GOTOXY ; </pre>
161	<b>U/MOD</b>	This word is not in TI Forth, but can be created with the following definition:  <pre> : U/MOD U/ ; </pre>
161	<b>/LOOP</b>	This word is not in TI Forth.
162	<b>OCTAL</b>	<b>OCTAL</b> does not exist in TI Forth. See p. 163 for definition.
164-5		Numbers in TI Forth may only be punctuated with periods. Commas, slashes and other marks are not permitted. Any number containing a period ( . ) is considered double-length. In later examples using <b>D.</b> and <b>UD.</b> , replace all punctuation in the inputs with decimal points. It is recommended that you not place more than one decimal place in each number if you want valid output.
166	<b>UD.</b>	This word is already defined in TI Forth.
173	<b>D-</b>	This word is not in TI Forth, but can be created with the following definition:  <pre> : D- DMINUS D+ ; </pre>
173	<b>DNEGATE</b>	This word is not in TI Forth, but can be created with the following definition:

Page	Word	Changes Required
		<b>: DENEGATE DMINUS ;</b>
173	<b>DMAX</b>	This word is not in TI Forth, but can be created with the following definition: <b>: DMAX 2OVER 2OVER D- SWAP DROP 0&lt;</b> <b>IF 2SWAP ENDIF</b> <b>2DROP ;</b>
173	<b>DMIN</b>	This word is not in TI Forth, but can be created with the following definition: <b>: DMIN 2OVER 2OVER 2SWAP D- SWAP DROP 0&lt;</b> <b>IF 2SWAP ENDIF</b> <b>2DROP ;</b>
173	<b>D=</b>	This word is not in TI Forth, but can be created with the following definition: <b>: D= D- 0= SWAP 0= AND ;</b>
173	<b>D0=</b>	This word is not in TI Forth, but can be created with the following definition: <b>: D0= 0. D= ;</b>
173	<b>D&lt;</b>	This word is not in TI Forth, but can be created with the following definition: <b>: D&lt; D- SWAP DROP 0&lt;;</b>
173	<b>DU&lt;</b>	This word is not in TI Forth, but can be created with the following definition: <b>: DU&lt; ROT SWAP OVER OVER</b> <b>U&lt;</b> <b>IF (determined less using high order halves)</b> <b>DROP DROP DROP DROP 1</b> <b>ELSE (test if high halves equal)</b> <b>=</b> <b>IF (equal so just test low halves)</b> <b>U&lt;</b> <b>ELSE (test fails)</b> <b>DROP DROP 0</b> <b>ENDIF</b> <b>ENDIF ;</b>
174	<b>M+</b>	This word is not in TI Forth, but can be created with the following definition:



Page	Word	Changes Required
		: <b>M+ 0 D+ ;</b>
174	<b>M/</b>	This word is different in TI Forth and can be changed with the following definition: : <b>M/ M/ SWAP DROP ;</b>
174	<b>M*/</b>	Not available in TI Forth because no triple precision arithmetic has been included. This could be created using either a relatively complicated colon definition or by using the Assembler included with TI Forth.
183ff		Variables in TI Forth are required to be initialized at creation, thus the word <b>VARIABLE</b> takes the top item on the stack and places it into the variable as its initial value. For example, <b>12 VARIABLE DATE</b> both creates the variable <b>DATE</b> and initializes it to 12. If desired, the advanced user can use the words <b>&lt;BUILDS</b> and <b>DOES&gt;</b> to create a new defining word, <b>VARIABLE</b> , which has exactly the behavior of <b>VARIABLE</b> as used in this section. The code to do this is: : <b>VARIABLE &lt;BUILDS 0 , DOES&gt; ;</b>
193	<b>2VARIABLE</b>	This word is not in TI Forth, but can be created with the following definition: : <b>2VARIABLE &lt;BUILDS 0. , , DOES&gt; ;</b>  This definition does not require a number to be on the stack when it is executed.
193	<b>2!</b>	This word is not in TI Forth, but can be created with the following definition: : <b>2! &gt;R R ! R&gt; 2+ ! ;</b>
193	<b>2@</b>	This word is not in TI Forth, but can be created with the following definition: : <b>2@ &gt;R R 2+ @ R&gt; @ ;</b>
193	<b>2CONSTANT</b>	This word is not in TI Forth, but can be created with the following definition: : <b>2CONSTANT &lt;BUILDS , , DOES&gt; 2@ ;</b>  This definition does <i>not</i> require a number on the stack.
199		You must place a 0 on the stack before executing <b>VARIABLE COUNTS 10 ALLOT</b> . This, however, initializes only the first element of the array <b>COUNTS</b> to 0. You must execute either the <b>FILL</b> or <b>ERASE</b> instruction at the bottom of the page to properly initialize the array.
204	<b>DUMP</b>	TI Forth already has a dump instruction which must be loaded from the disk. Dumps are always printed in hexadecimal. See Appendix D for location of <b>DUMP</b> .

Page	Word	Changes Required
207	<b>CREATE</b>	The <b>CREATE</b> word of TI Forth behaves somewhat differently. Hackers should consult fig-Forth documentation.
216	<b>EXECUTE</b>	Because this word operates a little differently in TI Forth, it must be preceded by the word <b>CFA</b> . The example should read: <b>' GREET CFA EXECUTE</b>
217		The example illustrating indirect execution must be modified to work in TI Forth: <b>' GREET CFA POINTER ! POINTER @ EXECUTE</b>
218	<b>[ ' ]</b>	In TI Forth, this word is unnecessary as the word <b>'</b> will take the following word of a definition when used in a definition.
219	<b>NUMBER</b>	In TI Forth, <b>NUMBER</b> is always able to convert double precision numbers.
219	<b>'NUMBER</b>	TI Forth does not use <b>'NUMBER</b> to locate the <b>NUMBER</b> routine.
220		In TI Forth, the name field is variable length and contains up to 31 characters. Also, the link field precedes the name field in TI Forth.
225	<b>EXIT</b>	This word is <b>;S</b> in TI Forth. <b>;S</b> is the word compiled by <b>;</b> so to create <b>EXIT</b> we might use: <b>: EXIT [COMPILE] ;S ; IMMEDIATE</b>
225	<b>I</b>	In TI Forth, the interpreter pointer is called <b>IP</b> , not <b>I</b> .
232		See Chapter 1 in this <i>TI Forth Instruction Manual</i> for instructions for loading elective blocks.
232	<b>RELOAD</b>	This instruction is not available in TI Forth.
233	<b>H</b>	This word is <b>DP</b> ( dictionary pointer ) in TI Forth.
235	<b>'S</b>	In TI Forth, <b>SP@</b> is used instead of <b>'S</b> .
240		See Appendix E in this <i>TI Forth Instruction Manual</i> for a complete list of user variables.
240	<b>&gt;IN</b>	This word is <b>IN</b> in TI Forth.
245	<b>LOCATE</b>	TI Forth does not support <b>LOCATE</b> .
256	<b>COPY</b>	In TI Forth, this word is <b>SCOPY</b> . <b>SCOPY</b> is disk resident. See Appendix D for location.
259	<b>[ ' ]</b>	Change the <b>[ ' ]</b> to <b>'</b> in the bottom example. In TI Forth, <b>'</b> will compile the address of the next word in the colon definition.
261	<b>&gt;TYPE</b>	Unnecessary in non-multiprogramming systems. Not present in TI Forth.
265	<b>RND</b>	TI Forth has two disk resident random number generators: <b>RND</b> and <b>RNDW</b> . See Appendix D for locations and descriptions. See also definitions for <b>SEED</b> and <b>RANDOMIZE</b> .

Page	Word	Changes Required
266	<b>MOVE</b>	In TI Forth, <b>MOVE</b> moves <i>u</i> words in memory, not <i>u</i> bytes. <b>MOVE</b> can be redefined to conform to <i>Starting FORTH</i> : <b>: MOVE 2/ MOVE ;</b>
266	<b>&lt;CMOVE</b>	Not present in TI Forth. Must be created with the Assembler if required. This word is used only when the source and destination regions of a move overlap and the destination is higher than the source.
270	<b>WORD</b>	In TI Forth, the word <b>WORD</b> does not leave an address on the stack.
270	<b>TEXT</b>	This word is not available in TI Forth, but can be defined as follows: <b>: TEXT PAD 72 BLANKS PAD HERE - 1-</b> <b>DUP ALLOT MINUS SWAP WORD ALLOT ;</b> If you want the count to also be stored at PAD, remove the <b>1-</b> from the definition.
277	<b>&gt;BINARY</b>	This is named <b>(NUMBER)</b> in TI Forth.
277		Because <b>WORD</b> does not leave an address on the stack, it is necessary to redefine <b>PLUS</b> as follows: <b>: PLUS 32 WORD DROP NUMBER + ." = " . ;</b>
279	<b>NUMBER</b>	This definition of <b>NUMBER</b> is not compatible with TI Forth.
281	<b>-TEXT</b>	Not in TI Forth. Use the definition on page 282.
292		TI Forth uses the word pair <b>&lt;BUILDS ... DOES&gt;</b> to define a new defining word. <b>&lt;BUILDS</b> calls <b>CREATE</b> as part of its function.
297		To create a byte <b>ARRAY</b> in TI Forth: <b>: ARRAY &lt;BUILDS OVER , * ALLOT</b> <b>DOES&gt; DUP @ ROT * + + 2+ ;</b>
298		Just a reminder! Don't forget to define <b>2*</b> <i>before</i> trying the example at the bottom of the page. Also, replace the word <b>CREATE</b> with <b>&lt;BUILDS</b> .
301	<b>(DO)</b>	This is the runtime behavior of <b>DO</b> just as listed. <b>2&gt;R</b> is not used, however.
301	<b>DO</b>	The given definition of <b>DO</b> is not compatible with TI Forth. TI Forth's definition of <b>DO</b> is much more complex because of compile-time error checking.
303	<b>(LITERAL)</b>	The TI Forth name for this word is <b>LIT</b> .
306		TI Forth remains in compilation mode until a <b>;</b> is typed.

## Appendix D The TI Forth Glossary

TI Forth words appear in this glossary on the left of the entry line for that word and in the order of the ASCII collating sequence, which is displayed as a handy reference at the bottom of each page of this appendix. The Forth screen on which the word is defined is right-justified on the entry line along with the **MENU** choice that will load its definition. If the word is part of the core system, it is listed as “RESIDENT”. The stack effects are listed on the second line. The stack effects on the return stack may also be shown. These will be indicated by “R:” following the “(” as in the following: “( R:  $n$  --- )”, which would mean that a 16-bit number  $n$  is removed from the top of the return stack after the word being described is executed.

### D.1 Explanation of Some Terms and Abbreviations

Term/Abbreviation	Meaning
<i>addr, addr<sub>1</sub>, ...</i>	memory address
<i>b</i>	byte
<i>col</i>	column position
<i>cccc, nnnn, xxxx</i>	string representation
<i>cfa</i>	code field address
<i>char</i>	ASCII character code
<i>count</i>	count ( length )
<i>d, d<sub>1</sub>, d<sub>2</sub>, ...</i>	signed double-precision number
<i>dotcol, dotcol<sub>1</sub>, dotcol<sub>2</sub>, ...</i>	dot column position
<i>dotrow, dotrow<sub>1</sub>, dotrow<sub>2</sub>, ...</i>	dot row position
<i>drive</i>	refers to DR0, DR1, DR2 (DSK1, DSK2, DSK3)
<i>flag</i>	Boolean flag
<i>false</i>	Boolean false flag (value = 0)
<i>f, f<sub>1</sub>, f<sub>2</sub>, ...</i>	floating point number
<i>lfa</i>	link field address
<i>n, n<sub>1</sub>, n<sub>2</sub>, ...</i>	signed single-precision number
<i>nfa</i>	name field address
<i>pfa</i>	parameter field address
<i>row</i>	row position
<i>rem</i>	remainder
<i>scr</i>	screen number
<i>spr</i>	sprite number
<i>true</i>	Boolean true flag (value ≠ 0)
<i>tol</i>	tolerance limit
<i>u</i>	unsigned single-precision number
<i>ud</i>	unsigned double-precision number
<i>vaddr</i>	VDP address

## D.2 TI Forth Word Descriptions

<b>!</b>	RESIDENT
( <i>n addr ---</i> )	
Store 16 bits of <i>n</i> at address. Pronounced “store”.	
<b>!"</b>	SCR 39 -COPY
( <i>addr ---</i> )	
A string terminated with a " must follow this word. This string will be stored at the specified address; however, the character count is not stored.	
<b>!CSP</b>	RESIDENT
( --- )	
Save the stack position in user variable <b>CSP</b> . Used as part of the compiler security.	
<b>#</b>	RESIDENT
( <i>d<sub>1</sub> --- d<sub>2</sub></i> )	
Generate from a double number <i>d<sub>1</sub></i> , the next ASCII character which is placed in an output string. Result <i>d<sub>2</sub></i> is the quotient after division by the value in <b>BASE</b> , and is maintained for further processing. Used between <b>&lt;#</b> and <b>#&gt;</b> . See <b>#S</b> .	
<b>#&gt;</b>	RESIDENT
( <i>d --- addr count</i> )	
Terminates numeric output conversion by dropping <i>d</i> , leaving the text address and character count suitable for <b>TYPE</b> .	
<b>#MOTION</b>	SCR 59 -GRAPH
( <i>n ---</i> )	
Sets sprite numbers 0 to <i>n</i> - 1 in automotion.	
<b>#S</b>	RESIDENT
( <i>d<sub>1</sub> --- d<sub>2</sub></i> )	
Generates ASCII text from <i>d<sub>1</sub></i> in the text output buffer, by the use of <b>#</b> , until a zero double number <i>d<sub>2</sub></i> results. Used between <b>&lt;#</b> and <b>#&gt;</b> .	
<b>'</b>	RESIDENT
( --- <i>pfa</i> )	
Used in the form:	
<b>' nnnn</b>	
Leaves the parameter field address of dictionary word <b>nnnn</b> . As a compiler directive, executes in a colon definition to compile the address of a literal. If the	

word is not found after a search of **CONTEXT** and **CURRENT** , an appropriate error message is given. Pronounced “tick”.

( RESIDENT

( --- )

Used in the form:

( cccc )

Ignore a comment that will be delimited by a right parenthesis on the same Forth screen. May occur during execution or in a colon definition. A blank after the leading parenthesis is required.

(+LOOP) RESIDENT

( n --- )

The runtime procedure compiled by **+LOOP** , which increments the loop index by *n* and tests for loop completion. See **+LOOP** .

(. ") RESIDENT

( --- )

The runtime procedure, compiled by **."** ,which transmits the following in-line text to the selected output device. See **."** .

( ;CODE ) RESIDENT

( --- )

The runtime procedure, compiled by **;CODE** , that rewrites the code field of the most recently defined word to point to the machine code sequence following **;CODE** . See **;CODE** .

(ABORT) RESIDENT

( --- )

Executes after an error when **WARNING** < 0. This word normally executes **ABORT** , but may be redefined (with care) to execute a user's alternative procedure.

(DO) RESIDENT

( --- )

The runtime procedure complied by **DO** which moves the loop control parameters to the return stack. See **DO** .

(DOES>) RESIDENT

( --- )

The run time procedure compiled by **DOES>** .

<b>(FIND)</b>	RESIDENT
$( \text{addr}_1 \text{ addr}_2 \text{ --- false} \mid \text{pfa } b \text{ true} )$	
Searches the dictionary starting at the name field address $\text{addr}_2$ , matching to the text at $\text{addr}_1$ . Returns parameter field address $\text{pfa}$ , length byte $b$ of name field, and <i>true</i> for a good match. If no match is found, only <i>false</i> is left.	
<b>(LINE)</b>	RESIDENT
$( n \text{ scr --- addr count} )$	
Convert the line number $n$ and the Forth screen $\text{scr}$ to the disk buffer address containing the data. A count of 64 indicates the full line text length.	
<b>(LOOP)</b>	RESIDENT
$( \text{---} )$	
The runtime procedure compiled by <b>LOOP</b> , which increments the loop index and tests for loop completion. See <b>LOOP</b> .	
<b>(NUMBER)</b>	RESIDENT
$( d_1 \text{ addr}_1 \text{ --- } d_2 \text{ addr}_2 )$	
Convert the ASCII text beginning at $\text{addr}_1 + 1$ with respect to <b>BASE</b> . The new value is accumulated into double number $d_1$ , being left as $d_2$ . $\text{addr}_2$ is the address of the first unconvertible digit. Used by <b>NUMBER</b> .	
<b>(OF)</b>	RESIDENT
$( \text{---} )$	
The run time procedure compiled by <b>OF</b> .	
<b>*</b>	RESIDENT
$( n_1 \text{ } n_2 \text{ --- } n_3 )$	
Leave the signed product of two signed numbers.	
<b>*/</b>	RESIDENT
$( n_1 \text{ } n_2 \text{ } n_3 \text{ --- } n_4 )$	
Leave the ratio $n_4 = n_1 * n_2 / n_3$ , where all are signed numbers. Retention of an intermediate 31-bit product permits greater accuracy than would be available with the sequence :	
$n_1 \text{ } n_2 \text{ } * \text{ } n_3 \text{ } /$	

**\*/MOD** RESIDENT

(  $n_1$   $n_2$   $n_3$  --- *rem quot* )

Leave the quotient *quot* and remainder *rem* of the operation  $n_1 * n_2 / n_3$ . A 31-bit intermediate product is used as for **\*/**.

**+** RESIDENT

(  $n_1$   $n_2$  ---  $n_3$  )

Leave the sum of  $n_1 + n_2$  as  $n_3$ .

**+!** RESIDENT

(  $n$  *addr* --- )

Add  $n$  to the value at the address. Pronounced “plus store”.

**+ -** RESIDENT

(  $n_1$   $n_2$  ---  $n_3$  )

Apply the sign of  $n_2$  to  $n_1$ , which is left as  $n_3$ .

**+BUFF** RESIDENT

( *addr*<sub>1</sub> --- *addr*<sub>2</sub> *flag* )

Advance the disk buffer address *addr*<sub>1</sub> to the address of the next buffer *addr*<sub>2</sub>. Boolean flag is false when *addr*<sub>2</sub> is the buffer presently pointed to by user variable **PREV**.

**+LOOP** RESIDENT

Runtime: (  $n_1$  --- ) Compilation: ( *addr*  $n_2$  --- )

Used in a colon-definition in the form:

**DO ...  $n_1$  +LOOP**

At run time, **+LOOP** selectively controls branching back to the corresponding **DO** based on  $n_1$ , the loop index and the loop limit. The signed increment  $n_1$  is added to the index and the total compared to the limit. The branch back to **DO** occurs until the new index is equal to or greater than the limit ( $n_1 > 0$ ), or until the new index is equal to or less than the limit ( $n_1 < 0$ ). Upon exiting the loop, the parameters are discarded and execution continues ahead.

At compile time, **+LOOP** compiles the runtime word (**+LOOP**) and the branch offset computed from **HERE** to the address left on the stack by **DO**. The value  $n_2$  is used for compile-time error checking.



<b>,</b>	RESIDENT
( <i>n</i> --- )	
Store <i>n</i> into the next available dictionary memory cell, advancing the dictionary pointer. Pronounced “comma”.	
<b>-</b>	RESIDENT
( <i>n</i> <sub>1</sub> <i>n</i> <sub>2</sub> --- <i>n</i> <sub>3</sub> )	
Leave the difference <i>n</i> <sub>3</sub> of <i>n</i> <sub>1</sub> − <i>n</i> <sub>2</sub> .	
<b>--&gt;</b>	RESIDENT
( --- )	
Continue interpretation with the next Forth screen on disk. Pronounced “next screen”.	
<b>-DUP</b>	RESIDENT
( <i>n</i> <sub>1</sub> --- <i>n</i> <sub>1</sub>   <i>n</i> <sub>1</sub> <i>n</i> <sub>1</sub> )	
Duplicate <i>n</i> <sub>1</sub> only if it is non-zero. This is usually used to copy a value just before <b>IF</b> , to eliminate the need for an <b>ELSE</b> clause to drop it.	
<b>-FIND</b>	RESIDENT
( --- <i>false</i>   <i>pfa count true</i> )	
Accepts the next text word (delimited by blanks) in the input stream to <b>HERE</b> , searches the <b>CONTEXT</b> and then <b>CURRENT</b> vocabularies for a matching entry. If found, the dictionary entry’s parameter field address <i>pfa</i> , its length byte <i>count</i> and <i>true</i> are left. Otherwise, only <i>false</i> is left.	
<b>-TRAILING</b>	RESIDENT
( <i>addr n</i> <sub>1</sub> --- <i>addr n</i> <sub>2</sub> )	
Adjusts the character count <i>n</i> <sub>1</sub> of a text string beginning at <i>addr</i> to suppress the output of trailing blanks, <i>i.e.</i> , the characters at <i>addr</i> + <i>n</i> <sub>2</sub> to <i>addr</i> + <i>n</i> <sub>1</sub> are blanks.	
<b>.</b>	RESIDENT
( <i>n</i> --- )	
Print a number from a signed 16-bit two’s complement value <i>n</i> , converted according to the numeric base stored in <b>BASE</b> . A trailing blank follows. Pronounced “dot”.	
<b>."</b>	RESIDENT
( --- )	
Used in the form:	
<b>." cccc"</b>	

	Compiles an in-line string <b>cccc</b> (delimited by the trailing " ) with an execution procedure to transmit the text to the selected output device. If executed outside a definition, <b>.</b> " will immediately print the text until the final " . See ( <b>.</b> " ).	
<b>.LINE</b>		RESIDENT
	( <i>n scr</i> --- )	
	Print on the terminal device, a line of text from the disk by its line number <i>n</i> and Forth screen number <i>scr</i> . Trailing blanks are suppressed.	
<b>.R</b>		RESIDENT
	( <i>n<sub>1</sub> n<sub>2</sub></i> --- )	
	Print the number <i>n<sub>1</sub></i> right aligned in a field whose width is <i>n<sub>2</sub></i> . No following blank is printed.	
<b>.S</b>		SCR 43 -DUMP
	( --- )	
	Prints the entire contents of the parameter stack as unsigned numbers in the current <b>BASE</b> .	
<b>/</b>		RESIDENT
	( <i>n<sub>1</sub> n<sub>2</sub></i> --- <i>n<sub>3</sub></i> )	
	Leave the quotient <i>n<sub>3</sub></i> of <i>n<sub>1</sub>/n<sub>2</sub></i> .	
<b>/MOD</b>		RESIDENT
	( <i>n<sub>1</sub> n<sub>2</sub></i> --- <i>rem n<sub>3</sub></i> )	
	Leave the remainder <i>rem</i> and signed quotient <i>n<sub>3</sub></i> of <i>n<sub>1</sub>/n<sub>2</sub></i> . The remainder has the sign of the dividend.	
<b>0 1 2 3</b>		RESIDENT
	( --- <i>n</i> )	
	These small numbers are used so often that it is attractive to define them by name in the dictionary as constants.	
<b>0&lt;</b>		RESIDENT
	( <i>n</i> --- <i>flag</i> )	
	Leave a true flag if the number is less than zero (negative), otherwise leave a false flag.	
<b>0=</b>		RESIDENT
	( <i>n</i> --- <i>flag</i> )	
	Leave a true flag if the number is equal to zero, otherwise leave a false flag.	

<b>0BRANCH</b>	RESIDENT
<p>( <i>flag</i> --- )</p> <p>The runtime procedure to conditionally branch. If <i>flag</i> is <i>false</i> (zero), the following in-line parameter is added to the interpretive pointer to branch ahead or back. Compiled by <b>IF</b> , <b>UNTIL</b> and <b>WHILE</b> .</p>	
<b>1+</b>	RESIDENT
<p>( <math>n_1</math> --- <math>n_2</math> )</p> <p>Increment <math>n_1</math> by 1.</p>	
<b>1-</b>	RESIDENT
<p>( <math>n_1</math> --- <math>n_2</math> )</p> <p>Decrement <math>n_1</math> by 1.</p>	
<b>2+</b>	RESIDENT
<p>( <math>n_1</math> --- <math>n_2</math> )</p> <p>Leave <math>n_1</math> incremented by 2 as <math>n_2</math>.</p>	
<b>2-</b>	RESIDENT
<p>( <math>n_1</math> --- <math>n_2</math> )</p> <p>Leave <math>n_1</math> decremented by 2 as <math>n_2</math>.</p>	
<b>:</b>	RESIDENT
<p>( --- )</p> <p>Used in the form called a colon definition:</p> <p style="text-align: center;"><b>: cccc ... ;</b></p> <p>Creates a dictionary entry defining <b>cccc</b> as equivalent to the following sequence of Forth word definitions '...' until the next <b>;</b> or <b>;CODE</b> . The compiling process is done by the text interpreter as long as <b>STATE</b> is non-zero. Other details are that the <b>CONTEXT</b> vocabulary is set to the <b>CURRENT</b> vocabulary and that words with the precedence bit set are executed rather than being compiled.</p>	
<b>: (traceable)</b>	SCR 44 -TRACE
<p>( --- )</p> <p>This is an alternate definition of <b>:</b> that adds the capability to colon definitions of being traced when they are executed. When a colon definition is compiled under the <b>TRACE</b> option, tracing output may be turned on with <b>TRON</b> and off with <b>TROFF</b> prior to executing the word so defined. After <b>TRON</b> is executed, each time the word is executed its name will be output along with the contents of the stack. See <b>TRACE</b> , <b>UNTRACE</b> , <b>TRON</b> and <b>TROFF</b> .</p>	

<b>;</b>	RESIDENT
( --- )	
Terminates a colon definition and stops further compilation. Compiles the runtime <b>;S</b> .	
<b>;CODE</b>	SCR 74 -CODE
( --- )	
Used in the form:	
<b>: cccc ... ;CODE</b>	
<i>assembly mnemonics</i>	
Stop compilation and terminate a new defining word <b>cccc</b> by compiling <b>(;CODE)</b> . Set the <b>CONTEXT</b> vocabulary to <b>ASSEMBLER</b> , assembling to machine code the following mnemonics.	
When <b>cccc</b> later executes in the form:	
<b>cccc nnnn</b>	
the word <b>nnnn</b> will be created with its execution procedure given by the machine code following <b>cccc</b> , that is, when <b>nnnn</b> is executed, it does so by jumping to the code after <b>nnnn</b> . An existing defining word must exist in <b>cccc</b> prior to <b>;CODE</b> .	
<b>;S</b>	RESIDENT
( --- )	
Stop interpretation of a Forth screen. <b>;S</b> is also the runtime word compiled at the end of a colon definition, which returns execution to the calling procedure.	
<b>&lt;</b>	RESIDENT
( $n_1$ $n_2$ --- <i>flag</i> )	
Leave a true flag if $n_1$ is less than $n_2$ , otherwise, leave a false flag.	
<b>&lt;#</b>	RESIDENT
( --- )	
Setup for pictured numeric output formatting using the words:	
<b>&lt;# # #S SIGN #&gt;</b>	
The conversion is done on a double number producing text at <b>PAD</b> (working downward toward <b>HERE</b> ), eventually suitable for output by <b>TYPE</b> . The picture template between <b>&lt;#</b> and <b>#&gt;</b> represents the output picture from right to left, <i>i.e.</i> , the rightmost digit is processed first. See <b>#</b> , <b>#S</b> , <b>SIGN</b> , <b>#&gt;</b> and <b>HOLD</b> .	

**<BUILDS**

RESIDENT

( --- )

Used within a colon-definition:

: **cccc** **<BUILDS ... DOES>** ... ;

Each time **cccc** is executed, **<BUILDS** defines a new word with a high level execution procedure. Executing **cccc** in the form:

**cccc nnnn**

uses **<BUILDS** to create a dictionary entry for **nnnn**. When **nnnn** is later executed, it has the address of its parameter area on the stack and executes the words after **DOES>** in **cccc**. **<BUILDS** and **DOES>** allow runtime procedures to be written in high-level rather than in assembler code (as required by **;CODE**).

**<CLOAD>**

SCR 21 BOOT SCR

( --- )

The runtime procedure compiled by **CLOAD**.**=**

RESIDENT

(  $n_1$   $n_2$  --- *flag* )Leave a true flag if  $n_1 = n_2$ , otherwise leave a false flag.**=CELLS**

RESIDENT

(  $addr_1$  ---  $addr_1$  |  $addr_2$  )

This instruction expects an address or an offset to be on the stack. If this number is odd, it is incremented by 1 to put it on the next even word boundary. Otherwise, it remains unchanged.

**>**

RESIDENT

(  $n_1$   $n_2$  --- *flag* )Leave a true flag if  $n_1$  is greater than  $n_2$ , otherwise leave a false flag.**>ARG**

SCR 45 -FLOAT

(  $f$  --- )Moves a floating point number  $f$  from the stack into the **ARG** register.**>F**

SCR 48 -FLOAT

( ---  $f$  )

This instruction expects to be followed by a string representing a legitimate floating point number terminated by a space. This string is converted into floating point and placed on the stack. This instruction can be used in colon definitions or directly from the keyboard.

<b>&gt;FAC</b>	SCR 45 -FLOAT
( <i>f</i> --- )	
Moves a floating point number from the stack into the <b>FAC</b> register.	
<b>&gt;R</b>	RESIDENT
( <i>n</i> --- ) ( R: --- <i>n</i> )	
Remove a number from the parameter stack and place as the most accessible number on the return stack. Use should be balanced with <b>R&gt;</b> in the same definition.	
<b>?</b>	RESIDENT
( <i>addr</i> --- )	
Print the value contained at address <i>addr</i> in free format according to the current <b>BASE</b> . This word is short for the two words, <b>@ .</b> .	
<b>?COMP</b>	RESIDENT
( --- )	
Issue error message if not compiling.	
<b>?CSP</b>	RESIDENT
( --- )	
Issue error message if stack position differs from value saved in <b>CSP</b> .	
<b>?ERROR</b>	RESIDENT
( <i>flag n</i> --- )	
Issue an error message number <i>n</i> if the Boolean flag is true.	
<b>?EXEC</b>	RESIDENT
( --- )	
Issue an error message if not executing.	
<b>?FLERR</b>	SCR 49 -FLOAT
( --- )	
Determines if the previous floating point operation resulted in an error. An appropriate error message is printed upon finding an error.	
<b>?KEY</b>	RESIDENT
( --- <i>char</i> )	
Scans the keyboard for input. If no key is pressed, a 0 is left on the stack. Else, the ASCII code of the key pressed is left on the stack.	

<b>?KEY8</b>	RESIDENT
( --- $n$ )	
Scans the keyboard for input. If no key is pressed, a 0 is left on the stack. Else, the 8-bit code of the key pressed is left on the stack.	
<b>?LOADING</b>	RESIDENT
( --- )	
Issue an error message if not loading.	
<b>?PAIRS</b>	RESIDENT
( $n_1$ $n_2$ --- )	
Issue an error message if $n_1$ does not equal $n_2$ . The message indicates that compiled conditionals do not match.	
<b>?STACK</b>	RESIDENT
( --- )	
Issue an error message if the stack is out of bounds.	
<b>?TERMINAL</b>	RESIDENT
( --- <i>flag</i> )	
Perform a test on the terminal keyboard for actuation of the break key ( <b>&lt;BREAK&gt;</b> ). A true flag indicates actuation. On the TI-99/4A, <b>&lt;FCTN+4&gt;</b> , <b>&lt;BREAK&gt;</b> and <b>&lt;CLEAR&gt;</b> are all the same key.	
<b>@</b>	RESIDENT
( <i>addr</i> --- $n$ )	
Leave the 16-bit contents $n$ of <i>addr</i> .	
<b>A\$\$M</b>	SCR 82 -ASSEMBLER
( --- )	
This word is compiled into the <b>FORTH</b> vocabulary and marks the end of the <b>ASSEMBLER</b> vocabulary. It is used by <b>CLOAD</b> .	
<b>ABORT</b>	RESIDENT
( --- )	
Clear the stacks and enter the execution state. Return control to the operator's terminal, printing an appropriate message.	
<b>ABS</b>	RESIDENT
( $n_1$ --- $n_2$ )	
Leave the absolute value of $n_1$ as $n_2$ .	

**AGAIN**

RESIDENT

Compilation: ( *addr n* --- )

Used in a colon definition in the form:

**BEGIN ... AGAIN**

At run time, **AGAIN** forces execution to return to corresponding **BEGIN** . There is no effect on the stack. Execution cannot leave the loop unless **R> DROP** is executed one level below.

At compile time, **AGAIN** compiles **BRANCH** with an offset from **HERE** to *addr*. The value *n* is used for compile time error checking.

**ALLOT**

RESIDENT

( *n* --- )

Add the signed number *n* to the dictionary pointer **DP** . May be used to reserve dictionary space or re-origin memory.

**ALTIN**

RESIDENT

( --- *addr* )

A user variable whose value is 0 if input is coming from the keyboard else its value is a pointer to the VDP address where the PAB (Peripheral Access Block) for the alternate input device is located.

**ALTOUT**

RESIDENT

( --- *addr* )

A user variable whose value is 0 if output is going to the monitor else its value is a pointer to the VDP address where the PAB (Peripheral Access Block) for the alternate output device is located.

**AND**

RESIDENT

( *n*<sub>1</sub> *n*<sub>2</sub> --- *n*<sub>3</sub> )

Leave the bitwise logical AND of *n*<sub>1</sub> and *n*<sub>2</sub> as *n*<sub>3</sub>.

**APPND**

SCR 69 -FILE

( --- )

Assigns the APPEND attribute to the file whose PAB (Peripheral Access Block) is pointed to by **PAB-ADDR** .

**ARG**

SCR 45 -FLOAT

( --- *addr* )

A constant which contains the address of the **ARG** register.



**ASSEMBLER**

SCR 74 -ASSEMBLER

( --- )

The name of the TI Forth Assembler vocabulary. Execution makes **ASSEMBLER** the **CONTEXT** vocabulary. **ASSEMBLER** is immediate, so it will execute during the creation of a colon definition to select this vocabulary at compile time. See **VOCABULARY** .

**ATN**

SCR 50 -FLOAT

(  $f_1$  ---  $f_2$  )

Calculates the arctangent in radians of  $f_1$  leaving the floating point result  $f_2$  on the stack.

**B/BUF**

RESIDENT

( ---  $n$  )

This constant leaves the number of bytes  $n$  per disk buffer, the byte count read from disk by **BLOCK** .

**B/BUF\$**

RESIDENT

( --- *addr* )

A user variable which contains the number of bytes per buffer.

**B/SCR**

RESIDENT

( ---  $n$  )

This constant leaves the number of blocks per editing screen. By convention, an editing screen is 1024 bytes organized as 16 lines of 64 characters each.

**B/SCR\$**

RESIDENT

( --- *addr* )

A user variable which contains the number of blocks per Forth screen.

**BACK**

RESIDENT

( *addr* --- )

Calculate the backward branch offset from **HERE** to *addr* and compile into the next available dictionary memory address.

**BASE**

RESIDENT

( --- *addr* )

A user variable containing the current number base used for input and output conversion.

**BASE->R**

RESIDENT

( --- )

Place the current number base on the return stack. Caution must be exercised when using **BASE->R** and **R->BASE** with **CLOAD** as these will cause the return stack to be polluted if a **LOAD** is aborted and the **BASE->R** is not balanced by a **R->BASE** at execution time. See **R->BASE** .

**BEEP**

SCR 60 -GRAPH

( --- )

Produces the sound associated with correct input or prompting.

**BEGIN**

RESIDENT

Compilation: ( --- *addr n* )

Occurs in a colon-definition in the form:

**BEGIN ... UNTIL**

**BEGIN ... AGAIN**

**BEGIN ... WHILE ... REPEAT**

At runtime, **BEGIN** marks the start of a sequence that may be repetitively executed. It serves as a return point from the corresponding **UNTIL** , **AGAIN** or **REPEAT** . When executing **UNTIL** , a return to **BEGIN** will occur if the top of the stack is false; for **AGAIN** and **REPEAT** a return to **BEGIN** always occurs.

At compile time, **BEGIN** leaves its return address *addr* and *n* for compiler error checking.

**BL**

RESIDENT

( --- *char* )

A constant that leaves the ASCII value for “blank”.

**BLANKS**

RESIDENT

( *addr count* --- )

Fill an area of memory beginning at *addr* with *count* blanks.

**BLK**

RESIDENT

( --- *addr* )

A user variable containing the block number being interpreted. If zero, input is being taken from the terminal input buffer.

**BLOAD**

RESIDENT

( *scr* --- *flag* )

Loads the binary image at *scr* which was created by **BSAVE** . **BLOAD** returns a true flag (1) if the load was not successful and a false flag (0) if the load was successful.

<b>BLOCK</b>	RESIDENT
( <i>n</i> --- <i>addr</i> )	
Leave the memory address of the block buffer containing block <i>n</i> . If the block is not already in memory, it is transferred from disk to whichever buffer was least recently written. If the block occupying that buffer has been marked as updated, it is written to disk before block <i>n</i> is read into the buffer. See also <b>BUFFER</b> , <b>R/W</b> , <b>UPDATE</b> and <b>FLUSH</b> .	
<b>BOOT</b>	RESIDENT
( --- )	
Examines the Forth screen designated as the boot screen (screen #3). If it contains only displayable characters (ASCII 32 – 127), it performs a <b>LOAD</b> on that screen.	
<b>BRANCH</b>	RESIDENT
( --- )	
The runtime procedure to unconditionally branch. An in-line offset is added to the interpretive pointer (IP) to branch ahead or back. <b>BRANCH</b> is compiled by <b>ELSE</b> , <b>AGAIN</b> , <b>REPEAT</b> , and <b>ENDOF</b> .	
<b>BSAVE</b>	SCR 83 -BSAVE
( <i>addr scr<sub>1</sub></i> --- <i>scr<sub>2</sub></i> )	
Places a binary image (starting at <i>scr<sub>1</sub></i> and going as far as necessary) of all dictionary contents between <i>addr</i> and <b>HERE</b> . The next available Forth screen number <i>scr<sub>2</sub></i> is returned on the stack. See <b>BLOAD</b> .	
<b>BUFFER</b>	RESIDENT
( <i>n</i> --- <i>addr</i> )	
Obtain the next memory buffer, assigning it to block <i>n</i> . If the contents of the buffer is marked as updated, it is written to the disk. The block is not read from the disk. The address left is the first cell within the buffer for data storage.	
<b>C!</b>	RESIDENT
( <i>b addr</i> --- )	
Store the low-order byte (8 bits) of <i>b</i> (16-bit number on the stack) at <i>addr</i> .	
<b>C,</b>	RESIDENT
( <i>b</i> --- )	
Store the low-order byte (8 bits) of <i>b</i> (16-bit number on the stack) into the next available dictionary byte, advancing the dictionary pointer. This instruction should be used with caution on byte addressing, word oriented computers such as the TI 9900.	

<b>C/L</b>	RESIDENT
( --- <i>n</i> )	
Returns on the stack the number of characters per line.	
<b>C/L\$</b>	RESIDENT
( --- <i>addr</i> )	
A user variable whose value is the number of characters per line.	
<b>C@</b>	RESIDENT
( <i>addr</i> --- <i>b</i> )	
Leave the 8-bit contents of the memory address on the stack.	
<b>CASE</b>	RESIDENT
Compilation: ( --- ) Runtime: ( <i>n</i> --- <i>n</i> )	
Used in a colon definition to initiate the construct:	
<pre> CASE     <i>n</i><sub>1</sub> OF ... ENDOF     <i>n</i><sub>2</sub> OF ... ENDOF     ... ENDCASE </pre>	
At runtime, <b>CASE</b> itself does nothing with the number <i>n</i> on the stack; but, it must be there for <b>OF</b> or <b>ENDCASE</b> to consume. If <i>n</i> = <i>n</i> <sub>1</sub> , the code between the immediately following <b>OF</b> ... <b>ENDOF</b> is executed. Execution then continues after <b>ENDCASE</b> . If <i>n</i> does not match any of the values preceding any <b>OF</b> , the code between the last <b>ENDOF</b> and <b>ENDCASE</b> is executed and may consume <i>n</i> ; but, one cell <i>must</i> be left for <b>ENDCASE</b> to consume. Execution then continues after <b>ENDCASE</b> .	
<b>CFA</b>	RESIDENT
( <i>pfa</i> --- <i>cfa</i> )	
Convert the parameter field address <i>pfa</i> of a definition to its code field address <i>cfa</i> .	
<b>CHAR</b>	SCR 57 -GRAPH
( <i>n</i> <sub>1</sub> <i>n</i> <sub>2</sub> <i>n</i> <sub>3</sub> <i>n</i> <sub>4</sub> <i>char</i> --- )	
Defines character # <i>char</i> to have the pattern specified by the 4 numbers ( <i>n</i> <sub>1</sub> , <i>n</i> <sub>2</sub> , <i>n</i> <sub>3</sub> , <i>n</i> <sub>4</sub> ) on the stack. The definition for character #0 by default resides at <b>800h</b> . Each character definition is 8 bytes long with each number on the stack representing two bytes.	

<b>CHAR-CNT!</b>	SCR 69 -FILE
( <i>n</i> --- )	
Used in file I/O to store in the current PAB the character count of a record to be transmitted by <b>WRT</b> .	
<b>CHAR-CNT@</b>	SCR 69 -FILE
( --- <i>n</i> )	
Used in file I/O to retrieve from the current PAB the character count of a record that has been read. Used by <b>RD</b> .	
<b>CHARPAT</b>	SCR 57 -GRAPH
( <i>char</i> --- <i>n</i> <sub>1</sub> <i>n</i> <sub>2</sub> <i>n</i> <sub>3</sub> <i>n</i> <sub>4</sub> )	
Places the 4-number (8-byte) pattern of a specified character <i>char</i> on the stack. By default, the definition for character #0 resides at <b>800h</b> .	
<b>CHK-STAT</b>	SCR 68 -FILE
( --- )	
Checks for errors following an I/O operation. If an error has occurred, an appropriate message is printed.	
<b>CLEAR</b>	RESIDENT
( <i>scr</i> --- )	
Fills the designated Forth screen with blanks.	
<b>CLINE</b>	SCR 66 -64SUPPORT
( <i>addr count n</i> --- )	
Prints one line of tiny characters on the display screen. <b>CLINE</b> expects on the stack the address <i>addr</i> of the line to be written in memory, the number of characters <i>count</i> in that line, and the line number <i>n</i> on which it is to be written on the display screen. <b>CLINE</b> calls <b>SMASH</b> to do the actual work. See <b>SMASH</b> and <b>CLIST</b> .	
<b>CLIST</b>	SCR 66 -64SUPPORT
( <i>scr</i> --- )	
Lists the specified Forth screen in tiny characters to the monitor. <b>CLIST</b> executes a multiple call to <b>CLINE</b> . See <b>CLINE</b> and <b>TCHAR</b> .	
<b>CLOAD</b>	SCR 21 BOOT SCR
( <i>scr</i> --- )	
Used in the form:	
<b><i>scr</i> CLOAD nnnn</b>	

**CLOAD** will load Forth screen *scr* only if the word **nnnn** (the last word loaded by *scr*) is not in the **CONTEXT** vocabulary. A screen number of 0 will suppress loading of the current Forth screen if the specified word has already been compiled.

**CLR-STAT** SCR 68 -FILE

( --- )

Clears (zeroes) the error code in bits 0–2 of the flag/status byte of the PAB (Peripheral Access Block) pointed to by **PAB-ADDR**.

**CLS** SCR 33 -SYNONYMS

( --- )

Clears display screen by filling the screen image table with blanks. The screen image table runs from **SCRN\_START** to **SCRN\_END**.

**CLSE** SCR 71 -FILE

( --- )

Closes the file whose PAB (Peripheral Access Block) is pointed to by **PAB-ADDR**.

**CMOVE** RESIDENT

( *addr<sub>1</sub> addr<sub>2</sub> count* --- )

Move *count* number of bytes from *addr<sub>1</sub>* to *addr<sub>2</sub>*. The contents of *addr<sub>1</sub>* is moved first proceeding toward high memory.

**CODE** SCR 74 -CODE

( --- )

A defining word initializing the definition of a code (assembly) word.

**COINC** SCR 61 -GRAPH

( *spr<sub>1</sub> spr<sub>2</sub> tol* --- *flag* )

Detects a coincidence between two given sprites within a specified tolerance limit *tol*. A true flag indicates a coincidence.

**COINCALL** SCR 61 -GRAPH

( --- *flag* )

Detects a coincidence between the visible portions of any two sprites on the display screen. A true flag indicates a coincidence.

**COINCXY** SCR 61 -GRAPH

( *dotcol dotrow spr tol* --- *flag* )

Detects a coincidence between a specified sprite and a given point (*dotcol, dotrow*) within a given tolerance limit *tol*. A true flag indicates a coincidence.

<b>COLD</b>	RESIDENT
( --- )	
The <b>COLD</b> start procedure to adjust the dictionary pointer to the minimum standard and restart via <b>ABORT</b> . May be called from the terminal to remove application programs and restart. <b>COLD</b> calls <b>BOOT</b> prior to calling <b>ABORT</b> .	
<b>COLOR</b>	SCR 58 -GRAPH
( $n_1$ $n_2$ $n_3$ --- )	
Causes a specified character set $n_3$ to have the given foreground $n_1$ and background $n_2$ colors.	
<b>COLTAB</b>	SCR 57 -GRAPH
( --- <i>vaddr</i> )	
A constant whose value is the beginning VDP address of the color table. The default value is <b>380h</b> .	
<b>COMPILE</b>	RESIDENT
( --- )	
When the word containing <b>COMPILE</b> executes, the execution address of the word following <b>COMPILE</b> is copied (compiled) into the dictionary. This allows specific compilation situations to be handled in addition to simply compiling an execution address (which the interpreter already does).	
<b>CONSTANT</b>	RESIDENT
( $n$ --- )	
A defining word used in the form:	
<b><math>n</math> CONSTANT cccc</b>	
to create word <b>cccc</b> , with its parameter field containing $n$ . When <b>cccc</b> is later executed, it will push the value of $n$ to the stack.	
<b>CONTEXT</b>	RESIDENT
( --- <i>addr</i> )	
A user variable containing a pointer to the vocabulary within which dictionary searches will first begin.	
<b>COS</b>	SCR 50 -FLOAT
( $f_1$ --- $f_2$ )	
Calculates the cosine of $f_1$ radians and leaves the floating point result $f_2$ on the stack.	

<b>COUNT</b>	RESIDENT
( <i>addr</i> <sub>1</sub> --- <i>addr</i> <sub>2</sub> <i>n</i> )	
Leave the byte address <i>addr</i> <sub>2</sub> and byte count <i>n</i> of a message text beginning at <i>addr</i> <sub>1</sub> . It is presumed that the first byte at <i>addr</i> <sub>1</sub> contains the text byte count and the actual text starts with the second byte. Typically, <b>COUNT</b> is followed by <b>TYPE</b> .	
<b>CR</b>	RESIDENT
( --- )	
Transmit a carriage return and a line feed to the selected output device.	
<b>CREATE</b>	RESIDENT
( --- )	
A defining word used in the form:	
<b>CREATE cccc</b>	
by such words as <b>CODE</b> and <b>CONSTANT</b> to create a dictionary header for a Forth definition. The code field contains the address of the word's parameter field. The new word is created in the <b>CURRENT</b> vocabulary.	
<b>CSP</b>	RESIDENT
( --- <i>addr</i> )	
A user variable temporarily storing the stack pointer position for compilation error checking.	
<b>CURPOS</b>	RESIDENT
( --- <i>addr</i> )	
A user variable that stores the current VDP (Visual Display Processor) cursor position.	
<b>CURRENT</b>	RESIDENT
( --- <i>addr</i> )	
A user variable pointing to the vocabulary into which new definitions will be compiled.	
<b>D+</b>	RESIDENT
( <i>d</i> <sub>1</sub> <i>d</i> <sub>2</sub> --- <i>d</i> <sub>3</sub> )	
Leave the double number sum of two double numbers ( <i>d</i> <sub>3</sub> = <i>d</i> <sub>1</sub> + <i>d</i> <sub>2</sub> ).	
<b>D+-</b>	RESIDENT
( <i>d</i> <sub>1</sub> <i>n</i> --- <i>d</i> <sub>2</sub> )	
Apply the sign of <i>n</i> to the double number <i>d</i> <sub>1</sub> , leaving it as <i>d</i> <sub>2</sub> .	



<b>D.</b>	RESIDENT
( <i>d</i> --- )	
Print a signed double number from a 32-bit two's complement value <i>d</i> . The high-order 16 bits are most accessible on the stack. Conversion is performed according to the current <b>BASE</b> . A blank follows. Pronounced "d dot".	
<b>D.R</b>	RESIDENT
( <i>d n</i> --- )	
Print a signed double number <i>d</i> right-aligned in a field <i>n</i> characters wide.	
<b>DABS</b>	RESIDENT
( <i>d</i> <sub>1</sub> --- <i>d</i> <sub>2</sub> )	
Leave the absolute value <i>d</i> <sub>2</sub> of a double number <i>d</i> <sub>1</sub> .	
<b>DCOLOR</b>	RESIDENT
( --- <i>addr</i> )	
A variable which contains the dot-color information used by <b>DOT</b> . Its value may be a two-digit HEX number which defines the foreground and background color, or it may be -1 which means no color information is changed in the VDP (Visual Display Processor).	
<b>DDOT</b>	SCR 63 -GRAPH
( <i>dotcol dotrow</i> --- <i>b vaddr</i> )	
The assembly code routine called by <b>DOT</b> . It expects a dot column and a dot row on the stack and returns a byte <i>b</i> with only one bit set and a VDP address <i>vaddr</i> . The dot referenced by ( <i>dotcol, dotrow</i> ) is translated by <i>ddot</i> to the address <i>vaddr</i> of the byte containing it and a mask <i>b</i> that locates the dot within the byte. [ <i>Editor's Note:</i> The original glossary entry was missing <i>b</i> and its description.]	
<b>DECIMAL</b>	RESIDENT
( --- )	
Set the numeric conversion <b>BASE</b> for decimal input/output.	
<b>DEFINITIONS</b>	RESIDENT
( --- )	
Used in the form:	
<b>cccc DEFINITIONS</b>	
Set the <b>CURRENT</b> vocabulary to the <b>CONTEXT</b> vocabulary. In the example, executing vocabulary name <b>cccc</b> makes it the <b>CONTEXT</b> vocabulary and executing <b>DEFINITIONS</b> makes both specify vocabulary <b>cccc</b> .	

**DELALL** SCR 61 -GRAPH

( --- )

Delete all sprites.

**DELSPR** SCR 61 -GRAPH

( *spr* --- )

Delete the specified sprite.

**DIGIT** RESIDENT

( *char*  $n_1$  --- *false* |  $n_2$  *true* )

Convert the ASCII character *char* (using number base  $n_1$ ) to its binary equivalent  $n_2$ , accompanied by a true flag. If the conversion is invalid, leave only a false flag. For example, **DECIMAL 53 10 DIGIT** will leave **5 1** on the stack because 53 is the ASCII code for '5' and is a legitimate digit in base 10. On the other hand, **DECIMAL 74 16 DIGIT** will leave only **0** on the stack because 74 is the ASCII code for 'J' and is *not* a legitimate digit in base 16. However, **DECIMAL 74 20 DIGIT** will leave **19 1** on the stack because 'J' *is* a legitimate digit in base 20.

**DISK\_BUF** RESIDENT

( --- *addr* )

A user variable that points to the first byte in VDP RAM of the 1K disk buffer.

**DISK-HEAD** SCR 40 -COPY

( --- )

Writes a disk header on Forth screen 0 that makes the disk compatible with the TI 99/4A Disk Manager and with TI BASIC.

**DISK\_HI** RESIDENT

( --- *addr* )

A user variable which contains the Forth screen number immediately above the Forth screen range wherein screen writes are permitted.

**DISK\_LO** RESIDENT

( --- *addr* )

A user variable which contains the first Forth screen number of the range wherein disk writes are permitted.

**DISK\_SIZE** RESIDENT

( --- *addr* )

A user variable whose value is the number of Forth screens logically assigned to a diskette.

<b>DLITERAL</b>	RESIDENT
Compilation: ( <i>d</i> --- ) Runtime: ( --- <i>d</i> ) Interpretation: ( --- ) Same behavior as <b>LITERAL</b> , <i>q.v.</i> , except for a double number <i>d</i>	
<b>DLT</b>	SCR 71 -FILE
( --- ) The file I/O routine that deletes the file whose PAB (Peripheral Access Block) is pointed to by <b>PAB-ADDR</b> .	
<b>DMINUS</b>	RESIDENT
( <i>d</i> <sub>1</sub> --- <i>d</i> <sub>2</sub> ) Convert <i>d</i> <sub>1</sub> to its double number two's complement <i>d</i> <sub>2</sub> .	
<b>DMODE</b>	SCR 63 -GRAPH
( --- <i>addr</i> ) A variable that determines which dot mode is currently in effect. A <b>DMODE</b> value of 0 indicates DRAW mode, a value of 1 indicates UNDRAW mode, and a value of 2 indicates DOT-TOGGLE mode. This variable is set by the <b>DRAW</b> , <b>UNDRAW</b> and <b>DTOG</b> words.	
<b>DO</b>	RESIDENT
Compilation: ( <i>addr n</i> --- ) Runtime: ( <i>n</i> <sub>1</sub> <i>n</i> <sub>2</sub> --- ) Occurs in a colon-definition in the form: <div style="text-align: center;"> <b>DO ... LOOP</b>  <b>DO ... +LOOP</b> </div> When compiling within the colon-definition, <b>DO</b> compiles ( <b>DO</b> ) , leaving the following address <i>addr</i> and <i>n</i> for later error checking. At run time, <b>DO</b> begins a sequence with repetitive execution controlled by a loop limit <i>n</i> <sub>1</sub> and an index with initial value <i>n</i> <sub>2</sub> . <b>DO</b> removes these from the stack. Upon reaching <b>LOOP</b> , the index is incremented by one. Until the new index equals or exceeds the limit, execution loops back to just after <b>DO</b> , otherwise the loop parameters are discarded and execution continues ahead. Both <i>n</i> <sub>1</sub> and <i>n</i> <sub>2</sub> are determined at runtime and may be the result of other operations. Within a loop, <b>I</b> will copy the current value of the index to the stack. See <b>I</b> , <b>LOOP</b> , <b>+LOOP</b> and <b>LEAVE</b> .	
<b>DOES&gt;</b>	RESIDENT
( --- ) A word which defines the runtime action within a high-level defining word. <b>DOES&gt;</b> alters the code field and first parameter of the new word to execute the sequence of compiled word addresses following <b>DOES&gt;</b> . It is always used in combination with	

**<BUILDS** . When the **DOES>** part executes it begins with the address of the first parameter of the new word on the stack. This allows interpretation using this area or its contents. Typical uses include the Forth assembler, multidimensional arrays and compiler generation.

**DOT** SCR 63 -GRAPH

( *dotcol dotrow ---* )

Plots a dot at (*dotcol, dotrow*) in whatever mode is selected by **DMODE** and in whatever color is selected by **DCOLOR** .

**DP** RESIDENT

( *--- addr* )

A user variable, the dictionary pointer, which contains the address of the next free memory above the dictionary. The value may be read by **HERE** and altered by **ALL0T** .

**DPL** RESIDENT

( *--- addr* )

A user variable containing the number of digits to the right of the decimal point on double integer input. It may also be used to hold output column location of a decimal point in user-generated formatting. The default value on single number input is -1.

**DR0 DR1 DR2** RESIDENT

( *---* )

Command to select disk drives by presetting **OFFSET** . The contents of **OFFSET** is added to the block number in **BLOCK** to allow for this selection. **OFFSET** is suppressed for error text so that it may always originate from drive 0.

**DRAW** SCR 63 -GRAPH

( *---* )

Sets **DMODE** equal to 0. This means that dots are plotted in the 'on' state.

**DRIVE** RESIDENT

( *n ---* )

Adjusts **OFFSET** so that the drive number on the stack becomes the first drive in the system.

**DROP** RESIDENT

( *n ---* )

Drop the top number from the stack.

<b>DSPLY</b>	SCR 69 -FILE
( --- )	
Assigns the attribute DISPLAY to the file pointed to by <b>PAB-ADDR</b> .	
<b>DSRLNK</b>	SCR 33 -SYNONYMS
( --- )	
Links a Forth program to any Device Service Routine (DSR) in ROM. Before this instruction may be used, a PAB must be set up in VDP RAM.	
<b>DTEST</b>	SCR 39 -COPY
( --- )	
Performs a non-destructive test of the disk in DR0 by attempting to read each Forth screen.	
<b>DTOG</b>	SCR 63 -GRAPH
( --- )	
Sets <b>DMODE</b> equal to 2. This means that each dot plotted takes on the opposite state as the dot currently at that location.	
<b>DUMP</b>	SCR 43 -DUMP
( <i>addr n</i> --- )	
Print the contents of <i>n</i> memory locations beginning at <i>addr</i> . Both addresses and contents are shown in hexadecimal notation. See <b>PAUSE</b> .	
<b>DUP</b>	RESIDENT
( <i>n</i> --- <i>n n</i> )	
Duplicates the value on the stack.	
<b>DXY</b>	SCR 59 -GRAPH
( <i>dotcol<sub>1</sub> dotrow<sub>1</sub> dotcol<sub>2</sub> dotrow<sub>2</sub></i> --- <i>n<sub>1</sub> n<sub>2</sub></i> )	
Places on the stack the square of the <i>x</i> distance <i>n<sub>1</sub></i> and the square of the <i>y</i> distance <i>n<sub>2</sub></i> between the points ( <i>dotcol<sub>1</sub>,dotrow<sub>1</sub></i> ) and ( <i>dotcol<sub>2</sub>,dotrow<sub>2</sub></i> ).	
<b>ECOUNT</b>	RESIDENT
( --- <i>addr</i> )	
A user variable that contains an error count. This is used to prevent error recursion.	
<b>ED@</b> ( <i>EDITOR1 Vocabulary</i> )	SCR 38 -EDITOR
( --- )	
Brings you back into the 40-column editor on the last Forth screen you edited. This screen is pointed to by <b>SCR</b> . Must be in Text mode.	

**ED@** (*EDITOR2 Vocabulary*) SCR 29 -64 SUPPORT

( --- )

Brings you back into the 64-column editor on the last Forth screen you edited. This screen is pointed to by **SCR**.

**EDIT** (*EDITOR1 Vocabulary*) SCR 38 -EDITOR

( *scr* --- )

Brings you into the 40-column editor on the specified Forth screen. Must be in Text mode.

**EDIT** (*EDITOR2 Vocabulary*) SCR 29 -64SUPPORT

( *scr* --- )

Brings you into the 64-column editor on the specified Forth screen.

**ELSE** RESIDENT

Compilation: (  $addr_1\ n_1$  ---  $addr_2\ n_2$  ) Runtime: ( --- )

Occurs within a colon-definition in the form:

**IF ... ELSE ... ENDIF**

At compile-time, **ELSE** emplaces **BRANCH**, reserving a branch offset and leaves the address  $addr_2$  and  $n_2$  for error testing. **ELSE** also resolves the pending forward from **IF** by calculating the offset from  $addr_1$  to **HERE** and storing it at  $addr_1$ .

At runtime, **ELSE** executes after the true part following **IF**. **ELSE** forces execution to skip over the following false part and resume execution after **ENDIF**. It has no stack effect.

**EMIT** RESIDENT

( *char* --- )

Transmit ASCII character *char* to the selected output device. **OUT** is incremented for each character output.

**EMIT8** RESIDENT

( *char* --- )

Transmit an 8-bit character *char* to the selected output device. **OUT** is incremented for each character output.

**EMPTY-BUFFERS** RESIDENT

( --- )

Mark all block buffers as empty, not necessarily affecting the contents. Updated blocks are not written to the disk. This is also an initialization procedure before first use of the disk.

**ENCLOSE**

RESIDENT

( *addr*<sub>1</sub> *char* --- *addr*<sub>1</sub> *n*<sub>1</sub> *n*<sub>2</sub> *n*<sub>3</sub> )

The text scanning primitive used by **WORD** . From the text address *addr*<sub>1</sub> and an ASCII-delimiting character *char*, is determined the byte offset *n*<sub>1</sub> to the first non-delimiter character, the offset *n*<sub>2</sub> to the delimiter after the text and the offset *n*<sub>3</sub> to the first character not included. This procedure will not process past an ASCII ‘null’ (0), treating it as an unconditional delimiter.

**END**

RESIDENT

( *flag* --- )

This is an alias or duplicate definition for **UNTIL** .

**ENDCASE**

RESIDENT

( *n* --- )

Terminates the **CASE** construct and, if actually executed at runtime because all intervening **OF ... ENDOF** clauses failed, removes the number *n* left on the stack. See **CASE** .

**ENDIF**

RESIDENT

Compilation: ( *addr* *n* --- )

Occurs in a colon-definition in the form:

**IF ... ENENDIF**

**IF ... ELSE ... ENENDIF**

At runtime, **ENDIF** serves only as the destination of a forward branch from **IF** or **ELSE** . It marks the conclusion of the conditional structure. **THEN** is another name for **ENDIF** . Both names are supported in fig-Forth. See also **IF** and **ELSE** .

At compile-time, **ENDIF** computes the forward branch offset from *addr* to **HERE** and stores it at *addr*. *n* is used for error tests.

**ENDOF**

RESIDENT

( --- )

Terminates the **OF** construct within the **CASE** construct. If executed at runtime, causes execution to proceed just beyond **ENDCASE** . See **OF** .

**ERASE**

RESIDENT

( *addr* *n* --- )

Clear *n* bytes of memory to zero starting at *addr*.

**ERROR**

RESIDENT

 $(n_1 \text{ --- } n_2 \ n_3)$ 

**ERROR** processes error notification and restarts the interpreter. **WARNING** is first examined. If **WARNING** < 1, **(ABORT)** is executed. The sole action of **(ABORT)** is to execute **ABORT**. This allows the user to (cautiously!) modify this behavior by redefining **(ABORT)**. **ABORT** clears the stacks and executes **QUIT**, which stops compilation and restarts the interpreter. If **WARNING** ≥ 0, **ERROR** leaves the contents of **IN**  $n_2$  and **BLK**  $n_3$  on the stack to assist in determining the location of the error. If **WARNING** > 0, **ERROR** prints the text of line  $n_1$ , relative to Forth screen 4 of drive 0. If **WARNING** = 0, **ERROR** prints  $n_1$  as an error number (as in a non-disk installation). The last thing **ERROR** does is to execute **QUIT**, which, as above, stops compilation and restarts the interpreter.

**EXECUTE**

RESIDENT

 $(cfa \text{ --- })$ 

Execute the definition whose code field address is on the stack. The code field address is also called the compilation address.

**EXP**

SCR 50 -FLOAT

 $(f_1 \text{ --- } f_2)$ 

Raises  $e$  to the power specified by the floating point number  $f_1$  on the stack and leaves the result  $f_2$  on the stack.

**EXPECT**

RESIDENT

 $(addr \ count \text{ --- })$ 

Transfer characters from the terminal to *addr* until **<ENTER>** or *count* characters have been received. One or more nulls are added at the end of the text.

**F!**

SCR 45 -FLOAT

 $(f \ addr \text{ --- })$ 

Stores a floating point number  $f$  into the 4 words (cells) beginning with the specified address.

**F\***

SCR 46 -FLOAT

 $(f_1 \ f_2 \text{ --- } f_3)$ 

Multiplies the top two floating point numbers on the stack and leaves the result on the stack.  $f_1 * f_2 = f_3$ .



<b>F+</b>	SCR 46 -FLOAT
$(f_1 f_2 \text{ --- } f_3)$	
Adds the top two floating point numbers on the stack and places the result on the stack. $f_1 * f_2 = f_3$ .	
<b>F-</b>	SCR 46 -FLOAT
$(f_1 f_2 \text{ --- } f_3)$	
Subtracts $f_2$ from $f_1$ and places the result on the stack ( $f_1 - f_2 = f_3$ ).	
<b>F-&gt;S</b>	SCR 46 -FLOAT
$(f \text{ --- } n)$	
Converts a floating point number $f$ on the parameter stack into a single precision number $n$ .	
<b>F-D"</b>	SCR 70 -FILE
$( \text{ --- } )$	
Expects a file descriptor ending with a " to follow. This instruction places the file descriptor in the PAB (Peripheral Access Block) pointed to by <b>PAB-ADDR</b> .	
<b>F.</b>	SCR 48 -FLOAT
$(f \text{ --- } )$	
Prints a floating point number in BASIC format to the output device.	
<b>F.R</b>	SCR 48 -FLOAT
$(f n \text{ --- } )$	
Prints the floating point number $f$ in BASIC format right justified in a field of width $n$ .	
<b>F/</b>	SCR 46 -FLOAT
$(f_1 f_2 \text{ --- } f_3)$	
Divides $f_1$ by $f_2$ and leaves the floating point quotient $f_3$ on the stack. $f_1 / f_2 = f_3$ .	
<b>F0&lt;</b>	SCR 49 -FLOAT
$(f \text{ --- } flag)$	
Compares the floating point number $f$ on the stack to 0. If it is less than 0, a true flag is left on the stack, else a false flag is left.	
<b>F0=</b>	SCR 49 -FLOAT
$(f \text{ --- } flag)$	
Compares the floating point number $f$ on the stack to 0. If it is equal to 0, a true flag is left on the stack, else a false flag is left.	

<b>F&lt;</b>	SCR 49 -FLOAT
$(f_1 f_2 \text{ --- } flag)$ Leaves a true flag if $f_1 < f_2$ , else leaves a false flag.	
<b>F=</b>	SCR 49 -FLOAT
$(f_1 f_2 \text{ --- } flag)$ Leaves a true flag if $f_1 = f_2$ , else leaves a false flag.	
<b>F&gt;</b>	SCR 49 -FLOAT
$(f_1 f_2 \text{ --- } flag)$ Leaves a flag if $f_1 > f_2$ , else leaves a false flag.	
<b>F@</b>	SCR 45 -FLOAT
$(addr \text{ --- } f)$ Retrieves the floating point contents $f$ of the given address (4 words) and places it on the stack.	
<b>FAC</b>	SCR 45 -FLOAT
$(\text{--- } addr)$ A constant which contains the address of the <b>FAC</b> register.	
<b>FAC-&gt;S</b>	SCR 46 -FLOAT
$(\text{--- } n)$ Converts a floating point number in <b>FAC</b> to a single precision number and places it on the parameter stack.	
<b>FAC&gt;</b>	SCR 45 -FLOAT
$(\text{--- } f)$ Brings a floating point number $f$ from <b>FAC</b> to the stack.	
<b>FAC&gt;ARG</b>	SCR 46 -FLOAT
$(\text{---})$ Moves a floating point number from <b>FAC</b> into <b>ARG</b> .	
<b>FADD</b>	SCR 45 -FLOAT
$(\text{---})$ Adds the floating point number in <b>FAC</b> to the floating point number in <b>ARG</b> and leaves the result in <b>FAC</b> .	

<b>FDIV</b>	SCR 45 -FLOAT
( --- )	
Divides the floating point number in <b>FAC</b> by the floating point number in <b>ARG</b> leaving the quotient in <b>FAC</b> .	
<b>FDROP</b>	SCR 45 -FLOAT
( <i>f</i> --- )	
Drops the top floating point number <i>f</i> from the stack.	
<b>FDUP</b>	SCR 45 -FLOAT
( <i>f</i> --- <i>ff</i> )	
Duplicates the top floating point number <i>f</i> on the stack.	
<b>FENCE</b>	RESIDENT
( --- <i>addr</i> )	
A user variable containing an address (usually the NFA of a Forth word) below which <b>FORGET</b> ing is trapped. To <b>FORGET</b> below this point the user must alter the contents of <b>FENCE</b> . It is possible to set the value of <b>FENCE</b> to a value that is actually less than the address of the end of the last word in the core dictionary ( <b>TASK</b> ) such that <b>UNFORGETTABLE</b> [ <i>sic</i> ] will report false; however, <b>FORGET</b> will still trap that error.	
<b>FF.</b>	SCR 48 -FLOAT
( <i>f</i> <i>n</i> <sub>1</sub> <i>n</i> <sub>2</sub> --- )	
Prints the floating point number <i>f</i> with <i>n</i> <sub>2</sub> digits following the decimal point and a maximum of <i>n</i> <sub>1</sub> digits.	
<b>FF.R</b>	SCR 48 -FLOAT
( <i>f</i> <i>n</i> <sub>1</sub> <i>n</i> <sub>2</sub> <i>n</i> <sub>3</sub> --- )	
Prints the floating point number <i>f</i> , with <i>n</i> <sub>2</sub> digits following the decimal point, right justified in a field of width <i>n</i> <sub>3</sub> with a maximum of <i>n</i> <sub>1</sub> digits.	
<b>FILE</b>	SCR 68 -FILE
( <i>vaddr</i> <sub>1</sub> <i>addr</i> <i>vaddr</i> <sub>2</sub> --- )	
A defining word which permits you to create a word by which a file will be known. You must place on the stack the <b>PAB-ADDR</b> , <b>PAB-BUF</b> and <b>PAB-VBUF</b> addresses you wish to be associated with the file.	
Used in the form:	
<b><i>vaddr</i><sub>1</sub> <i>addr</i> <i>vaddr</i><sub>2</sub> FILE cccc</b>	
When <b>cccc</b> executes, <b>PAB-ADDR</b> , <b>PAB-BUF</b> and <b>PAB-VBUF</b> are set to <i>vaddr</i> <sub>1</sub> , <i>addr</i> and <i>vaddr</i> <sub>2</sub> , respectively.	

<b>FILL</b>	RESIDENT
( <i>addr count b</i> --- )	
Fill memory beginning at <i>addr</i> with <i>count</i> bytes of byte <i>b</i> .	
<b>FIRST</b>	RESIDENT
( --- <i>addr</i> )	
A constant that leaves the address of the first (lowest) block buffer.	
<b>FIRST\$</b>	RESIDENT
( --- <i>addr</i> )	
A user variable which contains the first byte of the disk buffer area.	
<b>FLD</b>	RESIDENT
( --- <i>addr</i> )	
A user variable for control of number output field width. Presently unused in fig-Forth and TI Forth.	
<b>FLERR</b>	SCR 49 -FLOAT
( --- <i>n</i> )	
Returns on the stack the contents <i>n</i> of the floating point status register ( <b>8354h</b> ).	
<b>FLUSH</b>	RESIDENT
( --- )	
Writes to disk all disk buffers that have been marked as updated.	
<b>FMUL</b>	SCR 45 -FLOAT
( --- )	
Multiplies the floating point number in <b>FAC</b> with the floating point number in <b>ARG</b> leaving the product in <b>FAC</b> .	
<b>FORGET</b>	RESIDENT
( --- )	
Executed in the form:	
<b>FORGET cccc</b>	
Deletes the definition named <b>cccc</b> from the dictionary along with all dictionary entries physically following it.	
<b>FORGET</b> first checks the LFA of <b>cccc</b> to see if it is lower than the address in <b>FENCE</b> . If it is not, <b>FORGET</b> then checks whether it is lower than the address of the last byte of the core dictionary. If it is not lower than either of these addresses, <b>FORGET</b> updates <b>HERE</b> to the LFA of <b>cccc</b> , effectively deleting the desired part of the dictionary. Otherwise, an appropriate error message is displayed.	

**FORMAT-DISK**

SCR 33 -SYNONYMS

 $(n \text{ ---})$ 

Initializes the disk in DR0 ( $n = 0$ ), DR1 ( $n = 1$ ) or DR2 ( $n = 2$ ) for use with the Forth system. **Caution:** All data on the disk will be destroyed. Also, disks initialized by the Disk Manager may be used without any changes. Drive number  $n$  must be 0, 1 or 2.

**FORTH**

RESIDENT

 $(\text{---})$ 

The name of the primary vocabulary. Execution makes **FORTH** the **CONTEXT** vocabulary. Until additional user vocabularies are defined, new user definitions become a part of **FORTH** because it is at that point also the **CURRENT** vocabulary. **FORTH** is immediate, so it will execute during the creation of a colon definition to select this vocabulary at compile time.

**FORTH-COPY**

SCR 39 -COPY

 $(\text{---})$ 

Copies the entire disk in DR1 onto the disk in DR0.

**FORTH\_LINK**

RESIDENT

 $(\text{--- addr})$ 

A user variable used for vocabulary linkage.

**FOVER**

SCR 45 -FLOAT

 $(f_1 f_2 \text{ --- } f_1 f_2 f_1)$ 

Copies the second floating point number on the stack to the top of the stack.

**FRND**

SCR 46 FLOAT

 $(\text{--- } f)$ 

Generates a pseudo-random floating point number greater than or equal to 0 and less than 1.

**FSUB**

SCR 45 -FLOAT

 $(\text{---})$ 

Subtracts the floating point number in **ARG** from the number in **FAC** and leaves the result in **FAC**.

**FSWAP**

SCR 45 -FLOAT

 $(f_1 f_2 \text{ --- } f_2 f_1)$ 

Swaps the top two floating point numbers on the stack.

<b>FXD</b>	SCR 68 -FILE
( --- )	
Assigns the attribute <b>FIXED</b> to the file whose PAB (Peripheral Access Block) is pointed to by <b>PAB-ADDR</b> .	
<b>GCHAR</b>	SCR 58 -GRAPH
( <i>col row</i> --- <i>char</i> )	
Returns on the stack the ASCII code <i>char</i> of the character currently at ( <i>col,row</i> ). <i>Note:</i> Rows and columns are numbered from 0.	
<b>GET-FLAG</b>	SCR 68 -FILE
( --- <i>b</i> )	
Retrieves the flag byte <i>b</i> from the current PAB and places it on the stack.	
<b>GOTOXY</b>	RESIDENT
( <i>col row</i> --- )	
Places the cursor at the designated column <i>col</i> and row <i>row</i> position. <i>Note:</i> Rows and columns are numbered from 0.	
<b>GPLLNK</b>	SCR 33 -SYNONYMS
( <i>addr</i> --- )	
Links a Forth program to the Graphics Programming Language (GPL) routine located at the given address.	
<b>GRAPHICS</b>	SCR 52 -GRAPH1
( --- )	
Converts from present display screen mode into standard Graphics mode configurations.	
<b>GRAPHICS2</b>	SCR 54 -GRAPH2
( --- )	
Converts from present Forth screen mode into standard Graphics2 mode configuration.	
<b>HCHAR</b>	SCR 57 -GRAPH
( <i>col row count char</i> --- )	
Prints a horizontal stream of a specified character <i>char</i> beginning at ( <i>col,row</i> ) and having a length <i>char</i> . <i>Note:</i> Rows and columns are numbered from 0.	
<b>HERE</b>	RESIDENT
( --- <i>addr</i> )	
Leave the address of the next available dictionary location.	

<b>HEX</b>	RESIDENT
( --- )	
Set the numeric conversion base to sixteen (hexadecimal).	
<b>HLD</b>	RESIDENT
( --- <i>addr</i> )	
A user variable that holds the address of the latest character of text during numeric output conversion.	
<b>HOLD</b>	RESIDENT
( <i>char</i> --- )	
Used between <# and #> to insert an ASCII character into a pictured numeric output string, <i>e.g.</i> , <b>2E HOLD</b> will place a decimal point.	
<b>HONK</b>	SCR 60 -GRAPH
( --- )	
Produces the sound associated with incorrect input.	
<b>I</b>	RESIDENT
( --- <i>n</i> )	
Used within a <b>DO</b> loop to copy the loop index to the stack. Other use is implementation dependent. <b>I</b> is a synonym for <b>R</b> .	
<b>ID.</b>	RESIDENT
( <i>nfa</i> --- )	
Print a definition's name from its name field address <i>nfa</i> .	
<b>IF</b>	RESIDENT
Compilation: ( --- <i>addr n</i> ) Runtime: ( <i>flag</i> --- )	
Occurs in a colon definition in form:	
<b>IF (true part) ... ENDIF</b>	
<b>IF (true part) ... ELSE (false part) ... ENDIF</b>	
At compile time, <b>IF</b> compiles <b>0BRANCH</b> and reserves space for an offset at <i>addr</i> ; <i>addr</i> and <i>n</i> are used later for resolution of the offset and error testing.	
At runtime, <b>IF</b> selects execution based on a Boolean flag. If <i>flag</i> is <i>true</i> (non-zero), execution continues ahead through the true part. If <i>flag</i> is <i>false</i> (zero), execution skips to just after <b>ELSE</b> to execute the false part. After either part, execution resumes after <b>ENDIF</b> . <b>ELSE</b> and its false part are optional. If missing, false execution skips to just after <b>ENDIF</b> .	

**IMMEDIATE**

RESIDENT

( --- )

Mark the most recently made definition so that when encountered at compile time, it will be executed rather than being compiled. *i.e.*, the precedence bit in its header is set. This method allows definitions to handle unusual compiling situations rather than build them into the fundamental compiler. The user may force compilation of an immediate definition by preceding it with [**COMPILE**] .

**IN**

RESIDENT

( --- *addr* )

A user variable containing the byte offset within the current input text buffer (terminal or disk) from which the next text will be accepted. **WORD** uses and moves the value of **IN** .

**INDEX**

SCR 73 -PRINT

(  $n_1$   $n_2$  --- )

Prints to the terminal a list of the line #0 comments from Forth screen  $n_1$  through Forth screen  $n_2$ . See **PAUSE** .

**INPT**

SCR 69 -FILE

( --- )

Assigns the attribute INPUT to the file whose PAB is pointed to by **PAB-ADDR** .

**INT**

SCR 50 -FLOAT

(  $f_1$  ---  $f_2$  )

Leaves the integer portion of a floating point number on the stack.

**INTERPRET**

RESIDENT

( --- )

The outer text interpreter, which sequentially executes or compiles text from the input stream (terminal or disk) depending on **STATE** . If the word name cannot be found after a search of **CONTEXT** and then **CURRENT** , it is converted into a number according to the current base. That also failing, an error message echoing the name with a “?” will be given. Text input will be taken according to the convention for **WORD** . If a decimal point is found as part of a number, a double number value will be left. The decimal point has no other purpose than to force this action. See **NUMBER** .

**INTLNK**

RESIDENT

( --- *addr* )

A user variable which is a pointer to the Interrupt Service linkage.



<b>INTRNL</b>	SCR 69 -FILE
( --- )	
Assigns the attribute INTERNAL to the file whose PAB is pointed to by <b>PAB-ADDR</b> .	
<b>ISR</b>	RESIDENT
( --- <i>addr</i> )	
A user variable that initially contains the address of the interrupt service linkage code to install an Interrupt Service Routine. The user must modify <b>ISR</b> to contain the CFA of the routine to be executed each 1/60 second. Next, the contents of <b>83C4h</b> must be modified to point to this address. Note that the interrupt service linkage code address is also available in <b>INTLNK</b> .	
<b>J</b>	RESIDENT
( --- <i>n</i> )	
Copies the loop index of the next outer loop to the stack.	
<b>JOYST</b>	SCR 60 -GRAPH
( <i>n</i> <sub>1</sub> --- <i>char</i> <i>n</i> <sub>2</sub> <i>n</i> <sub>3</sub> )	
Allows you to accept input from joystick #1 and the left side of the keyboard ( <i>n</i> <sub>1</sub> = 1) or from joystick #2 and the right side of the keyboard ( <i>n</i> <sub>1</sub> = 2). Values returned are the character code <i>char</i> of the key pressed, the <i>x</i> status <i>n</i> <sub>2</sub> and the <i>y</i> status <i>n</i> <sub>3</sub> .	
<b>KEY</b>	RESIDENT
( --- <i>char</i> )	
Leave the ASCII value of the next terminal key struck.	
<b>KEY8</b>	RESIDENT
( --- <i>char</i> )	
Leave the 8-bit value of the next terminal key struck.	
<b>L/SCR</b>	RESIDENT
( --- <i>n</i> )	
Returns on the stack the number of lines per Forth screen.	
<b>LATEST</b>	RESIDENT
( --- <i>nfa</i> )	
Leave the name field address <i>nfa</i> of the most recently defined word in the <b>CURRENT</b> vocabulary. At compile time, this “latest” word will be the most recently compiled word.	

**LD**

SCR 71 -FILE

( *n* --- )

The file I/O process to load a program file from a disk into VDP RAM. The parameter *n* specifies the maximum number of bytes to be loaded and is usually the size of the file on disk. The file's PAB must be set up and be the current PAB, to which **PAB-ADDR** points, before executing this word.

**LDCR**

SCR 88

( *n*<sub>1</sub> *n*<sub>2</sub> *addr* --- )

Performs a TMS9900 LDCR instruction. The CRU base address *addr* will be shifted left one bit and stored in workspace register R12 prior to executing the TMS9900 LDCR instruction. The value *n*<sub>1</sub> is transferred to the CRU with a field width of *n*<sub>2</sub> bits.

**LEAVE**

RESIDENT

( --- )

Force termination of a **DO** loop at the next opportunity by setting the loop limit equal to the current value of the index. The index itself remains unchanged, and the execution proceeds normally until **LOOP** or **+LOOP** is encountered.

**LFA**

RESIDENT

( *pfa* --- *lfa* )

Convert the parameter field address *pfa* of a dictionary definition to its link field address *lfa*.

**LIMIT**

RESIDENT

( --- *addr* )

A constant which leaves the address *addr* just above the highest memory available for a disk buffer.

**LIMIT\$**

RESIDENT

( --- *addr* )

A user variable that contains the address just above the highest memory available for a disk buffer. The address of **LIMIT\$** is left on the stack.

**LINE**

SCR 64 -GRAPH

( *dotcol*<sub>1</sub> *dotrow*<sub>1</sub> *dotcol*<sub>2</sub> *dotrow*<sub>2</sub> --- )

The high resolution graphics routine which plots a line from (*dotcol*<sub>1</sub>,*dotrow*<sub>1</sub>) to (*dotcol*<sub>2</sub>,*dotrow*<sub>2</sub>). **DCOLOR** and **DMODE** must be set before this instruction is used.

<b>LIST</b>	RESIDENT
( <i>scr</i> --- )	
Lists the specified Forth screen to the output device. See <b>PAUSE</b> .	
<b>LIT</b>	RESIDENT
( --- <i>n</i> )	
Within a colon-definition, <b>LIT</b> is automatically compiled before each 16-bit literal number encountered in input text. Later execution of <b>LIT</b> causes the contents of the next dictionary address to be pushed to the stack.	
<b>LITERAL</b>	RESIDENT
Compilation: ( <i>n</i> --- ) Runtime: ( --- <i>n</i> ) Interpretation: ( --- )	
During compilation, compiles the stack value <i>n</i> as a 16-bit literal. This will execute during a colon definition. The intended use is:	
: <b>xxx</b> [ <i>calculation</i> ] <b>LITERAL</b> ;	
Compilation is suspended for the compile-time calculation of a value. Compilation is resumed and <b>LITERAL</b> compiles this value.	
At runtime, <i>n</i> is pushed to the stack. Interpretation of <b>LITERAL</b> does nothing, unlike other compiling words.	
<b>LOAD</b>	RESIDENT
( <i>n</i> --- )	
Begin interpretation of Forth screen <i>n</i> . Loading will terminate at the end of the Forth screen or at <b>;S</b> . See <b>;S</b> and <b>--&gt;</b> .	
<b>LOG</b>	SCR 50 -FLOAT
( $f_1$ --- $f_2$ )	
The floating point operation which returns the natural logarithm $f_1$ of the floating point number $f_2$ on the stack.	
<b>LOOP</b>	RESIDENT
Compilation: ( <i>addr n</i> --- )	
Occurs in a colon definition in the form:	
<b>DO</b> ... <b>LOOP</b>	
At runtime, <b>LOOP</b> selectively controls branching back to the corresponding <b>DO</b> based on the loop index and limit. The loop index is incremented by one and compared to the limit. The branch back to <b>DO</b> occurs until the index equals or exceeds the limit. At that time, the parameters are discarded and execution continues ahead.	
At compile time, <b>LOOP</b> compiles ( <b>LOOP</b> ) and uses <i>addr</i> to calculate an offset to <b>DO</b> . <i>n</i> is used for error testing.	

<b>M*</b>	RESIDENT
$( n_1 n_2 \text{ --- } d )$	
A mixed magnitude math operation that leaves the double number signed product $d$ of two signed numbers, $n_1$ and $n_2$ .	
<b>M/</b>	RESIDENT
$( d n_1 \text{ --- } n_2 n_3 )$	
A mixed magnitude math operator that leaves the signed remainder $n_2$ and signed quotient $n_3$ , from a double number dividend $d$ and divisor $n_1$ . The remainder takes its sign from the dividend.	
<b>M/MOD</b>	RESIDENT
$( ud_1 u_2 \text{ --- } u_3 ud_4 )$	
An unsigned mixed magnitude math operation that leaves an unsigned double quotient $ud_4$ and a single remainder $u_3$ , from a double dividend $ud_1$ and a single divisor $u_2$ .	
<b>MAGNIFY</b>	SCR 60 -GRAPH
$( n_1 \text{ --- } )$	
Alters the sprite magnification factor to be $n_1$ . The value of $n_1$ must be 0, 1, 2 or 3.	
<b>MAX</b>	RESIDENT
$( n_1 n_2 \text{ --- } n_3 )$	
Leave the greater $n_3$ of the two numbers, $n_1$ and $n_2$ .	
<b>MCHAR</b>	SCR 62 -GRAPH
$( n \text{ col row --- } )$	
Places a square of color $n$ at $( \text{col}, \text{row} )$ . Used in multicolor mode.	
<b>MENU</b>	SCR 20 BOOT SCR
$( \text{ --- } )$	
Displays the available Load Options.	
<b>MESSAGE</b>	RESIDENT
$( n \text{ --- } )$	
Print on the selected output device the text of line $n$ relative to screen 4 of drive 0. The value of $n$ may be positive or negative. <b>MESSAGE</b> may be used to print incidental text such as report headers. If <b>WARNING</b> = 0, the message will simply be printed as a number (disk unavailable).	

<b>MIN</b>	RESIDENT
( $n_1$ $n_2$ --- $n_3$ )	
Leave the smaller $n_3$ of the two numbers ( $n_1$ and $n_2$ ).	
<b>MINIT</b>	SCR 62 -GRAPH
( --- )	
Initializes the monitor screen for use with <b>MCHAR</b> .	
<b>MINUS</b>	RESIDENT
( $n_1$ --- $n_2$ )	
Leave the two's complement $n_2$ of a number $n_1$ .	
<b>MOD</b>	RESIDENT
( $n_1$ $n_2$ --- <i>rem</i> )	
Leave the remainder <i>rem</i> of $n_1/n_2$ , with the same sign as $n_1$ .	
<b>MON</b>	SCR 33 -SYNONYMS
( --- )	
Exit to the TI 99/4A color bar display screen.	
<b>MOTION</b>	SCR 59 -GRAPH
( $n_1$ $n_2$ <i>spr</i> --- )	
Assigns a horizontal $n_1$ and vertical $n_2$ velocity to the specified sprite <i>spr</i> .	
<b>MOVE</b>	RESIDENT
( <i>addr</i> <sub>1</sub> <i>addr</i> <sub>2</sub> $n$ --- )	
Move the contents of $n$ memory cells (16-bit contents) beginning at <i>addr</i> <sub>1</sub> into $n$ cells beginning at <i>addr</i> <sub>2</sub> . The contents of <i>addr</i> <sub>1</sub> is moved first.	
<b>MULTI</b>	SCR 53 -GRAPH
( --- )	
Converts from present display screen mode into standard Multicolor mode configuration.	
<b>MYSELF</b>	RESIDENT
( --- )	
Used in a colon definition. Places the code field address (CFA) of a word into its own definition. This permits recursion.	

<b>NFA</b>	RESIDENT
( <i>pfa</i> --- <i>nfa</i> )	
Convert the parameter field address <i>pfa</i> of a definition to its name field address <i>nfa</i> .	
<b>NOP</b>	RESIDENT
( --- )	
A do-nothing instruction. <b>NOP</b> is useful for patching as in assembly code.	
<b>NUMBER</b>	RESIDENT
( <i>addr</i> --- <i>d</i> )	
Convert a character string left at <i>addr</i> with the character count in the first byte, to a signed double number <i>d</i> , using the current numeric base. If a decimal point is encountered in the text, its position will be given in <b>DPL</b> , but no other effect occurs. If numeric conversion is not possible, an error message will be given.	
<b>OF</b>	RESIDENT
( <i>n</i> --- <i>n</i>   )	
Initiates the <b>OF ... ENDOF</b> construct inside of the <b>CASE</b> construct. <i>n</i> is compared to the value which was on top of the stack when <b>CASE</b> was executed. If the numbers are identical, the words between <b>OF</b> and <b>ENDOF</b> will be executed. Otherwise, <i>n</i> is put back on the stack. See <b>CASE</b> .	
<b>OFFSET</b>	RESIDENT
( --- <i>addr</i> )	
A user variable which may contain a block offset to disk drives. The contents of <b>OFFSET</b> is added to the stack number by <b>BLOCK</b> . Messages issued by <b>MESSAGE</b> are independent of <b>OFFSET</b> . See <b>BLOCK</b> , <b>DR0</b> and <b>MESSAGE</b> .	
<b>OPN</b>	SCR 71 -FILE
( --- )	
Opens the file whose PAB is pointed to by <b>PAB-ADDR</b> .	
<b>OR</b>	RESIDENT
( <i>n</i> <sub>1</sub> <i>n</i> <sub>2</sub> --- <i>n</i> <sub>3</sub> )	
Leave the bit-wise logical OR <i>n</i> <sub>3</sub> of two 16-bit values, <i>n</i> <sub>1</sub> and <i>n</i> <sub>2</sub> .	
<b>OUT</b>	RESIDENT
( --- <i>addr</i> )	
A user variable that contains a value incremented by <b>EMIT</b> and <b>EMIT8</b> . The user may alter and examine <b>OUT</b> to control display formatting.	

<b>OUTPT</b>	SCR 69 -FILE
( --- )	
Assigns the attribute OUTPUT to the file whose PAB is pointed to by <b>PAB-ADDR</b> .	
<b>OVER</b>	RESIDENT
( $n_1$ $n_2$ --- $n_1$ $n_2$ $n_1$ )	
Copy the second stack value $n_1$ to the top of the stack.	
<b>PAB-ADDR</b>	SCR 68 -FILE
( --- <i>vaddr</i> )	
A variable containing the VDP address of the first byte of the current PAB (Peripheral Access Block).	
<b>PAB-BUF</b>	SCR 68 -FILE
( --- <i>addr</i> )	
A variable which holds the address of the area in CPU RAM used as the source or destination of the data to be transferred to/from a file. This is a file I/O word.	
<b>PAB-VBUF</b>	SCR 68 -FILE
( --- <i>vaddr</i> )	
A variable pointing to a VDP RAM buffer which serves as a temporary buffer when transferring data to/from a file. The VDP address stored in <b>PAB-VBUFF</b> is also stored in the file's PAB.	
<b>PABS</b>	RESIDENT
( --- <i>vaddr</i> )	
A user variable which points to a region in VDP RAM, which has been set aside for creating PABs.	
<b>PAD</b>	RESIDENT
( --- <i>addr</i> )	
Leave the address of the text output buffer, which is a fixed offset (68 bytes in TI Forth) above <b>HERE</b> . Every time <b>HERE</b> changes, <b>PAD</b> is updated.	
<b>PAUSE</b>	RESIDENT
( --- <i>flag</i> )	
The words <b>LIST</b> , <b>INDEX</b> , <b>DUMP</b> and <b>VLIST</b> all call the word <b>PAUSE</b> . <b>PAUSE</b> allows the user to temporarily halt the output by pressing any key. Pressing another key will allow continuation. To exit one of these routines prematurely, press <b>&lt;BREAK&gt;</b> ( <b>&lt;CLEAR&gt;</b> or <b>&lt;FCTN+<sub>4</sub>&gt;</b> ).	

<b>PDT</b>	SCR 57 -GRAPH
( --- <i>vaddr</i> )	
A constant which contains the VDP address of the Pattern Descriptor Table. Default value is <b>800h</b> .	
<b>PFA</b>	RESIDENT
( <i>nfa</i> --- <i>pfa</i> )	
Convert the name field address <i>nfa</i> of a compiled definition to its parameter field address <i>pfa</i> .	
<b>PI</b>	SCR 50 -FLOAT
( --- <i>f</i> )	
A floating point approximation of $\pi$ to 13 significant figures. (3.141592653590)	
<b>PREV</b>	RESIDENT
( --- <i>addr</i> )	
A user variable containing the address of the disk buffer most recently referenced. The <b>UPDATE</b> command marks this buffer to be later written to disk.	
<b>PUT-FLAG</b>	SCR 68 -FILE
( <i>b</i> --- )	
Writes the flag byte <i>b</i> into the appropriate PAB referenced by <b>PAB-ADDR</b> .	
<b>QUERY</b>	RESIDENT
( --- )	
Input 80 characters of text (or until <b>&lt;ENTER&gt;</b> is pressed) from the operator's terminal. Text is positioned at the address contained in <b>TIB</b> with <b>IN</b> set to 0.	
<b>QUIT</b>	RESIDENT
( --- )	
Clear the return stack, stop compilation and return control to the operator's terminal. No message is given, including the usual "ok".	
<b>R</b>	RESIDENT
( --- <i>n</i> ) ( R: <i>n</i> --- <i>n</i> )	
Copy the top of the return stack to the parameter stack.	
<b>R#</b>	RESIDENT
( --- <i>addr</i> )	
A user variable which may contain the location of an editing cursor or other file-related function.	



<b>R-&gt;BASE</b>	RESIDENT
( --- ) ( R: <i>n</i> --- )	
Restore the current base from the return stack. See <b>BASE-&gt;R</b> .	
<b>R/W</b>	RESIDENT
( <i>addr</i> <i>n</i> <sub>1</sub> <i>flag</i> --- )	
The fig-Forth standard disk read/write linkage. The source or destination block buffer address is <i>addr</i> , <i>n</i> <sub>1</sub> is the sequential number of the referenced block and <i>flag</i> indicates whether the operation is write ( <i>flag</i> = 0) or read ( <i>flag</i> = 1). <b>R/W</b> determines the location on mass storage, performs the read/write and error checking.	
<b>R0</b>	RESIDENT
( --- <i>addr</i> )	
A user variable containing the initial location of the return stack. Pronounced "r zero". See <b>RP!</b> .	
<b>R&gt;</b>	RESIDENT
( --- <i>n</i> ) ( R: <i>n</i> --- )	
Remove the top value from the return stack and leave it on the parameter stack. See <b>&gt;R</b> and <b>R</b> .	
<b>RANDOMIZE</b>	SCR 33 -SYNONYMS
( --- )	
Creates an unpredictable seed for the random number generator.	
<b>RD</b>	SCR 71 -FILE
( --- <i>count</i> )	
The file I/O instruction that reads from the current PAB. This instruction uses <b>PAB-BUF</b> and <b>PAB-VBUF</b> .	
<b>RDISK</b>	RESIDENT
( <i>addr</i> <i>n</i> <sub>1</sub> <i>n</i> <sub>2</sub> --- <i>n</i> <sub>3</sub> )	
The primitive routine that performs disk reads. The address where the block is to be written in CPU RAM is <i>addr</i> . The block number <i>n</i> <sub>1</sub> , the number of bytes per block is <i>n</i> <sub>2</sub> and <i>n</i> <sub>3</sub> is the returned error code.	
<b>REC-LEN</b>	SCR 69 -FILE
( <i>b</i> --- )	
Stores the length <i>b</i> of the record for the upcoming write into the appropriate byte in the current PAB.	

**REC-NO**

SCR 69 -FILE

 $( n \text{ ---} )$ Writes a record number  $n$  into the appropriate location in the current PAB.**REPEAT**

RESIDENT

Compilation:  $( addr \ n \text{ ---} )$ 

Used within a colon-definition in the form:

**BEGIN ... WHILE ... REPEAT**At runtime, **REPEAT** forces an unconditional branch back to just after the corresponding **BEGIN**.At compile-time, **REPEAT** compiles **BRANCH** and the offset from **HERE** to  $addr$ .  $n$  is used for error testing.**RLTV**

SCR 69 -FILE

 $( \text{---} )$ Assigns the attribute RELATIVE to the file whose PAB is pointed to by **PAB-ADDR**.**RND**

SCR 33 -SYNONYMS

 $( n_1 \text{ --- } n_2 )$ Generates a positive random integer  $n_2$  greater than or equal to 0 and less than  $n_1$ .**RNDW**

SCR 33 -SYNONYMS

 $( \text{--- } n )$ 

Generates a random word. The value of the word may be positive or negative depending on whether the sign bit is set.

**ROT**

RESIDENT

 $( n_1 \ n_2 \ n_3 \text{ --- } n_2 \ n_3 \ n_1 )$ Rotate the top three values on the stack, bringing the third  $n_1$  to the top.**RP!**

RESIDENT

 $( \text{---} )$ A procedure to initialize the return stack pointer from user variable **R0**.**RSTR**

SCR 71 -FILE

 $( n \text{ ---} )$ Restores the file whose PAB is pointed to by the current PAB to the specified record number  $n$ .

<b>S-&gt;D</b>	RESIDENT
( <i>n</i> --- <i>d</i> )	
Sign-extend a single number <i>n</i> to form a double number <i>d</i> .	
<b>S-&gt;F</b>	SCR 46 -FLOAT
( <i>n</i> --- <i>f</i> )	
Converts a single-precision number <i>n</i> on the stack to a floating point number <i>f</i> .	
<b>S-&gt;FAC</b>	SCR 46 -FLOAT
( <i>n</i> --- )	
Takes a single-precision number <i>n</i> from the stack, converts it to floating point, and leaves it in FAC.	
<b>S0</b>	RESIDENT
( --- <i>addr</i> )	
User variable that points to the base of the parameter stack. Pronounced “s zero”. See <b>SP!</b> .	
<b>SATR</b>	SCR 57 -GRAPH
( --- <i>vaddr</i> )	
A constant whose value <i>vaddr</i> is the VDP address of the Sprite Attribute List. Default value is <b>300h</b> .	
<b>SB0</b>	SCR 88 -CRU
( <i>addr</i> --- )	
This word expects to find on the stack the CRU address <i>addr</i> of the bit to be set to 1. <b>SB0</b> will put this address into workspace register R12, shift it left (double it) and execute <b>0 SB0</b> , to effect setting the bit. See CRU documentation in the <i>Editor/Assembler Manual</i> for more information.	
<b>SBZ</b>	SCR 88 -CRU
( <i>addr</i> --- )	
This word expects to find on the stack the CRU address <i>addr</i> of the bit to be reset to 0. <b>SBZ</b> will put this address into workspace register R12, shift it left (double it) and execute <b>0 SBZ</b> , to effect resetting the bit. See CRU documentation in the <i>Editor/Assembler Manual</i> for more information.	
<b>SCOPY</b>	SCR 39 -COPY
( <i>scr</i> <sub>1</sub> <i>scr</i> <sub>2</sub> --- )	
Copies the source Forth screen <i>scr</i> <sub>1</sub> to the destination Forth screen <i>scr</i> <sub>2</sub> on disk. Does not destroy the source screen.	

**SCR** RESIDENT

( --- *addr* )

A user variable containing the Forth screen number most recently referenced by **LIST** or **EDIT**.

**SCREEN** SCR 58 -GRAPH

( *n* --- )

Changes the display screen color to the color specified *n*. The foreground (FG) and background (BG) screen colors must be placed in the low-order byte of *n*, with FG the high-order 4 bits and BG the low-order 4 bits, *e.g.*, *n* = 27 (**1Bh**) for black on light yellow.

**SCRN\_END** RESIDENT

( --- *addr* )

A user variable containing the address *addr* of the byte immediately following the last byte of the display screen image table to be used as the logical display screen.

**SCRN\_START** RESIDENT

( --- *addr* )

A user variable containing the address *addr* of the first byte of the display screen image table to be used as the logical display screen.

**SCRN\_WIDTH** RESIDENT

( --- *addr* )

A user variable which contains the number of characters that will fit across the display screen. (32 or 40) Used by the display screen scroller.

**SCRATCH** SCR 71 -FILE

( *n* --- )

Removes the specified record *n* from the RELATIVE file whose PAB is pointed to by **PAB-ADDR**. [*Editor's Note:* This word should *never* be used. TI never implemented this operation for files. It will *always* result in a file I/O error message.]

**SEED** SCR 33 -SYNONYMS

( *n* --- )

Places a new seed *n* into the random number generator.

**SET-PAB** SCR 68 -FILE

( --- )

This instruction assumes that **PAB-ADDR** is set. It then zeroes out the PAB (Peripheral Access Block) pointed to by **PAB-ADDR** and places the contents of **PAB-VBUF** into the appropriate word of the PAB. This initializes the PAB.

<b>SETFL</b>	SCR 45 -FLOAT
$(f_1 f_2 \text{ --- } )$ Performs <b>&gt;FAC</b> on $f_2$ and <b>&gt;ARG</b> on $f_1$ .	
<b>SIGN</b>	RESIDENT
$( n d \text{ --- } d )$ Stores a minus sign (ASCII 45 or <b>2Dh</b> ) at the current location in a converted numeric output string in the text output buffer if $n$ is negative. At the time $n$ is evaluated, it is discarded; but, double number $d$ is maintained for continued conversion until <b>#&gt;</b> removes it from the stack. Must be used between <b>&lt;#</b> and <b>#&gt;</b> . Using <b>SIGN</b> implies that $d$ can be negative, which means that $d$ should be used to produce $n$ . You should then replace $d$ with its absolute value ( $ d $ ) on the stack by using <b>DABS</b> . This can be done by pushing $d$ to the stack and executing <b>SWAP OVER DABS</b> : $( d \text{ --- } n  d  )$ prior to <b>&lt;# ... SIGN ... #&gt;</b> .	
<b>SIN</b>	SCR 50 -FLOAT
$(f_1 \text{ --- } f_2 )$ Finds the SIN $f_2$ of the floating point number $f_1$ on the stack and leaves the result $f_2$ on the stack.	
<b>SLA</b>	RESIDENT
$( n_1 \text{ count --- } n_2 )$ Arithmetically shifts the number $n_1$ on the stack $\text{count}$ bits to the left, leaving the result $n_2$ on the stack. Shifting by $\text{count}$ will be modulo 16 except when $\text{count} = 0$ , which causes 16 bits to be shifted. To create a word which does not perform a 16-bit shift when $\text{count}$ is zero, use the following definition for the same stack contents: <div style="text-align: center;"> <b>: SLA0 -DUP IF SLA ENDIF ;</b> </div>	
<b>SLIT</b>	SCR 20 BOOT SCR
$( \text{ --- } \text{addr} )$ <b>SLIT</b> is similar to <b>LIT</b> but acts on strings instead of numbers. <b>SLIT</b> places the address $\text{addr}$ of the string following it on the stack. It modifies the top of the return stack to point to just after the string.	
<b>SMASH</b>	SCR 65 -64SUPPORT
$( \text{addr}_1 \text{ count}_1 n \text{ --- } \text{addr}_2 \text{ vaddr count}_2 )$ The assembly code routine that formats a line of tiny characters. It expects the address $\text{addr}_1$ of the line in memory, the number $\text{count}_1$ of characters per line, and the line number $n$ to which it is to be written. It returns on the stack the line buffer address $\text{addr}_2$ , a VDP address $\text{vaddr}$ , and a character count $\text{count}_2$ . See <b>CLIST</b> and <b>CLINE</b> .	

**SMOVE**

SCR 39 COPY

*( scr<sub>1</sub> scr<sub>2</sub> count --- )*

Copies *count* Forth screens beginning with the source Forth screen *scr<sub>1</sub>* to the destination Forth screen *scr<sub>2</sub>*. Overlapping screen ranges may be specified without detrimental effects.

**SMTN**

SCR 57 -GRAPH

*( --- vaddr )*

A constant whose value is the VDP address of the Sprite Motion Table. Default value is **780h**.

**SMUDGE**

RESIDENT

*( --- )*

Used during word definition to toggle the smudge bit in a definition's name field. This prevents an uncompleted definition from being found during dictionary searches until compilation is completed without error.

**SP!**

RESIDENT

*( --- )*

A procedure to initialize the parameter stack pointer from **S0**, the user variable that points to the base of the parameter stack.

**SP@**

RESIDENT

*( --- addr )*

This word returns the address of the top of the stack as it was before **SP@** was executed, *e.g.*, **1 2 SP@ @ . . .** would type 2 2 1.

**SPACE**

RESIDENT

*( --- )*

Transmit a blank character (ASCII 32|**20h**) to the output device.

**SPACES**

RESIDENT

*( n --- )*

Transmit *n* blank characters (ASCII 32|**20h**) to the output device.

**SPCHAR**

SCR 58 -GRAPH

*( n<sub>1</sub> n<sub>2</sub> n<sub>3</sub> n<sub>4</sub> char --- )*

Defines a character *char* in the Sprite Descriptor Table to have the pattern composed of the 4 words (cells) on the stack.

<b>SPDTAB</b>	SCR 57 -GRAPH
( --- <i>vaddr</i> )	
A constant whose value is the VDP address of the Sprite Descriptor Table. Default value is <b>800h</b> . Notice that this coincides with the Pattern Descriptor Table.	
<b>SPLIT</b>	SCR 55 -SPLIT
( --- )	
Converts from present display screen mode into standard Split mode configuration.	
<b>SPLIT2</b>	SCR 55 -SPLIT
( --- )	
Converts from present display screen mode into standard Split2 mode configuration.	
<b>SPRCOL</b>	SCR 58 -GRAPH
( <i>n spr</i> --- )	
Changes color of the given sprite number <i>spr</i> to the color <i>n</i> specified.	
<b>SPRDIST</b>	SCR 60 -GRAPH
( <i>spr<sub>1</sub> spr<sub>2</sub></i> --- <i>n</i> )	
Returns on the stack the square of the distance <i>n</i> between two specified sprites, <i>spr<sub>1</sub></i> and <i>spr<sub>2</sub></i> . Distance is measured in pixels and the maximum distance that can be detected accurately is 181 pixels.	
<b>SPRDISTXY</b>	SCR 60 -GRAPH
( <i>dotcol dotrow spr</i> --- <i>n</i> )	
Places on the stack <i>n</i> , the square of the distance between the point ( <i>dotcol</i> , <i>dotrow</i> ) and a given sprite <i>spr</i> . Distance is measured in pixels and the maximum distance that can be detected accurately is 181 pixels.	
<b>SPRGET</b>	SCR 59 -GRAPH
( <i>spr</i> --- <i>dotcol dotrow</i> )	
Returns the dot column <i>dotcol</i> and dot row <i>dotrow</i> position of sprite <i>spr</i> .	
<b>SPRITE</b>	SCR 59 -GRAPH
( <i>dotcol dotrow n char spr</i> --- )	
Defines sprite number <i>spr</i> to have the specified location ( <i>dotcol</i> , <i>dotrow</i> ), color <i>n</i> , and character pattern <i>char</i> . The size of the sprite will depend on the magnification factor.	
<b>SPRPAT</b>	SCR 59 -GRAPH
( <i>char spr</i> --- )	
Changes the character pattern of a given sprite <i>spr</i> to <i>char</i> .	

**SPRPUT**

SCR 59 -GRAPH

 $( \textit{dotcol dotrow spr} \text{ ---} )$ Places a given sprite *spr* at location (*dotcol*,*dotrow*).**SQNTL**

SCR 69 -FILE

 $( \text{ ---} )$ Assigns the attribute SEQUENTIAL to the file whose PAB is pointed to by **PAB-ADDR**.**SQR**

SCR 50 -FLOAT

 $( f_1 \text{ ---} f_2 )$ Finds the square root of a floating point number  $f_1$  and leaves the result  $f_2$  on the stack.**SRA**

RESIDENT

 $( n_1 \textit{ count} \text{ ---} n_2 )$ Arithmetically shifts  $n_1$  count bits to the right and leaves the result  $n_2$  on the stack. Shifting by *count* will be modulo 16 except when *count* = 0, which causes 16 bits to be shifted. To create a word which does not perform a 16-bit shift when count is zero, use the following definition for the same stack contents:**: SRA0 -DUP IF SRA ENDIF ;****SRC**

RESIDENT

 $( n_1 \textit{ count} \text{ ---} n_2 )$ Performs a circular right shift of count bits on  $n_1$  leaving the result  $n_2$  on the stack. If *count* is 0, 16 bits are shifted. To create a word which does not perform a 16-bit shift when *count* is zero, use the following definition for the same stack contents:**: SRC0 -DUP IF SRC ENDIF ;****SRL**

RESIDENT

 $( n_1 \textit{ count} \text{ ---} n_2 )$ Performs a logical right shift of *count* bits on  $n_1$  and leaves the result  $n_2$  on the stack. If *count* is 0, 16 bits are shifted. To create a word which does not perform a 16-bit shift when count is zero, use the following definition for the same stack contents:**: SRL0 -DUP IF SRL ENDIF ;**



<b>SSDT</b>	SCR 58 -GRAPH
( <i>vaddr</i> --- )	
Places the Sprite Descriptor Table at the specified VDP address <i>vaddr</i> and initializes all sprite tables. The address given must be on an even 2K boundary. This instruction must be executed before sprites can be used.	
<b>STAT</b>	SCR 71 -FILE
( --- <i>b</i> )	
Reads the status of the current PAB and returns the status byte <i>b</i> to the stack. See the <i>Editor/Assembler Manual</i> for the meaning of each bit of the status byte.	
<b>STATE</b>	RESIDENT
( --- <i>addr</i> )	
A user variable containing the compilation state. A non-zero value indicates compilation. The value itself may be implementation dependent.	
<b>STCR</b>	SCR 88
( <i>n<sub>1</sub> addr</i> --- <i>n<sub>2</sub></i> )	
Performs the TMS9900 STCR instruction. The field width is <i>n<sub>1</sub></i> , the CRU base address is <i>addr</i> , and the returned value is <i>n<sub>2</sub></i> . The CRU base address will be shifted left 1 bit and stored in workspace register R12 prior to executing the TMS9900 STCR instruction.	
<b>STR</b>	SCR 47 -FLOAT
( --- )	
Converts the number in the FAC to a string, which is placed in PAD. The string is in BASIC format. Used by <b>F.</b> and <b>F.R.</b>	
<b>STR.</b>	SCR 47 -FLOAT
( <i>n<sub>1</sub> n<sub>2</sub> n<sub>3</sub></i> --- )	
Converts the number in the FAC to a string which is placed in PAD. The maximum number of output digits is <i>n<sub>1</sub></i> ( <b>STR.</b> places <i>n<sub>1</sub></i> in the byte at FAC+11). Calling <b>STR.</b> with <i>n<sub>1</sub></i> = 0 is identical to calling <b>STR</b> . The number of significant digits of output is <i>n<sub>2</sub></i> ( <b>STR.</b> places <i>n<sub>2</sub></i> in the byte at FAC+12). The number of digits to be output after the decimal point is <i>n<sub>3</sub></i> ( <b>STR.</b> places <i>n<sub>3</sub></i> in the byte at FAC+13). See the GPL STR routine on page 254 in the <i>Editor/Assembler Manual</i> for more detail.	
<b>SV</b>	SCR 71 -FILE
( <i>count</i> --- )	
Performs the file I/O save operation. The number of bytes <i>count</i> to be saved will be the size of the file on disk. The file's PAB must be set up and be the current PAB, to which <b>PAB-ADDR</b> points, before executing this word.	

<b>SWAP</b>	RESIDENT
$(n_1 n_2 \text{ --- } n_2 n_1)$ Exchange the top two values on the stack.	
<b>SWCH</b>	SCR 72 -PRINT
$(\text{ --- })$ A special purpose word which permits <b>EMIT</b> to output characters to an RS232 device rather than to the screen. See <b>UNSWCH</b> .	
<b>SWPB</b>	RESIDENT
$(n_1 \text{ --- } n_2)$ Reverses the order of the two bytes in $n_1$ and leaves the new number as $n_2$ .	
<b>SYS\$</b>	RESIDENT
$(\text{ --- addr})$ A user variable that contains the address of the system support entry point.	
<b>SYSTEM</b>	RESIDENT
$(n \text{ --- })$ Calls the system synonyms. You must specify an offset $n$ into a jump table for the routine you wish to call. The offset $n$ must be one of the predefined even numbers. See system Forth screen 33 for offsets 0 – 26.	
<b>TAN</b>	SCR 50 -FLOAT
$(f_1 \text{ --- } f_2)$ Finds the tangent of the floating point number ( $f_1$ = angle in radians) on the stack and leaves the result $f_2$ .	
<b>TASK</b>	RESIDENT
$(\text{ --- })$ A no-operation word or null definition, <b>TASK</b> is the last word defined in the resident Forth vocabulary of TI Forth and the last word that cannot be forgotten using <b>FORGET</b> . Its definition is simply : <b>TASK</b> ; . Its address can be used to <b>BSAVE</b> a personalized TI Forth system disk (see Chapter 11): ' <b>TASK 21 BSAVE</b> ( <i>Be sure to back up the original disk before trying this!</i> ). By redefining <b>TASK</b> at the beginning of an application, you can mark the boundary between applications. By forgetting <b>TASK</b> and re-compiling, an application can be discarded in its entirety. You will be able to <b>FORGET</b> each instance of the definition of <b>TASK</b> except the first one described above.	

<b>TB</b>	SCR 88 -CRU
( <i>addr</i> --- <i>flag</i> )	
<b>TB</b> performs the TMS9900 TB instruction. The bit at CRU address <i>addr</i> is tested by this instruction. Its value ( <i>flag</i> = 1 0 ) is returned to the stack. The CRU base address <i>addr</i> will be shifted left one bit and stored in workspace register R12 prior to executing the TMS9900 TB instruction.	
<b>TCHAR</b>	SCR 65 & 67 -64SUPPORT
( --- <i>addr</i> )	
Points to the array that holds the tiny character definitions for the 64-column editor. See <b>CLIST</b> .	
<b>TEXT</b>	SCR 51 -TEXT
( --- )	
Converts from present display screen mode into standard Text mode configuration.	
<b>THEN</b>	RESIDENT
( --- )	
An alias for <b>ENDIF</b> .	
<b>TIB</b>	RESIDENT
( --- <i>addr</i> )	
A user variable containing the address of the terminal input buffer.	
<b>TOGGLE</b>	RESIDENT
( <i>addr</i> <i>b</i> --- )	
Complement (XOR) the contents of the byte at <i>addr</i> by the bit pattern of byte <i>b</i> .	
<b>TRACE</b>	SCR 44 -TRACE
( --- )	
Forces the following colon definitions to be compiled in such a way that they can be traced. See <b>TRON</b> , <b>TROFF</b> and <b>UNTRACE</b> .	
<b>TRAVERSE</b>	RESIDENT
( <i>addr</i> <sub>1</sub> <i>n</i> --- <i>addr</i> <sub>2</sub> )	
Traverse the name field of a fig-Forth variable-length name field. The starting point <i>addr</i> <sub>1</sub> is the address of either the length byte or the last letter. If <i>n</i> = 1, the direction is toward high memory; if <i>n</i> = -1, the direction is toward low memory. The resulting address <i>addr</i> <sub>2</sub> points to the other end of the name.	

**TRIAD**

SCR 72 -PRINT

*( scr --- )*

Display on the RS232 device the three Forth screens that include screen number *scr*, beginning with a Forth screen evenly divisible by three. Output is suitable for source text records and includes a reference line at the bottom taken from line 15 of screen 4: "TI FORTH --- a fig-FORTH extension".

**TRIADS**

SCR 73 -PRINT

*( scr<sub>1</sub> scr<sub>2</sub> --- )*

May be thought of as a multiple **TRIAD**, *q.v.*. You must specify a Forth screen range. **TRIADS** will execute **TRIAD** as many times as necessary to cover that range.

**TROFF**

SCR 44 -TRACE

*( --- )*

Once a routine has been compiled with the **TRACE** option, it may be executed with or without a trace. To implement a trace, type **TRON** before execution. To execute without a trace, type **TROFF**. See **TRON**, **TRACE** and **UNTRACE**.

**TRON**

SCR 44 -TRACE

*( --- )*

See **TROFF**.

**TYPE**

RESIDENT

*( addr count --- )*

Transmit *count* characters from *addr* to the selected output device.

**U**

RESIDENT

*( --- n )*

Places the contents *n* of workspace register UP (R8) on the stack. Register U contains the base address of the user variable area. This is quicker than executing **U@**, which accomplishes the same thing.

**U\***

RESIDENT

*( u<sub>1</sub> u<sub>2</sub> --- ud )*

Leave the unsigned double number product *ud* of two unsigned numbers, *u<sub>1</sub>* and *u<sub>2</sub>*.

**U.**

RESIDENT

*( u --- )*

Prints an unsigned number *u* to the output device.

<b>U.R</b>	RESIDENT
( <i>u n</i> --- )	
Prints an unsigned number <i>u</i> right justified in a field of width <i>n</i> .	
<b>U/</b>	RESIDENT
( <i>ud u<sub>1</sub></i> --- <i>rem quot</i> )	
Leave the unsigned remainder <i>rem</i> and unsigned quotient <i>quot</i> from the unsigned double dividend <i>ud</i> and unsigned divisor <i>u<sub>1</sub></i> .	
<b>U0</b>	RESIDENT
( --- <i>addr</i> )	
A user variable that points to the base of the user variable area.	
<b>U&lt;</b>	RESIDENT
( <i>u<sub>1</sub> u<sub>2</sub></i> --- <i>flag</i> )	
Leaves a true flag if <i>u<sub>1</sub></i> is less than <i>u<sub>2</sub></i> , else leaves a false flag.	
<b>UD.</b>	RESIDENT
( <i>ud</i> --- )	
Prints an unsigned double number <i>ud</i> to the output device.	
<b>UD.R</b>	RESIDENT
( <i>ud n</i> --- )	
Prints an unsigned double number <i>ud</i> right justified in a field of length <i>n</i> .	
<b>UNCONS\$</b>	RESIDENT
( --- <i>addr</i> )	
A user variable which contains the base address of the user variable initial value table, which is used to initialize the user variables at a <b>COLD</b> start.	
<b>UNDRAW</b>	SCR 62 -GRAPH
( --- )	
Sets <b>DMODE</b> to 1. This means that dots are plotted in the off mode.	
<b>UNFORGETABLE</b> [ <i>sic</i> ]	RESIDENT
( <i>addr</i> --- <i>flag</i> )	
Decides whether or not a word can be forgotten. A true flag is returned if the address is not located between <b>FENCE</b> and <b>HERE</b> . Otherwise, a false flag is left. See <b>FORGET</b> . It is possible to set the value of <b>FENCE</b> to a value that is actually less than the address of the end of the last word in the core dictionary ( <b>TASK</b> ) such that <b>UNFORGETABLE</b> [ <i>sic</i> ] will report false; however, <b>FORGET</b> will still trap that error.	

**UNSWCH**

SCR 72 -PRINT

( --- )

Causes the computer to send output to the display screen instead of an RS232 device. See **SWCH** .

**UNTIL**

RESIDENT

Compilation: ( *addr n* --- ) Runtime: ( *flag* --- )

Occurs within a colon-definition in the form:

**BEGIN ... UNTIL**

At compile-time, **UNTIL** compiles (**0BRANCH**) and an offset from **HERE** to *addr*. Number *n* is used for error tests.

At runtime, **UNTIL** controls the conditional branch back to the corresponding **BEGIN** . If flag is *false*, execution returns to just after **BEGIN** ; if *true*, execution continues ahead.

**UNTRACE**

SCR 44 -TRACE

( --- )

Colon definitions that have been compiled under the **TRACE** option must be recompiled under the **UNTRACE** option to remove the tracing capability. **TRACE** and **UNTRACE** can be used alternately to select words to be traced.

**UPDATE**

RESIDENT

( --- )

Marks the most recently referenced block pointed to by **PREV** as altered. The block will subsequently be transferred automatically to disk should its buffer be required for storage of a different block. See **FLUSH** .

**UPDT**

SCR 69 -FILE

( --- )

Assigns the attribute UPDATE to the file whose PAB is pointed to by **PAB-ADDR** .

**USE**

RESIDENT

( --- *addr* )

A user variable containing the address of the block buffer to use next as the least recently written.

**USER**

RESIDENT

( *n* --- )

A defining word used in the form:

**n USER cccc**

which creates a user variable **cccc**. The parameter field of **cccc** contains *n* as a fixed offset relative to the user variable base address pointed to by workspace register UP (R8) for this user variable. When **cccc** is later executed, it places the sum of its offset and the user area base address on the stack as the storage address of that particular variable. You should only use the even numbers **68h – 7Eh** for *n*. You should actually avoid **68h** as well because the TI Forth boot screen (screen 3) uses that offset for defining user variable **VDPMDE**, leaving **6Ah – 7Eh** as the available offsets—enough for 11 user variables.

**VAL** SCR 47 -FLOAT

( --- )

Causes the string at PAD to be converted into a floating point number and put into the FAC. The string must have a leading length byte with no embedded blanks.

**VAND** SCR 33 -SYNONYMS

( *b vaddr* --- )

Performs a logical AND on the byte at the specified VDP location *vaddr* and the given byte *b*. The result byte is stored back into the VDP address.

**VARIABLE** RESIDENT

( *n* --- )

A defining word used in the form:

***n* VARIABLE cccc**

When **VARIABLE** is executed, it creates the definition **cccc** with its parameter field initialized to *n*. When **cccc** is later executed, the address of its parameter field (containing *n*) is left on the stack, so that a fetch or store may access this location.

**VCHAR** SCR 57 -GRAPH

( *col row count char* --- )

Prints on the display screen a vertical stream of length *count* of the specified character *char*. The first character of the stream is located at (*col,row*). Rows and columns are numbered from 0 beginning at the upper left of the display screen.

**VFILL** SCR 33 -SYNONYMS

( *vaddr count b* --- )

Fills *count* locations beginning at the given VDP address *vaddr* with the specified byte *b*.

**VLIST** SCR 43 -DUMP

( --- )

Prints the names of all words defined in the **CONTEXT** vocabulary. See **PAUSE**.

**VMBR**

SCR 33 -SYNONYMS

*( vaddr addr count --- )*

Reads *count* bytes beginning at the given VDP address *vaddr* and places them at *addr*.

**VMBW**

SCR 33 -SYNONYMS

*( addr vaddr count --- )*

Writes *count* bytes from *addr* into VDP beginning at the given VDP address *vaddr*.

**VOC - LINK**

RESIDENT

*( --- addr )*

A user variable containing the address of a field in the definition of the most recently created vocabulary. All vocabulary names are linked by these fields to allow control for forgetting with **FORGET** through multiple vocabularies.

**VOCABULARY**

RESIDENT

*( --- )*

A defining word used in the form:

**VOCABULARY   cccc**

to create a vocabulary definition **cccc** . Subsequent use of **cccc** will make it the **CONTEXT** vocabulary which is searched first by **INTERPRET** . The sequence **cccc DEFINITIONS** will also make **cccc** the **CURRENT** vocabulary into which new definitions are placed.

**cccc** will be so chained as to include all definitions of the vocabulary in which **cccc** is itself defined. All vocabularies ultimately chain to Forth. By convention, vocabulary names are to be declared **IMMEDIATE** . See **VOC - LINK** .

**VOR**

SCR 33 -SYNONYMS

*( b vaddr --- )*

Performs a logical OR on the byte at the specified VDP address and the given byte *b*. The result byte is stored back into the VDP address.

**VRBL**

SCR 68 -FILE

*( --- )*

Assigns the attribute VARIABLE to the file whose PAB is pointed to by **PAB-ADDR** .

**VSBR**

SCR 33 -SYNONYMS

*( vaddr --- b )*

Reads a single byte from the given VDP address *vaddr* and places its value *b* on the stack.



<b>VSBW</b>	SCR 33 -SYNONYMS
( <i>b vaddr</i> --- )	
Writes a single byte <i>b</i> into the given VDP address <i>vaddr</i> .	
<b>VWTR</b>	SCR 33 -SYNONYMS
( <i>b n</i> --- )	
Writes the given byte <i>b</i> into the specified VDP write-only register <i>n</i> .	
<b>VXOR</b>	SCR 33 -SYNONYMS
( <i>b vaddr</i> --- )	
Performs a logical XOR on the byte at the specified VDP address <i>vaddr</i> and the given byte <i>b</i> . The result byte is stored back into the VDP address <i>vaddr</i> .	
<b>WARNING</b>	RESIDENT
( --- <i>addr</i> )	
A user variable initialized by <b>COLD</b> at system startup containing a value controlling messages. If <b>WARNING</b> > 0, a disk is present and Forth screen 4 of drive 0 is the base location for messages. If <b>WARNING</b> = 0, no disk is present and messages will be presented by number. If <b>WARNING</b> < 0 when <b>ERROR</b> executes, <b>ERROR</b> will execute ( <b>ABORT</b> ) , which can be redefined to execute a user-specified procedure instead of the default <b>ABORT</b> . See <b>MESSAGE</b> , <b>ERROR</b> .	
<b>WDISK</b>	RESIDENT
( <i>addr n<sub>1</sub> n<sub>2</sub></i> --- <i>n<sub>3</sub></i> )	
The primitive routine which performs a disk write. The CPU RAM location of the block to be written is <i>addr</i> . The block number is <i>n<sub>1</sub></i> , the number of bytes per block is <i>n<sub>2</sub></i> and the returned error code is <i>n<sub>3</sub></i> .	
<b>WHERE</b> ( <i>EDITOR1 Vocabulary</i> )	SCR 38 -EDITOR
( <i>n<sub>1</sub> n<sub>2</sub></i> --- )	
When an error occurs on a <b>LOAD</b> instruction, typing <b>WHERE</b> will bring you into the 40-column editor and place the cursor at the exact location of the error. <b>WHERE</b> consumes the two numbers, <i>n<sub>1</sub></i> and <i>n<sub>2</sub></i> , left on the stack by the <b>LOAD</b> error.	
<b>WHERE</b> ( <i>EDITOR2 Vocabulary</i> )	SCR 29 -64SUPPORT
( <i>n<sub>1</sub> n<sub>2</sub></i> --- )	
When an error occurs on a <b>LOAD</b> instruction, typing <b>WHERE</b> will bring you into the 64-column editor and place the cursor at the exact location of the error. <b>WHERE</b> consumes the two numbers, <i>n<sub>1</sub></i> and <i>n<sub>2</sub></i> , left on the stack by the <b>LOAD</b> error.	

**WHILE**

RESIDENT

Compilation: ( *addr*<sub>1</sub> *n*<sub>1</sub> --- *addr*<sub>1</sub> *n*<sub>1</sub> *addr*<sub>2</sub> *n*<sub>2</sub> ) Runtime: ( *flag* --- )

Occurs in a colon-definition in the form:

**BEGIN** ... **WHILE** (true part) ... **REPEAT**

At compile time, **WHILE** emplaces (**0BRANCH**) and leaves *addr*<sub>2</sub> of the reserved offset. The stack values will be resolved by **REPEAT**.

At runtime, **WHILE** selects conditional execution based on *flag*. If *flag* is *true* (non-zero), **WHILE** continues execution of the true part through to **REPEAT**, which then branches back to **BEGIN**. If *flag* is *false* (zero), execution skips to just after **REPEAT**, exiting the structure.

**WIDTH**

RESIDENT

( --- *addr* )

A user variable containing the maximum number of letters saved in the compilation of a definition's name. It must be 1 – 31, with a default value of 31. The name character count and its natural characters are saved up to the value in **WIDTH**. The value may be changed at any time within the above limits.

**WLITERAL**

SCR 20 BOOT SCR

( --- )

A compiling word which compiles **SLIT** and the string which follows **WLITERAL** into the dictionary.

Used in the form: **WLITERAL cccc**

**WORD**

RESIDENT

( *char* --- )

Read the text characters from the input stream being interpreted until a delimiter *char* is found, storing the packed character string beginning at the dictionary buffer **HERE**. **WORD** leaves the character count in the first byte followed by the input characters and ends with two or more blanks. Leading occurrences of *char* are ignored. If **BLK** is zero, text is taken from the terminal input buffer, otherwise from the disk block stored in **BLK**. See **BLK**, **IN**.

**WRT**

SCR 71 -FILE

( *count* --- )

Performs the file I/O write operation. You must specify the number of bytes *count* to be written.

**XMLLNK**

SCR 33 -SYNONYMS

 $(addr \ ---)$ 

Links a Forth program to a routine in ROM or to a routine located in the memory expansion unit. A ROM address *addr* or XML vector must be specified as in the *Editor/Assembler Manual*.

**XOR**

RESIDENT

 $(n_1 \ n_2 \ --- \ n_3)$ 

Leave  $n_3$ , the bitwise logical exclusive OR (XOR) of  $n_1$  and  $n_2$ .

**[**

RESIDENT

 $( \ --- )$ 

Used in a colon-definition in the form:

```
      : xxxx [ words ] more ;
```

Suspend compilation. The words after **[** are executed, not compiled. This allows calculation or compilation exceptions before resuming compilation with **]**. See **LITERAL** and **]**.

**[COMPILE]**

RESIDENT

 $( \ --- )$ 

Used in a colon definition in the form:

```
      : xxxx [COMPILE] FORTH ;
```

**[COMPILE]** will force the compilation of an immediate definition that would otherwise execute during compilation. The above example will select the Forth vocabulary when **xxxx** executes rather than at compile time.

**]**

RESIDENT

 $( \ --- )$ 

Resume compilation to the completion of a colon definition. See **[**.

**^**

SCR 50 -FLOAT

 $(f_1 \ f_2 \ --- \ f_3)$ 

Returns  $f_3$  on the stack  $f_1$  raised to the  $f_2$  power. The operands must be floating point numbers.

**message**

SCR 84

 $( \ --- )$ 

A replacement for **MESSAGE** that contains the error messages in memory instead of on the disk. When Forth screen #84 is loaded, the error messages are compiled into the space formerly occupied by the fifth disk buffer leaving only four working disk buffers. **MESSAGE** is patched so that it now points to message.

## Appendix E User Variables in TI Forth

The purpose of this appendix is to detail the User Variables in TI Forth to assist in their use and to provide the necessary information to change or add to this list as necessary. A more comprehensive description of each of these variables is provided in Appendix D. The table follows these comments in two layouts. The first is in address offset order and the second is in alphabetical order by variable name.

The user may use even numbers **6Ah** through **7Eh** to create his/her own user variables. See the definition of **USER** in Appendix D.

### *E.1 TI Forth User Variables (Address Offset Order)*

Name	Offset	Initial Value	Description
<b>UCONS\$</b>	6h		Base of User Var initial value table
<b>S0</b>	8h		Base of Stack
<b>R0</b>	Ah		Base of Return Stack
<b>U0</b>	Ch		Base of User Variables
<b>TIB</b>	Eh		Terminal Input Buffer address
<b>WIDTH</b>	10h	31	Name length in dictionary
<b>DP</b>	12h		Dictionary Pointer
<b>SYS\$</b>	14h		Address of System Support
<b>CURPOS</b>	16h		Cursor location in VDP RAM
<b>INTLNK</b>	18h		Pointer to Interrupt Service Linkage
<b>WARNING</b>	1Ah	1	Message Control
<b>C/L\$</b>	1Ch	64	Characters per Line
<b>FIRST\$</b>	1Eh		Beginning of Disk Buffers
<b>LIMIT\$</b>	20h		End of Disk Buffers
<b>B/BUF\$</b>	22h	1024	Bytes per Buffer
<b>B/SCR\$</b>	24h	1	Blocks per Forth Screen
<b>DISK_LO</b>	26h	1	Low end Disk Fence
<b>DISK_HI</b>	28h	90	High end Disk Fence
<b>DISK_SIZE</b>	2Ah	90	Logical Disk Size in Forth Screens
<b>DISK_BUF</b>	2Ch	1000h	VDP location of 1K Buffer
<b>PABS</b>	2Eh	460h	VDP location for PABs
<b>SCRN_WIDTH</b>	30h	40	Display Screen Width in Characters
<b>SCRN_START</b>	32h	0	Display Screen Image Start in VDP
<b>SCRN_END</b>	34h	960	Display Screen Image End in VDP
<b>ISR</b>	36h		Interrupt Service Pointer
<b>ALTIN</b>	38h	0	Alternate Input Pointer
<b>ALTOUT</b>	3Ah	0	Alternate Output Pointer
<b>FENCE</b>	3Ch		Dictionary Fence
<b>BLK</b>	3Eh		Block being interpreted
<b>IN</b>	40h		Byte offset in text buffer
<b>OUT</b>	42h		Incremented by <b>EMIT</b>
<b>SCR</b>	44h		Last Forth Screen referenced

Name	Offset	Initial Value	Description
<b>OFFSET</b>	46h		Block offset to disks
<b>CONTEXT</b>	48h		Pointer to Context Vocabulary
<b>CURRENT</b>	4Ah		Pointer to Current Vocabulary
<b>STATE</b>	4Ch		Compilation State
<b>BASE</b>	4Eh		Number Base for Conversions
<b>DPL</b>	50h		Decimal Point Location
<b>FLD</b>	52h		Field Width (unused)
<b>CSP</b>	54h		Stack Pointer for error checking
<b>R#</b>	56h		Editing Cursor location
<b>HLD</b>	58h		Holds address during numeric conversion
<b>USE</b>	5Ah		Next Block Buffer to Use
<b>PREV</b>	5Ch		Most recently accessed disk buffer
[unavailable]	5Eh		—Do Not Use—
[unavailable]	60h		—Do Not Use—
<b>FORTH_LINK</b>	62h		Forth Vocabulary base
<b>ECOUNT</b>	64h		Error control
<b>VOC - LINK</b>	66h		Vocabulary linkage
<b>VDPMDE</b>	68h		VDP Mode (defined in TI Forth Screen #3)
[user to define]	6Ah		—available to user—
[user to define]	6Ch		—available to user—
[user to define]	6Eh		—available to user—
[user to define]	70h		—available to user—
[user to define]	72h		—available to user—
[user to define]	74h		—available to user—
[user to define]	76h		—available to user—
[user to define]	78h		—available to user—
[user to define]	7Ah		—available to user—
[user to define]	7Ch		—available to user—
[user to define]	7Eh		—available to user—

**E.2 TI Forth User Variables (Variable Name Order)**

Name	Offset	Initial Value	Description
ALTIN	38h	0	Alternate Input Pointer
ALTOUT	3Ah	0	Alternate Output Pointer
B/BUF\$	22h	1024	Bytes per Buffer
B/SCR\$	24h	1	Blocks per Forth Screen
BASE	4Eh		Number Base for Conversions
BLK	3Eh		Block being interpreted
C/L\$	1Ch	64	Characters per Line
CONTEXT	48h		Pointer to Context Vocabulary
CSP	54h		Stack Pointer for error checking
CURPOS	16h		Cursor location in VDP RAM
CURRENT	4Ah		Pointer to Current Vocabulary
DISK_BUF	2Ch	1000h	VDP location of 1K Buffer
DISK_HI	28h	90	High end Disk Fence
DISK_LO	26h	1	Low end Disk Fence
DISK_SIZE	2Ah	90	Logical Disk Size in Forth Screens
DP	12h		Dictionary Pointer
DPL	50h		Decimal Point Location
ECOUNT	64h		Error control
FENCE	3Ch		Dictionary Fence
FIRST\$	1Eh		Beginning of Disk Buffers
FLD	52h		Field Width (unused)
FORTH_LINK	62h		Forth Vocabulary base
HLD	58h		Holds address during numeric conversion
IN	40h		Byte offset in text buffer
INTLNK	18h		Pointer to Interrupt Service Linkage
ISR	36h		Interrupt Service Pointer
LIMIT\$	20h		End of Disk Buffers
OFFSET	46h		Block offset to disks
OUT	42h		Incremented by <b>EMIT</b>
PABS	2Eh	460h	VDP location for PABs
PREV	5Ch		Most recently accessed disk buffer
R#	56h		Editing Cursor location
R0	Ah		Base of Return Stack
S0	8h		Base of Stack
SCR	44h		Last Forth Screen referenced
SCRN_END	34h	960	Display Screen Image End in VDP
SCRN_START	32h	0	Display Screen Image Start in VDP
SCRN_WIDTH	30h	40	Display Screen Width in Characters
STATE	4Ch		Compilation State
SYS\$	14h		Address of System Support
TIB	Eh		Terminal Input Buffer address
U0	Ch		Base of User Variables
UCONS\$	6h		Base of User Var initial value table
USE	5Ah		Next Block Buffer to Use

Name	Offset	Initial Value	Description
<b>VDPMD</b>	<b>68h</b>		VDP Mode (defined in TI Forth Screen #3)
<b>VOC - LINK</b>	<b>66h</b>		Vocabulary linkage
<b>WARNING</b>	<b>1Ah</b>	<b>1</b>	Message Control
<b>WIDTH</b>	<b>10h</b>	<b>31</b>	Name length in dictionary
[unavailable]	<b>5Eh</b>		—Do Not Use—
[unavailable]	<b>60h</b>		—Do Not Use—
[user to define]	<b>6Ah</b>		—available to user—
[user to define]	<b>6Ch</b>		—available to user—
[user to define]	<b>6Eh</b>		—available to user—
[user to define]	<b>70h</b>		—available to user—
[user to define]	<b>72h</b>		—available to user—
[user to define]	<b>74h</b>		—available to user—
[user to define]	<b>76h</b>		—available to user—
[user to define]	<b>78h</b>		—available to user—
[user to define]	<b>7Ah</b>		—available to user—
[user to define]	<b>7Ch</b>		—available to user—
[user to define]	<b>7Eh</b>		—available to user—

## Appendix F TI Forth Load Option Directory

The load options are displayed on the TI Forth welcome screen and may subsequently be displayed by typing **MENU**. The load options allow you to load only the Forth extensions you wish to use.

You will notice, for example, that the **-EDITOR** option also loads the Forth screens that **-SYNONYMS** loads. The words loaded by **-SYNONYMS** are prerequisites for the words loaded by **-EDITOR**. If, by chance, the **-SYNONYMS** words were already in the dictionary at the time you type **-EDITOR**, they would not be loaded again. This is called a conditional load.

### F.1 Option: **-SYNONYMS**

Starting screen: 33

Words loaded:

<b>VSBW</b>	<b>VMBW</b>	<b>VSBR</b>
<b>VMBR</b>	<b>VWTR</b>	<b>GPLLNK</b>
<b>XMLLNK</b>	<b>DSRLNK</b>	<b>CLS</b>
<b>FORMAT-DISK</b>	<b>VFILL</b>	<b>VAND</b>
<b>VOR</b>	<b>VXOR</b>	<b>MON</b>
<b>RNDW</b>	<b>RND</b>	<b>SEED</b>
<b>RANDOMIZE</b>		

### F.2 Option: **-EDITOR (40-Column Editor)**

Starting screen: 34

Prerequisite options loaded:

**-SYNONYMS**

Words loaded:

<b>EDIT</b>	<b>ED@</b>	<b>WHERE</b>
-------------	------------	--------------

### F.3 Option: **-COPY**

Starting screen: 39

Words loaded:

<b>!"</b>	<b>DTEST</b>	<b>SCOPY</b>
<b>SMOVE</b>	<b>FORTH-COPY</b>	<b>DISK-HEAD</b>



**F.4 Option: -DUMP**

Starting screen: 42

Words loaded:

<b>DUMP</b>	<b>.S</b>	<b>VLIST</b>
-------------	-----------	--------------

**F.5 Option: -TRACE**

Starting screen: 44

Prerequisite options loaded:

**-DUMP**

Words loaded:

<b>TRACE</b>	<b>UNTRACE</b>	<b>TRON</b>
<b>TROFF</b>	<b>: (alternate)</b>	

**F.6 Option: -FLOAT**

Starting screen: 45

Prerequisite options loaded:

**-SYNONYMS**

Words loaded:

<b>FDUP</b>	<b>FDROP</b>	<b>FOVER</b>
<b>FSWAP</b>	<b>F!</b>	<b>F@</b>
<b>&gt;FAC</b>	<b>SETFL</b>	<b>FADD</b>
<b>FMUL</b>	<b>F+</b>	<b>F-</b>
<b>F*</b>	<b>F/</b>	<b>S-&gt;FAC</b>
<b>FAC-&gt;S</b>	<b>FAC&gt;ARG</b>	<b>F-&gt;S</b>
<b>S-&gt;F</b>	<b>FRND</b>	<b>STR</b>
<b>STR.</b>	<b>VAL</b>	<b>F\$</b>
<b>&gt;F</b>	<b>F.R</b>	<b>F.</b>
<b>FF.R.</b>	<b>FF.</b>	<b>F0&lt;</b>
<b>F0=</b>	<b>F&gt;</b>	<b>F=</b>
<b>F&lt;</b>	<b>FLERR</b>	<b>?FLERR</b>
<b>INT</b>	<b>^</b>	<b>SQR</b>
<b>EXP</b>	<b>LOG</b>	<b>COS</b>
<b>SIN</b>	<b>TAN</b>	<b>ATN</b>
<b>PI</b>		

***F.7 Option: -TEXT***

Starting screen: 51

Prerequisite options loaded:

**-SYNONYMS**

Words loaded:

**TEXT**

***F.8 Option: -GRAPH1***

Starting screen: 52

Prerequisite options loaded:

**-SYNONYMS**

Words loaded:

**GRAPHICS**

***F.9 Option: -MULTI***

Starting screen: 53

Prerequisite options loaded:

**-SYNONYMS**

Words loaded:

**MULTI**

***F.10 Option: -GRAPH2***

Starting screen: 54

Prerequisite options loaded:

**-SYNONYMS**

Words loaded:

**GRAPHICS2**

***F.11 Option: -SPLIT***

Starting screen: 55

Prerequisite options loaded:

**-SYNONYMS      -GRAPH2**

Words loaded:

**SPLIT              SPLIT2*****F.12 Option: -VDPMODES***

Starting screen: 51

Prerequisite options loaded:

**-SYNONYMS      -TEXT              -GRAPH1  
-MULTI          -GRAPH2          -SPLIT*****F.13 Option: -GRAPH***

Starting screen: 57

Prerequisite options loaded:

**-SYNONYMS      -CODE**

Words loaded:

<b>CHAR</b>	<b>CHARPAT</b>	<b>VCHAR</b>
<b>HCHAR</b>	<b>COLOR</b>	<b>SCREEN</b>
<b>GCHAR</b>	<b>SSDT</b>	<b>SPCHAR</b>
<b>SPRCOL</b>	<b>SPRPAT</b>	<b>SPRPUT</b>
<b>SPRITE</b>	<b>MOTION</b>	<b>#MOTION</b>
<b>SPRGET</b>	<b>DXY</b>	<b>SPRDIST</b>
<b>SPRDISTXY</b>	<b>MAGNIFY</b>	<b>JOYST</b>
<b>COINC</b>	<b>COINCXY</b>	<b>COINCALL</b>
<b>DELSPR</b>	<b>DELALL</b>	<b>MINIT</b>
<b>MCHAR</b>	<b>DRAW</b>	<b>UNDRAW</b>
<b>DTOG</b>	<b>DOT</b>	<b>LINE</b>

***F.14 Option: -FILE***

Starting screen: 68

Prerequisite options loaded:

**-SYNONYMS**

Words loaded:

<b>FILE</b>	<b>GET - FLAG</b>	<b>PUT - FLAG</b>
<b>SET - PAB</b>	<b>CLR - STAT</b>	<b>CHK - STAT</b>
<b>FXD</b>	<b>VRBL</b>	<b>DSPLY</b>
<b>INTRNL</b>	<b>I / OMD</b>	<b>INPT</b>
<b>OUTPT</b>	<b>UPDT</b>	<b>APPND</b>
<b>SQNTL</b>	<b>RLTV</b>	<b>REC - LEN</b>
<b>CHAR - CNT !</b>	<b>CHAR - CNT@</b>	<b>REC - NO</b>
<b>N - LEN !</b>	<b>F - D "</b>	<b>DOI / O</b>
<b>OPN</b>	<b>CLSE</b>	<b>RD</b>
<b>WRT</b>	<b>RSTR</b>	<b>LD</b>
<b>SV</b>	<b>DLT</b>	<b>SCRCH<sup>18</sup></b>
<b>STAT</b>		

***F.15 Option: -PRINT***

Starting screen: 72

Prerequisite options loaded:

**-SYNONYMS      -FILE**

Words loaded:

<b>SWCH</b>	<b>UNSWCH</b>	<b>?ASCII</b>
<b>TRIAD</b>	<b>TRIADS</b>	<b>INDEX</b>

***F.16 Option: -CODE***

Starting screen: 74

Words loaded:

**CODE                      ; CODE**

---

<sup>18</sup> See footnote 12, page 52.

***F.17 Option: -ASSEMBLER***

Starting screen: 75

Prerequisite options loaded:

**-CODE**

Words loaded:

Entire Assembler vocabulary. See Chapter 9.

***F.18 Option: -64SUPPORT ( 64-Column Editor )***

Starting screen: 22

Prerequisite options loaded:

**-SYNONYMS  
-GRAPH2**

**-GRAPH  
-SPLIT**

**-TEXT**

Words loaded:

**EDIT  
CLIST**

**ED@  
CLINE**

**WHERE**

***F.19 Option: -BSAVE***

Starting screen: 83

Words loaded:

**BSAVE**

***F.20 Option: -CRU***

Starting screen: 88

Prerequisite options loaded:

**-CODE**

Words loaded:

**SB0  
LDCR**

**SBZ  
STCR**

**TB**

## Appendix G Assembly Source for CODEd Words

Several words on the Forth System Disk have been written in TMS9900 code to increase their execution speeds and/or decrease their size. They include the words:

<b>SBO</b>	— a CRU instruction
<b>SBZ</b>	— a CRU instruction
<b>TB</b>	— a CRU instruction
<b>LDCR</b>	— a CRU instruction
<b>STCR</b>	— a CRU instruction
<b>DDOT</b>	— used by the dot plotting routine
<b>SMASH</b>	— used by <b>CLINE</b> and <b>CLIST</b>
<b>TCHAR</b>	— definitions for the tiny characters
<b>MON</b>	— returns to 99/4A color bar screen

These words have been coded in hexadecimal on your System disk, thus they do not require that the TI Forth Assembler be in memory before they can be loaded. Their Assembly source code (written in Forth Assembler) is listed on the following pages.

*Editor's Notes:* I detected a few errors and items in need of clarification in the TI Forth Assembler source code listed in this section. The errors are corrected in red text on the TI Forth screens in this section. The corrected lines are also highlighted in gray. The errors are as follows:

1. Screen 40, line 5: In the code for **SBZ**, the first **\*SP** should be **\*SP+**. The TMS9900-coded word on screen 88 of the TI Forth system diskette is correct.
2. Screen 43: There are several errors on this screen:
  - a. **DTAB** is supposed to be an initialized table of 12 cells (24 bytes), not just the one cell defined on this screen in the original (see screen 62 of the TI Forth system diskette to verify).
  - b. **DDOT** is missing **1 \*SP MOV**, and **NEXT**, from the end of the definition of **DDOT**, which can be verified by examining the code compiled into the dictionary from the source code here with screen 63 of the TI Forth system diskette.
3. Screen 44: This screen is missing the definitions of two variables (tables), viz., **TCHAR** and **LB**.
4. Screen 45 clarifications:
  - a. It should be noted that the definition of **TCHAR** in screen 45 is not actually Assembly source code. It is high-level Forth source code. It is also larger by two cells than the actual table stored on screen 67 of the TI Forth system disk. This was undoubtedly done on purpose to show the reader a neatly formatted table of the characters encoded. If you actually wanted to copy the table, resulting from loading screen 45, to screen 67 of the system disk, you would need to start the 384-byte transfer from an offset two cells beyond the beginning of **TCHAR**.
  - b. The comment, ( **^0** ) (Shift+0), on line 5 is a substitute for ( **)** , a syntax error.

```

SCR #40
0 ( SOURCE FOR CRU WORDS )  BASE->R  HEX
1 CODE SB0
2     *SP+ 0C MOV,  0C 0C A,
3         0 SB0,  NEXT,
4 CODE SBZ
5     *SP+ 0C MOV,  0C 0C A,
6         0 SBZ,  NEXT,
7 CODE TB
8     *SP 0C MOV,  0C 0C A,
9     *SP CLR,      0 TB,
10    EQ IF,
11    *SP INC,
12    ENDIF,
13    NEXT,
14
15 R->BASE -->

SCR #41
0 ( SOURCE FOR CRU WORDS ) BASE->R HEX
1 0C CONSTANT CRU
2 CODE LDCR
3     *SP+ CRU MOV,  CRU CRU A, *SP+ 1 MOV,
4     *SP+ 0 MOV,  01 OF ANDI,
5     NE IF,
6         01 08 CI,
7         LTE IF,
8         0 SWPB,
9         ENDIF,
10    ENDIF,
11    01 06 SLA,      01 3000 ORI,  01 X,
12 NEXT,
13
14
15 R->BASE -->

SCR #42
0 (SOURCE FOR CRU WORDS )  BASE->R  HEX
1 CODE STCR
2     *SP+ CRU MOV,  CRU CRU A,  *SP 01 MOV,
3     0 CLR,  01 000F ANDI,  01 02 MOV,
4     01 06 SLA,  01 3400 ORI,  01 X,
5     02 02 MOV,
6     NE IF,
7         02 08 CI,
8         LTE IF,
9         0 SWPB,
10    ENDIF,
11    ENDIF,
12    0 *SP MOV,
13 NEXT,
14
15 R->BASE

```

## SCR #43

0 ( SOURCE FOR DDOT )

1 BASE->R HEX 8040 VARIABLE DTAB 2010 , 804 , 201 , 7FBF , DFEF ,  
2 F7FB , FDFE , 8040 , 2010 , 804 , 201 ,

3 CODE DDOT

4 \*SP+ 1 MOV, \*SP 3 MOV, 1 2 MOV,  
5 3 4 MOV, 1 7 ANDI, 3 7 ANDI,  
6 2 F8 ANDI, 4 F8 ANDI, 2 5 SLA,  
7 2 1 A, 4 1 A, 1 2000 AI,  
8 4 CLR, DTAB 3 @(?) 4 MOVB,  
9 4 SWPB, 4 \*SP MOV, SP DECT,

10 1 \*SP MOV,

11 NEXT,

12

13

14

15 R->BASE

## SCR #44

0 ( SOURCE FOR SMASH ) BASE->R HEX

1 0 VARIABLE TCHAR 17E ALLOT 43 BLOCK TCHAR 180 CMOVE

2 TCHAR 7C - CONSTANT TC 0 VARIABLE LB FE ALLOT

3 CODE SMASH ( ADDR #CHAR LINE# --- LB VADDR CNT )

4 \*SP+ 1 MOV, \*SP+ 2 MOV, \*SP 3 MOV, 4 LB LI,  
5 4 \*SP MOV, SP DECT, 1 SWPB, 1 2000 AI,  
6 1 \*SP MOV, 2 1 MOV, 1 INC, 1 FFFE ANDI, SP DECT,  
7 1 2 SLA, 1 \*SP MOV,  
8 3 2 A, BEGIN, 2 3 C, GT WHILE, 5 CLR, 6 CLR,  
9 3 \*?+ 5 MOVB, 3 \*?+ 6 MOVB, 5 6 SRL, 6 6 SRL,  
10 BEGIN, TC 5 @(?) 0 MOV, TC 6 @(?) 1 MOV, 1 4 SRC,  
11 C 4 LI, BEGIN, 0 B MOV, B F000 ANDI, 1 7 MOV, 7 F00 ANDI,  
12 B 7 SOC, 7 4 \*?+ MOVB, 0 C SRC, 1 C SRC, C DEC, EQ UNTIL,  
13 5 INCT, 6 INCT, 5 C MOV, C 2 ANDI, EQ UNTIL, REPEAT,  
14 NEXT,  
15 R->BASE



## SCR #45

0 ( DEFINITIONS FOR TINY CHARACTERS ) BASE-&gt;R HEX

```

1 0EEE VARIABLE TCHAR EEEE ,
2 0000 , 0000 , ( ) 0444 , 4404 , ( ! ) 0AA0 , 0000 , ( " )
3 08AE , AEA2 , ( # ) 04EC , 46E4 , ( $ ) 0A24 , 448A , ( % )
4 06AC , 4A86 , ( & ) 0480 , 0000 , ( ' ) 0248 , 8842 , ( ( )
5 0842 , 2248 , ( ^0 ) 04EE , 0400 , ( * ) 0044 , E440 , ( + )
6 0000 , 0048 , ( , ) 0000 , E000 , ( - ) 0000 , 0004 , ( . )
7 0224 , 4488 , ( / ) 04AA , AAA4 , ( 0 ) 04C4 , 4444 , ( 1 )
8 04A2 , 488E , ( 2 ) 0C22 , C22C , ( 3 ) 02AA , AE22 , ( 4 )
9 0E8C , 222C , ( 5 ) 0688 , CAA4 , ( 6 ) 0E22 , 4488 , ( 7 )
10 04AA , 4AA4 , ( 8 ) 04AA , 622C , ( 9 ) 0004 , 0040 , ( : )
11 0004 , 0048 , ( ; ) 0024 , 8420 , ( < ) 000E , 0E00 , ( = )
12 0084 , 2480 , ( > ) 04A2 , 4404 , ( ? ) 04AE , AEA4 , ( @ )
13 04AA , EAAA , ( A ) 0CAA , CAAC , ( B ) 0688 , 8886 , ( C )
14 0CAA , AAAC , ( D ) 0E88 , C88E , ( E ) 0E88 , C888 , ( F )
15 -->

```

## SCR #46

0 ( TINY CHARACTERS CONTINUED )

```

1 04A8 , 8AA6 , ( G ) 0AAA , EAAA , ( H ) 0E44 , 444E , ( I )
2 0222 , 22A4 , ( J ) 0AAC , CAAA , ( K ) 0888 , 888E , ( L )
3 0AEE , AAAA , ( M ) 0AAE , EEAA , ( N ) 0EAA , AAEE , ( O )
4 0CAA , C888 , ( P ) 0EAA , AAEC , ( Q ) 0CAA , CAAA , ( R )
5 0688 , 422C , ( S ) 0E44 , 4444 , ( T ) 0AAA , AAEE , ( U )
6 0AAA , AA44 , ( V ) 0AAA , AEEA , ( W ) 0AA4 , 44AA , ( X )
7 0AAA , E444 , ( Y ) 0E24 , 488E , ( Z ) 0644 , 4446 , ( [ )
8 0884 , 4422 , ( \ ) 0C44 , 444C , ( ] ) 044A , A000 , ( $ )
9 0000 , 000F , ( _ ) 0420 , 0000 , ( ` ) 0004 , AEAA , ( a )
10 000C , ACAC , ( b ) 0006 , 8886 , ( c ) 000C , AAAC , ( d )
11 000E , 8C8E , ( e ) 000E , 8C88 , ( f ) 0004 , A8A6 , ( g )
12 000A , AEAA , ( h ) 000E , 444E , ( i ) 0002 , 22A4 , ( j )
13 000A , CCAA , ( k ) 0008 , 888E , ( l ) 000A , EEAA , ( m )
14 000A , EEEA , ( n ) 000E , AAEE , ( o ) 000C , AC88 , ( p )
15 -->

```

## SCR #47

0 ( TINY CHARACTERS CONCLUDED )

```

1 000E , AAEC , ( q ) 000C , ACAA , ( r ) 0006 , 842C , ( s )
2 000E , 4444 , ( t ) 000A , AAEE , ( u ) 000A , AA44 , ( v )
3 000A , AEEA , ( w ) 000A , A4AA , ( x ) 000A , AE44 , ( y )
4 000E , 248E , ( z ) 0644 , 8446 , ( { ) 0444 , 0444 , ( | )
5 0C44 , 244C , ( } ) 02E8 , 0000 , ( ~ ) 0EEE , EEEE , ( DEL )
6
7
8
9
10
11
12
13
14
15 R->BASE

```

SCR #48

0 ( SOURCE FOR MON ) BASE->R HEX

1

2 CODE MON

3 0 4E4F LI, 1 2000 LI,

4 BEGIN,

5 0 1 \*?+ MOV,

6 1 4000 CI,

7 EQ UNTIL,

8 0 @() BLWP,

9

10

11

12

13

14

15 R->BASE

## Appendix H Error Messages

Error#	Message	Probable Causes
1	empty stack	Procedure being executed attempts to pop a number off the parameter stack when there is no number on the parameter stack. The error may have occurred long before it is detected as Forth checks for this condition only when control returns to the outer interpreter.
2	dictionary full	The user dictionary space is full. Too many definitions have been compiled.
3	has incorrect address mode	Not used by TI Forth. Some fig-Forth assemblers use this message.
4	isn't unique	This message is more a warning than an error. It informs the user that a word with the same name as the one just compiled is already in the <b>CURRENT</b> or <b>CONTEXT</b> vocabulary.
6	disk error	This has several possible causes: No disk in disk drive, disk not initialized, disk drive or controller not connected properly, disk drive or controller not plugged in. The diskette may be damaged with some sector having a hard error.
7	full stack	The procedure being executed is leaving extra unwanted numbers on the parameter stack resulting in a stack overflow.
9	file I/O error	Any file I/O operation which results in an error will return this message. The <b>GET-FLAG</b> instruction will fetch the status byte. An error code of 0 indicates no error only if the COND bit (bit 2) of the STATUS byte located at <b>837Ch</b> is <i>not</i> set.

### code meaning

00	Bad device name
01	Device is write protected
02	Bad open attribute
03	Illegal operation
04	Out of table or buffer space on the device
05	Attempt to read past EOF
06	Device error
07	File error. Attempt to open nonexistent file, <i>etc.</i>

Error#	Message	Probable Causes																
10	floating point error	<p>This error message will be issued only when <b>?FLERR</b> is executed and a true flag is returned. <b>FLERR</b> may be executed to fetch the floating point status byte.</p> <table><tr><th>code</th><th>meaning</th></tr><tr><td>01</td><td>Overflow</td></tr><tr><td>02</td><td>Syntax</td></tr><tr><td>03</td><td>Integer overflow on conversion</td></tr><tr><td>04</td><td>Square root of negative</td></tr><tr><td>05</td><td>Negative number to non-integer power</td></tr><tr><td>06</td><td>Logarithm of a non-positive number</td></tr><tr><td>07</td><td>Invalid argument in a trigonometric function</td></tr></table>	code	meaning	01	Overflow	02	Syntax	03	Integer overflow on conversion	04	Square root of negative	05	Negative number to non-integer power	06	Logarithm of a non-positive number	07	Invalid argument in a trigonometric function
code	meaning																	
01	Overflow																	
02	Syntax																	
03	Integer overflow on conversion																	
04	Square root of negative																	
05	Negative number to non-integer power																	
06	Logarithm of a non-positive number																	
07	Invalid argument in a trigonometric function																	
11	disk fence violation	An attempt has been made to write to a screen outside the disk fence area. The values of <b>DISK_LO</b> and <b>DISK_HI</b> must be changed to include this screen before it may be written to.																
12	can't load from screen 0	Self explanatory. Loading from screen 0 is Forth's indication for loading from the keyboard.																
17	compilation only, use in definition	Occurs when conditional constructs such as <b>DO ... LOOP</b> or <b>IF ... THEN</b> are executed outside a colon definition.																
18	execution only	Occurs when you attempt to compile a compiling word into a colon definition.																
19	conditionals not paired	A <b>DO</b> has been left with a <b>LOOP</b> , an IF has no corresponding <b>THEN</b> , etc.																
20	definition not finished	A <b>;</b> was encountered and the parameter stack was not at the same height as when the preceding <b>:</b> was encountered. For example, <b>--&gt;</b>																
23	off current editing screen	Not used in TI Forth.																
24	declare vocabulary	Not used in TI Forth due to the way TI Forth's <b>FORGET</b> is configured.																
25	bad jump token	Improper use of jump tokens or conditionals in the TI Forth Assembler.																

## Appendix I Contents of the TI Forth Diskette

The Forth screens that follow have been modified from the original to fix known bugs as documented in Appendix J. The changed lines are highlighted in gray and the actual changes are marked by red text.

SCR #2

```
0           T I   F O R T H
1
2           THIS VERSION OF THE FORTH LANGUAGE
3           IS BASED ON THE fig-FORTH MODEL
4
5           THE ADDRESS OF THE FORTH INTEREST GROUP IS:
6
7           FORTH INTEREST GROUP
8           P.O. BOX 1105
9           SAN CARLOS, CA 94070
10
11          TEXAS INSTRUMENTS PERSONNEL WITH SIGNIFICANT
12          INPUT TO THIS VERSION INCLUDE:
13          LEON TIETZ
14          LESLIE O'HAGAN
15          EDWARD E. FERGUSON
```

## SCR #3

```

0 ( WELCOME SCREEN ) 0 0 GOTOXY ." BOOTING..." CR
1 BASE->R HEX 10 83C2 C! ( QUIT OFF! )
2 DECIMAL ( 84 LOAD ) 20 LOAD 16 SYSTEM MENU
3 HEX 68 USER VDPMD 1 VDPMD ! DECIMAL
4 : -SYNONYMS 33 LOAD ; : -EDITOR 34 LOAD ; : -COPY 39 LOAD ;
5 : -DUMP 42 LOAD ; : -TRACE 44 LOAD ; : -FLOAT 45 LOAD ;
6 : -TEXT 51 LOAD ; : -GRAPH1 52 LOAD ; : -MULTI 53 LOAD ;
7 : -GRAPH2 54 LOAD ; : -SPLIT 55 LOAD ; : -GRAPH 57 LOAD ;
8 : -FILE 68 LOAD ; : -PRINT 72 LOAD ; : -CODE 74 LOAD ;
9 : -ASSEMBLER 75 LOAD ; : -64SUPPORT 22 LOAD ;
10 : -VDPMODES -TEXT -GRAPH1 -MULTI -GRAPH2 -SPLIT ;
11 : -BSAVE 83 LOAD ; : -CRU 88 LOAD ;
12
13
14
15 R->BASE

```

## SCR #4

```

0 ( ERROR MESSAGES )
1 empty stack
2 dictionary full
3 has incorrect address mode
4 isn't unique.
5
6 disk error
7 full stack
8
9 file i/o error
10 floating point error
11 disk fence violation
12 can't load from screen zero
13
14
15 TI FORTH --- a fig-FORTH extension

```

## SCR #5

```

0 ( ERROR MESSAGES )
1 compilation only, use in definition
2 execution only
3 conditionals not paired
4 definition not finished
5 in protected dictionary
6 use only when loading
7 off current editing screen
8 declare vocabulary
9 bad jump token
10
11
12
13
14
15

```

## SCR #20

```
0 ( CONDITIONAL LOAD )
1 : MENU CR 272 265 DO I MESSAGE CR LOOP CR CR CR ;
2 : SLIT ( --- ADDR OF STRING LITERAL )
3   R> DUP C@ 1+ =CELLS OVER + >R ;
4
5 : WLITERAL ( WLITERAL word )
6   BL STATE @
7   IF COMPILE SLIT WORD HERE C@ 1+ =CELLS ALLOT
8   ELSE WORD HERE ENDIF ; IMMEDIATE -->
9 -SYNONYMS      -EDITOR      -COPY
10 -DUMP          -TRACE       -FLOAT
11 -TEXT          -GRAPH1      -MULTI
12 -GRAPH2        -SPLIT       -VDPMODES
13 -GRAPH         -FILE        -PRINT
14 -CODE          -ASSEMBLER    -64SUPPORT
15 -BSAVE         -CRU
```

## SCR #21

```

0 ( CONDITIONAL LOAD )
1 : <CLOAD> ( SCREEN STRING_ADDR --- )
2   CONTEXT @ @ (FIND)
3   IF DROP DROP 0=
4     IF BLK @
5       IF R> DROP R> DROP
6     ENDIF
7   ENDIF
8   ELSE -DUP
9     IF LOAD
10    ENDIF
11  ENDIF ;
12 : CLOAD ( scr_no CLOAD name )
13   [COMPILE] WLITERAL STATE @
14   IF COMPILE <CLOAD> ELSE <CLOAD> ENDIF
15 ; IMMEDIATE

```

## SCR #22

```

0 ( 64 COLUMN EDITOR ) 0 CLOAD ED@
1 BASE->R DECIMAL 57 R->BASE CLOAD LINE BASE->R DECIMAL 51 R->BASE
2 CLOAD TEXT BASE->R DECIMAL 54 R->BASE CLOAD GRAPHICS2 BASE->R
3 DECIMAL 55 R->BASE CLOAD SPLIT
4 BASE->R DECIMAL 65 R->BASE CLOAD CLIST
5 BASE->R HEX ( 3800 ' SATR ! )
6 VOCABULARY EDITOR2 IMMEDIATE EDITOR2 DEFINITIONS
7 0 VARIABLE CUR
8 : !CUR 0 MAX B/SCR B/BUF * 1- MIN CUR ! ;
9 : +CUR CUR @ + !CUR ;
10 : +LIN CUR @ C/L / + C/L * !CUR ;          DECIMAL
11 : LINE. DO I SCR @ (LINE) I CLINE LOOP ;
12 : BCK 0 0 GOTOXY QUIT ;
13 : PTR SCR @ B/SCR * CUR @ B/BUF /MOD ROT + BLOCK + ;
14 : R/C CUR @ C/L /MOD ; ( --- COL ROW )    R->BASE -->
15

```

## SCR #23

```

0 ( 64 COLUMN EDITOR ) BASE->R HEX
1
2 : CINIT 3800 DUP ' SPDTAB ! 800 / 6 VWTR 3800 ' SATR !
3 SATR 2 0 DO DUP >R D000 SP@ R> 2 VMBW DROP 4 + LOOP DROP
4 0000 0000 0000 0000 5 SPCHAR 0 CUR !
5 F090 9090 9090 90F0 6 SPCHAR 0 1 F 5 0 SPRITE ; DECIMAL
6
7 : PLACE CUR @ 64 /MOD 8 * 1+ SWAP 4 * 1- DUP 0< IF DROP 0 ENDIF
8 SWAP 0 SPRPUT ;
9 : UP -64 +CUR PLACE ;
10 : DOWN 64 +CUR PLACE ;
11 : LEFT -1 +CUR PLACE ;
12 : RIGHT 1 +CUR PLACE ;
13 : CGOTOXY ( COL ROW --- ) 64 * + !CUR PLACE ;
14
15 R->BASE -->

```



## SCR #24

```

0 ( 64 COLUMN EDITOR ) BASE->R
1
2 DECIMAL
3
4 : .CUR CUR @ C/L /MOD CGOTOXY ;
5 : DELHALF PAD 64 BLANKS PTR PAD C/L R/C DROP - CMOVE ;
6
7 : DELLIN R/C SWAP MINUS +CUR PTR PAD C/L CMOVE DUP L/SCR SWAP
8   DO PTR 1 +LIN PTR SWAP C/L CMOVE LOOP
9   0 +LIN PTR C/L 32 FILL C/L * !CUR ;
10 : INSLIN R/C SWAP MINUS +CUR L/SCR +LIN DUP 1+ L/SCR 0 +LIN
11   DO PTR -1 +LIN PTR SWAP C/L CMOVE -1 +LOOP
12   PAD PTR C/L CMOVE C/L * !CUR ;
13 : RELINE R/C SWAP DROP DUP LINE. UPDATE .CUR ;
14 : +.CUR +CUR .CUR ;
15 R->BASE -->

```

## SCR #25

```

0 ( 64 COLUMN EDITOR ) BASE->R DECIMAL
1 : -TAB PTR DUP C@ BL >
2   IF BEGIN 1- DUP -1 +CUR C@ BL =
3     UNTIL
4   ENDIF
5   BEGIN CUR @ IF 1- DUP -1 +CUR C@ BL > ELSE .CUR 1 ENDIF UNTIL
6   BEGIN CUR @ IF 1- DUP -1 +CUR C@ BL = DUP IF 1 +.CUR ENDIF
7     ELSE .CUR 1 ENDIF
8   UNTIL DROP ;
9 : TAB PTR DUP C@ BL = 0=
10  IF BEGIN 1+ DUP 1 +CUR C@ BL =
11    UNTIL
12  ENDIF
13  CUR @ 1023 = IF .CUR 1
14    ELSE BEGIN 1+ DUP 1 +CUR C@ BL > UNTIL .CUR
15    ENDIF DROP ; R->BASE -->

```

## SCR #26

```

0 ( 64 COLUMN EDITOR ) BASE->R
1 DECIMAL
2 : !BLK PTR C! UPDATE ;
3 : BLNKS PTR R/C DROP C/L SWAP - 32 FILL ;
4 : HOME 0 0 CGOTOXY ;
5 : REDRAW SCR @ CLIST UPDATE .CUR ;
6 : SCRNO CLS 0 0 GOTOXY ." SCR #" SCR @ BASE->R DECIMAL U.
7   R->BASE CR ;
8 : +SCR SCR @ 1+ DUP SCR ! SCRNO CLIST ;
9 : -SCR SCR @ 1- 0 MAX DUP SCR ! SCRNO CLIST ;
10 : DEL PTR DUP 1+ SWAP R/C DROP C/L SWAP - CMOVE 32
11   PTR R/C DROP - C/L + 1- C! ;
12 : INS 32 PTR DUP R/C DROP C/L SWAP - + SWAP DO
13   I C@ LOOP DROP PTR DUP R/C DROP C/L SWAP - + 1- SWAP 1- SWAP
14   DO I C! -1 +LOOP ; R->BASE -->
15

```

## SCR #27

```

0 ( 64 COLUMN EDITOR 15JUL82 LA0 )          BASE->R DECIMAL
1 0 VARIABLE BLINK 0 VARIABLE OKEY
2 10 CONSTANT RL 150 CONSTANT RH 0 VARIABLE KC RH VARIABLE RLOG
3 : RKEY BEGIN ?KEY -DUP 1 BLINK +! BLINK @ DUP 60 < IF 6 0 SPRPAT
4 ELSE 5 0 SPRPAT ENDIF 120 = IF 0 BLINK ! ENDIF
5     IF ( SOME KEY IS PRESSED ) KC @ 1 KC +! 0 BLINK !
6     IF ( WAITING TO REPEAT ) RLOG @ KC @ <
7     IF ( LONG ENOUGH ) RL RLOG ! 1 KC ! 1 ( FORCE EXT)
8     ELSE OKEY @ OVER =
9     IF DROP 0 ( NEED TO WAIT MORE )
10    ELSE 1 ( FORCE EXIT ) DUP KC ! ENDIF
11    ENDIF
12    ELSE ( NEW KEY ) 1 ( FORCE LOOP EXIT ) ENDIF
13    ELSE ( NO KEY PRESSED) RH RLOG ! 0 KC ! 0
14    ENDIF
15 UNTIL DUP OKEY ! ; R->BASE -->

```

## SCR #28

```

0 ( 64 COLUMN EDITOR ) BASE->R HEX
1 : EDT VDPME @ 5 = 0= IF SPLIT ENDIF CINIT !CUR R/C CGOTOXY
2 DUP DUP SCR ! SCRNO CLIST BEGIN RKEY
3 CASE 08 OF LEFT ENDOF 0C OF -SCR ENDOF
4 0A OF DOWN ENDOF 03 OF DEL RELINE ENDOF
5 0B OF UP ENDOF 04 OF INS RELINE ENDOF
6 09 OF RIGHT ENDOF 07 OF DELLIN REDRAW ENDOF
7 0E OF HOME ENDOF 06 OF INSLIN REDRAW ENDOF
8 02 OF +SCR ENDOF 16 OF TAB ENDOF
9 0D OF 1 +LIN .CUR PLACE ENDOF 7F OF -TAB ENDOF
10 01 OF DELHALF BLNKS RELINE ENDOF
11 0F OF 5 0 SPRPAT CLS SCRNO DROP 300 ' SATR ! QUIT ENDOF
12 1E OF INSLIN BLNKS REDRAW ENDOF
13 DUP 1F > OVER 7F < AND IF DUP !BLK R/C SWAP DROP DUP SCR @
14 (LINE) ROT CLINE 1 +.CUR ELSE 7 EMIT ENDIF ENDCASE AGAIN ;
15 R->BASE -->

```

## SCR #29

```

0 ( 64 COLUMN EDITOR ) BASE->R HEX
1 FORTH DEFINITIONS
2 : EDIT EDITOR2 0 EDT ;
3 : WHERE EDITOR2 B/SCR /MOD SWAP B/BUF * ROT + 2- EDT ;
4
5 : ED@ EDITOR2 SCR @ SCRNO EDIT ;
6
7
8
9
10
11
12
13
14
15 R->BASE

```

## SCR #33

```

0 ( SYSTEM CALLS 09JUL82 LCT) 0 CLOAD RANDOMIZE
1 BASE->R DECIMAL 74 R->BASE CLOAD ;CODE
2 BASE->R DECIMAL
3 : VSBW 0 SYSTEM ; : VMBW 2 SYSTEM ;
4 : VSBR 4 SYSTEM ; : VMBR 6 SYSTEM ;
5 : VWTR 8 SYSTEM ; : GPLLNK 0 33660 C! 10 SYSTEM ;
6 : XMLLNK 12 SYSTEM ; : DSRLNK 8 14 SYSTEM ;
7 : CLS 16 SYSTEM ; : FORMAT-DISK 1+ 18 SYSTEM ;
8 : VFILL 20 SYSTEM ; : VAND 22 SYSTEM ; : VOR 24 SYSTEM ;
9 : VXOR 26 SYSTEM ;      HEX
10 CODE MON 0200 , 4E4F , 0201 , 2000 , CC40 , 0281 , 4000 , 16FC ,
11      0420 , 0000 ,
12 : RNDW 83C0 DUP @ 6FE5 * 7AB9 + 5 SRC DUP ROT ! ;
13 : RND RNDW ABS SWAP MOD ; : SEED 83C0 ! ;
14 : RANDOMIZE 8802 C@ DROP 0 BEGIN 1+ 8802 C@ 80 AND UNTIL SEED ;
15 R->BASE

```

## SCR #34

```

0 ( SCREEN EDITOR 09JUL82 LCT) 0 CLOAD ED@
1 BASE->R DECIMAL 33 R->BASE CLOAD RANDOMIZE
2 BASE->R HEX VOCABULARY EDITOR1 IMMEDIATE EDITOR1 DEFINITIONS
3 : BOX 8F7 8F1 D0 84 I VSBW LOOP ;
4 : CUR R# ;
5 : !CUR 0 MAX B/SCR B/BUF * 1- MIN CUR ! ;
6 : +CUR CUR @ + !CUR ;
7 : +LIN CUR @ C/L / + C/L * !CUR ;
8 0 VARIABLE S_H DECIMAL
9 : FTYPE 40 * 124 + SWAP VMBW ;
10 : LISTA DECIMAL 0 0 GOTOXY DUP SCR !
11 : ." SCR # " . CR CR CR 16 0 D0 I 3 .R CR LOOP ;
12 : ROWCAL S_H @ IF 29 + ENDIF ;
13 : LINE. DO I SCR @ (LINE) DROP ROWCAL 35 I FTYPE LOOP ;
14 : LISTB L/SCR 0 LINE. ;
15 R->BASE -->

```

## SCR #35

```

0 ( SCREEN EDITOR 09JUL82 LCT)
1
2 : LISTL BASE->R LISTA 4 1 GOTOXY
3 ."      1      2      3      " 4 2 GOTOXY
4 ." .....0.....0.....0....."
5 0 S_H ! LISTB R->BASE ;
6 : LISTR BASE->R DROP 4 1 GOTOXY
7 ." 3      4      5      6      " 4 2 GOTOXY
8 ." 0.....0.....0.....0....."
9 1 S_H ! LISTB R->BASE ;
10 : BCK 0 L/SCR 2+ GOTOXY QUIT ;
11 : PTR SCR @ B/SCR * CUR @ B/BUF /MOD ROT + BLOCK + ;
12 : R/C CUR @ C/L /MOD ; ( --- COL ROW )
13 : DELHALF PAD 64 BLANKS PTR PAD C/L R/C DROP - CMOVE ;
14
15 -->

```

## SCR #36

```

0 ( SCREEN EDITOR 12JUL82 LCT) BASE->R DECIMAL
1 : .CUR CUR @ C/L /MOD 3 + SWAP 4 + DUP S_H @
2   IF 32 > IF 29 - ELSE SCR @ LISTL ENDIF
3   ELSE 39 < 0= IF SCR @ LISTR 29 - ENDIF
4   ENDIF SWAP GOTOXY ;
5 : DELLIN R/C SWAP MINUS +CUR PTR PAD C/L CMOVE DUP L/SCR SWAP
6   DO PTR 1 +LIN PTR SWAP C/L CMOVE LOOP
7   0 +LIN PTR C/L 32 FILL C/L * !CUR ;
8 : INSLIN R/C SWAP MINUS +CUR L/SCR +LIN DUP 1+ L/SCR 0 +LIN
9   DO PTR -1 +LIN PTR SWAP C/L CMOVE -1 +LOOP
10  PAD PTR C/L CMOVE C/L * !CUR ;
11 : RELINE R/C SWAP DROP DUP 13 EMIT LINE. UPDATE .CUR ;
12 : +.CUR +CUR .CUR ;
13 : TAB PTR DUP @ 32 = 0= IF BEGIN 1+ DUP 1 +CUR C@ 32 = UNTIL
14   ENDIF CUR @ 1023 = IF .CUR 1 ELSE BEGIN 1+ DUP 1 +CUR C@ 32 >
15   UNTIL .CUR ENDIF ; R->BASE -->

```

## SCR #37

```

0 ( SCREEN EDITOR 12JUL82 LCT) BASE->R DECIMAL
1 : -TAB PTR DUP C@ 32 > IF BEGIN 1- DUP -1 +CUR C@ 32 = UNTIL
2   ENDIF BEGIN CUR @ IF 1- DUP -1 +CUR C@ 32 > ELSE .CUR 1 ENDIF
3   UNTIL BEGIN CUR @ IF 1- DUP -1 +CUR C@ 32 = DUP IF 1 +.CUR
4   ENDIF ELSE .CUR 1 ENDIF UNTIL ; : !BLK PTR C! UPDATE 1 +.CUR ;
5 : BLNKS PTR R/C DROP C/L SWAP - 32 FILL ;
6 : FLIP S_H @ IF -29 ELSE 29 ENDIF +.CUR ;
7 : REDRAW SCR @ S_H @ IF LISTR ELSE LISTL ENDIF UPDATE .CUR ;
8 : NEWSR 0 SWAP LISTL !CUR .CUR ;
9 : +SCR SCR @ 1+ NEWSR ;
10 : -SCR SCR @ 1- 0 MAX NEWSR ;
11 : DEL PTR DUP 1+ SWAP R/C DROP C/L SWAP - CMOVE 32
12   PTR R/C DROP - C/L + 1- C! ;
13 : INS 32 PTR DUP R/C DROP C/L SWAP - + SWAP DO
14   I C@ LOOP DROP PTR DUP R/C DROP C/L SWAP - + 1- SWAP 1- SWAP
15   DO I C! -1 +LOOP ; R->BASE -->

```

## SCR #38

```

0 ( SCREEN EDITOR 12JUL82 LCT) BASE->R HEX
1 : VED BOX SWAP CLS LISTL !CUR .CUR BEGIN KEY CASE
2   0F OF BCK                ENDOF 01 OF DELHALF BLNKS RELINE ENDOF
3   08 OF -1 +.CUR           ENDOF 02 OF +SCR                ENDOF
4   0A OF C/L +.CUR          ENDOF 0C OF -SCR                ENDOF
5   0B OF C/L MINUS +.CUR    ENDOF 03 OF DEL RELINE          ENDOF
6   09 OF 1 +.CUR            ENDOF 04 OF INS RELINE          ENDOF
7   0D OF 1 +LIN .CUR        ENDOF 07 OF DELLIN REDRAW       ENDOF
8   0E OF FLIP               ENDOF 06 OF INSLIN REDRAW       ENDOF
9   1E OF INSLIN BLNKS REDRAW ENDOF 16 OF TAB                ENDOF
10  7F OF -TAB ENDOF
11  DUP 1F > OVER 7F < AND IF DUP EMIT DUP !BLK ELSE 7 EMIT ENDIF
12  ENDCASE AGAIN ; FORTH DEFINITIONS
13 : WHERE EDITOR1 B/SCR /MOD SWAP B/BUF * ROT + 2- VED ;
14 : EDIT EDITOR1 0 VED ; : ED@ EDITOR1 SCR @ EDIT ;
15 R->BASE

```

## SCR #39

```

0 ( STRING STORE AND SCREEN COPY WORDS 12JUL82 LCT) 0 CONSTANT AD
1 0 CLOAD DISK-HEAD ( ADDR --- ) BASE->R HEX
2 : (!") R COUNT DUP 1+ =CELLS R> + >R >R SWAP R> CMOVE ;
3 : !" 22 STATE @ ( STORE STRING AT ADDR )
4     IF COMPILE (!") WORD HERE C@
5         1+ =CELLS ALLLOT
6     ELSE WORD HERE COUNT >R SWAP R> CMOVE
7     ENDIF ; IMMEDIATE DECIMAL ( SCREEN COPYING WORDS )
8 : DTEST 90 0 DO I DUP . BLOCK DROP LOOP ;
9 : SCOPY OFFSET @ + SWAP BLOCK 2- ! UPDATE FLUSH ; ( 1K BLOCKS )
10 : SMOVE >R OVER OVER - DUP 0< SWAP R MINUS > + 2 = IF
11     OVER OVER SWAP R + 1- SWAP R + 1- -1 ' AD ! ELSE 1 ' AD !
12     ENDIF R> 0 DO OVER OVER SCOPY AD + SWAP AD + SWAP LOOP DROP
13     DROP ;
14 : FORTH-COPY 90 0 DO I DUP . 90 + I SCOPY LOOP ;
15 R->BASE -->

```

## SCR #40

```

0 ( WRITE A HEAD COMPATABLE WITH THE DISK MANAGER 12JUL82 LCT)
1 BASE->R HEX
2 : DISK-HEAD 0 CLEAR 0 BLOCK ( START SECTOR 0)
3     DUP !" FORTH      " DUP A + 168 SWAP !
4     DUP C + 944 SWAP ! DUP E + 534B SWAP !
5     DUP 10 + 2000 SWAP ! DUP 12 + 26 0 FILL
6     DUP 38 + C8 FF FILL 100 + ( START SECTOR 1)
7     DUP 2 SWAP ! DUP 2+ FE 00 FILL
8     100 + ( START SECTOR 2)
9     DUP !" SCREENS  " DUP A + 0 SWAP !
10    DUP C + 2 SWAP ! DUP E + 165 SWAP !
11    DUP 10 + 80 SWAP ! DUP 12 + CA02 SWAP !
12    DUP 14 + 8 0 FILL DUP 1C + 2250 SWAP !
13    DUP 1E + 1403 SWAP ! DUP 20 + 4016 SWAP ! 22 + 0DE 0 FILL
14    FLUSH
15 ; R->BASE

```

## SCR #42

```

0 ( DUMP ROUTINES 12JUL82 LCT)
1 0 CLOAD VLIST BASE->R HEX
2 : DUMP8 -DUP
3 IF
4 BASE->R HEX 0 OUT ! SPACE OVER 4 U.R
5 OVER OVER 0 DO
6 DUP @ 0 <# # # # BL HOLD BL HOLD #> TYPE 2+ 2
7 +LOOP DROP 1F OUT @ - SPACES
8 0 DO
9 DUP C@ DUP 20 < OVER 7E > OR
10 IF DROP 2E ENDIF
11 EMIT 1+
12 LOOP
13 CR R->BASE
14 ENDIF ;
15 -->

```

## SCR #43

```

0 ( DUMP ROUTINES 12JUL82 LCT)
1 : DUMP CR 00 8 U/ >R SWAP R> -DUP
2 IF 0
3 DO 8 DUMP8 PAUSE IF SWAP DROP 0 SWAP LEAVE ENDIF LOOP
4 ENDIF SWAP DUMP8 DROP ;
5 : .S CR SP@ 2- S0 @ 2- ." | " OVER OVER = 0= IF
6 DO I @ U. -2 +LOOP ELSE DROP DROP ENDIF ;
7 : VLIST 80 OUT ! CONTEXT @ @
8 BEGIN DUP C@ 3F AND OUT @ + 25 >
9 IF CR 0 OUT ! ENDIF
10 DUP ID. PFA LFA @ SPACE DUP 0= PAUSE OR
11 UNTIL DROP ; R->BASE
12
13
14
15

```

## SCR #44

```

0 ( TRACE COLON WORDS-FORTH DIMENSIONS III/2 P.58 26OCT82 LCT)
1 0 CLOAD (TRACE) BASE->R DECIMAL 42 R->BASE CLOAD VLIST
2 FORTH DEFINITIONS
3 0 VARIABLE TRACF ( CONTROLS INSERTION OF TRACE ROUTINE )
4 0 VARIABLE TFLAG ( CONTROLS TRACE OUTPUT )
5 : TRACE 1 TRACF ! ;
6 : UNTRACE 0 TRACF ! ;
7 : TRON 1 TFLAG ! ;
8 : TROFF 0 TFLAG ! ;
9 : (TRACE) TFLAG @ ( GIVE TRACE OUTPUT? )
10 IF CR R 2- NFA ID. ( BACK TO PFA NFA FOR NAME )
11 .S ENDIF ; ( PRINT STACK CONTENTS )
12 : : ( REDEFINED TO INSERT TRACE WORD AFTER COLON )
13 ?EXEC !CSP CURRENT @ CONTEXT ! CREATE [ ' : CFA @ ] LITERAL
14 HERE 2- ! TRACF @ IF ' (TRACE) CFA DUP @ HERE 2- ! , ENDIF ]
15 ; IMMEDIATE

```

## SCR #45

```

0 ( FLOATING POINT <4 WORD> STACK ROUTINES 12JUL82 LCT)
1 0 CLOAD PI BASE->R DECIMAL 33 R->BASE CLOAD RANDOMIZE
2 BASE->R HEX
3 : FDUP SP@ DUP 2- SWAP 6 + DO I @ -2 +LOOP ;
4 : FDROP DROP DROP DROP DROP ;
5 : FOVER SP@ DUP 6 + SWAP E + DO I @ -2 +LOOP ;
6 : FSWAP FOVER >R >R >R >R >R >R >R >R ;
7 : FDROP R> R> R> R> R> R> R> R> ;
8 : F! 4 0 DO DUP >R ! R> 2+ LOOP DROP ;
9 : F@ 6 + 4 0 DO DUP >R @ R> 2- LOOP DROP ;
10 834A CONSTANT FAC 835C CONSTANT ARG
11 : >FAC FAC F! ; : >ARG ARG F! ; : FAC> FAC F@ ;
12 : SETFL >FAC >ARG ;
13 : FADD 0600 C SYSTEM ; : FSUB 0700 C SYSTEM ;
14 : FMUL 0800 C SYSTEM ; : FDIV 0900 C SYSTEM ;
15 R->BASE -->

```

## SCR #46

```

0 ( FLOATING POINT ARITHMETIC ROUTINES 12JUL82 LCT)
1 BASE->R HEX
2 : F+ SETFL FADD FAC> ;
3 : F- SETFL FSUB FAC> ;
4 : F* SETFL FMUL FAC> ;
5 : F/ SETFL FDIV FAC> ;
6 : S->FAC FAC ! 2300 C SYSTEM ;
7 : FAC->S 1200 C SYSTEM FAC @ ;
8 : FAC>ARG FAC ARG 8 CMOVE ;
9 : F->S >FAC FAC->S ;
10 : S->F S->FAC FAC> ;
11 DECIMAL
12 : FRND 3 0 DO 100 RND 100 RND 256 * + LOOP
13 : 100 RND 16128 + ;
14
15 R->BASE -->

```

## SCR #47

```

0 ( FLOATING POINT CONVERSION ROUTINES CONTINUED 12JUL82 LCT)
1 BASE->R HEX
2 : DOSTR FAC B + C! 14 GPLLNK
3 : FAC B + C@ 8300 + FAC C + C@ DUP PAD C!
4 : PAD 1+ SWAP CMOVE ;
5
6 ( NUMBER IN FAC CONVERTED TO BASIC STRING AND PLACED AT PAD)
7 : STR 0 DOSTR ;
8
9 ( NUMBER IN FAC CONVERTED TO FIXED STRING AND PLACED AT PAD)
10 : STR. FAC D + C! FAC C + C! DOSTR ;
11
12 ( STRING AT PAD CONVERTED TO NUMBER IN FAC)
13 : VAL PAD 1+ 1000 DUP FAC C + ! PAD C@ OVER OVER + 20 SWAP VSBW
14 : VMBW 1000 XMLLNK ;
15 R->BASE -->

```

## SCR #48

```

0 ( FLOATING POINT - COMPILE NO TO STACK 12JUL82 LCT) BASE->R HEX
1 : F$ PAD 1+ SWAP >R R CMOVE R> PAD C! VAL FAC> ;
2 : (>F) R COUNT DUP 1+ =CELLS R> + >R F$ ;
3 : >F 20 STATE @
4   IF   COMPILE (>F) WORD HERE C@
5       1+ =CELLS ALLOT
6   ELSE WORD HERE COUNT F$
7   ENDIF ; IMMEDIATE
8
9 ( FLOATING POINT OUTPUT ROUTINES )
10 : JST PAD C@ - SPACES PAD COUNT TYPE ;
11 : F.R >R >FAC STR R> JST ;
12 : F. 0 F.R ;
13 : FF.R >R >R >R >FAC R> 0 R> STR. R> JST ;
14 : FF. 0 FF.R ;
15 R->BASE -->

```

## SCR #49

```

0 ( FLOATING POINT COMPARE ROUTINES 12JUL82 LCT)
1 BASE->R HEX
2 : FCLEAN >R DROP DROP DROP R> ;
3
4 : F0< 0< FCLEAN ;
5
6 : F0= 0= FCLEAN ;
7
8 : FCOM SETFL 0A00 C SYSTEM 837C C@ ;
9 : F> FCOM 40 AND MINUS 0< ;
10 : F= FCOM 20 AND MINUS 0< ;
11 : F< FCOM 60 AND 0= ;
12 : FLERR 8354 C@ ;
13 : ?FLERR FLERR A ?ERROR ;
14
15 R->BASE -->

```

## SCR #50

```

0 ( FLOATING POINT TRANSCENDENTAL FUNCTIONS 12JUL82 LCT)
1 BASE->R HEX
2 0 VARIABLE LNKSAV
3 : GLNK 83C4 @ LNKSAV ! GPLLNK LNKSAV @ 83C4 ! ;
4 : INT >FAC 22 GLNK FAC> ;
5 : ^ SETFL ARG 836E @ 8 VMBW 24 GLNK FAC> 8 836E +! ;
6 : SQR >FAC 26 GLNK FAC> ;
7 : EXP >FAC 28 GLNK FAC> ;
8 : LOG >FAC 2A GLNK FAC> ;
9 : COS >FAC 2C GLNK FAC> ;
10 : SIN >FAC 2E GLNK FAC> ;
11 : TAN >FAC 30 GLNK FAC> ;
12 : ATN >FAC 32 GLNK FAC> ;
13 : PI >F 3.141592653590 ;
14
15 R->BASE

```



## SCR #51

```

0 ( CONVERT TO TEXT MODE CONFIGURATION 14SEP82 LA0)
1 0 CLOAD TEXT BASE->R DECIMAL 56 R->BASE CLOAD SETVDP2
2 BASE->R  HEX
3
4 : TEXT
5 0 3C0 20 VFILL ( BLANKS TO SCREEN IMAGE AREA )
6 28 SCR_N_WIDTH ! 0 SCR_N_START ! 3C0 SCR_N_END ! 460 PABS !
7 SETVDP1 1 VDPMD !
8 ( NOW SET VDP REGISTERS )
9 1 6 VWTR 0F4 7 VWTR
10 0F0 SETVDP2 ;
11
12
13
14
15 R->BASE

```

## SCR #52

```

0 ( CONVERT TO GRAPHICS MODE CONFIG 14SEP82 LA0)
1 0 CLOAD GRAPHICS BASE->R DECIMAL 56 R->BASE CLOAD SETVDP2
2 BASE->R  HEX
3
4 : GRAPHICS
5 0 300 20 VFILL ( BLANKS TO SCREEN IMAGE AREA ) 300 80 0 VFILL
6 380 20 F4 VFILL
7 20 SCR_N_WIDTH ! 0 SCR_N_START ! 300 SCR_N_END !
8 SETVDP1 2 VDPMD !
9 ( NOW SET VDP REGISTERS )
10 1 6 VWTR 0F4 7 VWTR
11 E0 SETVDP2 ;
12
13
14
15 R->BASE

```

## SCR #53

```

0 ( CONVERT TO MULTI-COLOR MODE CONFIG 14SEP82 LA0)
1 0 CLOAD MULTI BASE->R DECIMAL 56 R->BASE CLOAD SETVDP2
2 BASE->R  HEX
3
4 : MULTI 0B0 1 VWTR ( BLANK THE SCREEN )
5 -1 18 0 DO I 4 / 0FF SWAP DO 1+ I OVER VSBW 8 +LOOP LOOP DROP
6 800 800 0 VFILL ( INIT 256 CHAR PATTERNS TO 0 )
7 300 80 0 VFILL 380 20 0F4 VFILL
8 20 SCR_N_WIDTH ! 0 SCR_N_START ! 300 SCR_N_END ! 460 PABS !
9 1000 DISK_BUF ! ( RESTORE USER VARIABLES )
10 3 VDPMD !
11 ( NOW SET VDP REGISTERS )
12 4 6 VWTR 11 7 VWTR
13 0EB SETVDP2 ;
14
15 R->BASE

```

## SCR #54

```

0 ( CONVERT TO GRAPHICS2 MODE CONFIG 14SEP82 LA0)
1 0 CLOAD GRAPHICS2 BASE->R DECIMAL 56 R->BASE CLOAD SETVDP2
2 BASE->R HEX : GRAPHICS2 0A0 1 VWTR
3 -1 1B00 1800 D0 1+ DUP 0FF AND I VSBW LOOP DROP
4 1 PABS @ VSBW 16 PABS @ 1+ VSBW 1 ( #FILE) 834C C! PABS @ 8356 !
5 0A 0E SYSTEM ( SUBROUTINE TYPE DSRLNK TO SET 2 DISK BUFFERS )
6 0 1800 0F0 VFILL ( INIT COLOR TABLE )
7 2000 1800 0 VFILL ( INIT BIT MAP )
8 20 SCR_N_WIDTH ! 1800 SCR_N_START ! 1B00 SCR_N_END ! 1B00 PABS !
9 1C00 DISK_BUF ! ( USER VARIABLES NOW SET UP )
10 2 0 VWTR      6 2 VWTR ( SET VDP REGISTERS )
11 07F 3 VWTR    0FF 4 VWTR
12 70 5 VWTR     7 6 VWTR
13 0F1 7 VWTR    0E0 DUP 83D4 C! 1 VWTR      1BC0 836E ! ( VSPTR )
14 0 0 GOTOXY 4 VDPMD E ! 0 837A C! ;
15 R->BASE

```

## SCR #55

```

0 ( CONVERT TO SPLIT MODE CONFIG 14SEP82 LA0)
1 0 CLOAD SPLIT BASE->R DECIMAL 56 R->BASE CLOAD SETVDP2
2 BASE->R DECIMAL 54 R->BASE CLOAD GRAPHICS2
3 BASE->R HEX
4 : SPLIT GRAPHICS2 1A00 SCR_N_START ! 0A0 1 VWTR 3000 800 0FF
5 VFILL 3100 834A ! 18 GPLLNK 3300 834A ! 4A GPLLNK
6 1A00 100 20 VFILL 1000 800 0F4 VFILL 0 0 GOTOXY 0E0 1 VWTR
7 5 VDPMD E ! 0 837A C! ;
8
9 : SPLIT2 GRAPHICS2 1880 SCR_N_END ! 2000 400 0FF VFILL
10 2100 834A ! 18 GPLLNK 2300 834A ! 4A GPLLNK
11 1800 80 20 VFILL 0 400 0F4 VFILL 0 0 GOTOXY 6 VDPMD E !
12 0 837A C! ;
13
14
15 R->BASE

```

## SCR #56

```

0 ( VDPMODES 14SEP82 LA0 ) 0 CLOAD SETVDP2 BASE->R DECIMAL 33
1 R->BASE CLOAD RANDOMIZE BASE->R HEX
2 : SETVDP1 0B0 1 VWTR ( BLANK THE SCREEN )
3 800 800 0FF VFILL ( INIT 256 CHAR PATTERNS TO FF )
4 900 834A ! 18 GPLLNK ( LOAD CAPITAL LETTERS )
5 B00 834A ! 4A GPLLNK ( LOAD LOWER CASE -ON 99/4A ONLY ) ;
6 : SETVDP2 ( n --- ) 460 PABS !
7 1000 DISK_BUF ! ( RESTORE USER VARIABLES )
8 ( SET VDP REGISTERS )
9 0 0 VWTR 0 2 VWTR 0E 3 VWTR
10 1 4 VWTR 6 5 VWTR
11 3E0 836E ! ( VSPTR )
12 1 PABS @ VSBW 16 PABS @ 1+ VSBW 3 ( #FILE) 834C C! PABS @ 8356 !
13 0A 0E SYSTEM ( SUB TYPE DSRLNK TO SET 3 DISK BUF )
14 0 0 GOTOXY 0 837A C!
15 DUP 83D4 C! 1 VWTR ; R->BASE

```

## SCR #57

```

0 ( GRAPHICS PRIMITIVES 12JUL82 LCT) 0 CLOAD CHAR BASE->R DECIMAL
1 33 R->BASE CLOAD RANDOMIZE BASE->R DECIMAL 74 R->BASE CLOAD
2 ;CODE BASE->R HEX
3 380 CONSTANT COLTAB 300 CONSTANT SATR 780 CONSTANT SMTN
4 800 CONSTANT PDT 800 CONSTANT SPDTAB
5 : CHAR ( W1 W2 W3 W4 CH --- )
6 8 * PDT + >R -2 6 DO PAD I + ! -2 +LOOP PAD R> 8 VMBW ;
7 : CHARPAT ( CH --- W1 W2 W3 W4 )
8 8 * PDT + PAD 8 VMBR 8 0 DO PAD I + @ 2 +LOOP ;
9 : VCHAR ( X Y CNT CH --- )
10 >R >R SCR_N_WIDTH @ * + SCR_N_END @ SCR_N_START @ - SWAP
11 R> R> SWAP 0 DO SWAP OVER OVER SCR_N_START @ + VSBW SCR_N_WIDTH
12 @ + ROT OVER OVER /MOD IF 1+ SCR_N_WIDTH @ OVER OVER = IF -
13 ELSE DROP ENDIF ENDIF ROT DROP ROT LOOP DROP DROP DROP ;
14 R->BASE -->
15

```

## SCR #58

```

0 ( GRAPHICS PRIMITIVES 20OCT83 LA0) BASE->R HEX
1 : HCHAR ( X Y CNT CH --- )
2 >R >R SCR_N_WIDTH @ * + SCR_N_START @ + R> R> VFILL ;
3 : COLOR ( FG BG CHSET --- )>R SWAP 10 * + R> COLTAB + VSBW ;
4 : SCREEN ( COLOR --- ) 7 VWTR ;
5 : GCHAR ( X Y --- ASCII ) ( COLUMNS AND ROWS NUMBERED FROM 0 )
6 SCR_N_WIDTH @ * + SCR_N_START @ + VSBW ;
7 : SSDT ( ADDR --- ) ( SET SPRITE DESCRIPTOR TABLE ADDRESS )
8 DUP ' SPDTAB ! 800 / 6 VWTR ( RESET VDP REG 6 )
9 VDPMD @ 4 < IF SMTN 80 0 VFILL 300 ' SATR ! ENDIF
10 SATR 20 0 DO DUP >R D000 SP@ R> 2 VMBW DROP 4 + LOOP DROP
11 ( INIT ALL SPRITES ) ;
12 : SPCHAR ( W1 W2 W3 W4 CH# --- )
13 8 * SPDTAB + >R -2 6 DO PAD I + ! -2 +LOOP PAD R> 8 VMBW ;
14 : SPRCOL ( COL # --- ) 4 * SATR 3 + + DUP >R VSBW 0F0 AND OR
15 R> VSBW ; R->BASE -->

```

## SCR #59

```

0 ( GRAPHICS PRIMITIVES 20OCT83 LCT)
1 BASE->R HEX
2 : SPRPAT ( CH # --- ) 4 * SATR 2+ + VSBW ;
3 : SPRPUT ( DX DY # --- )
4 4 * SATR + >R 1- 100 U* DROP + SP@ R> 2 VMBW DROP ;
5 : SPRITE ( DX DY COL CH # --- ) ( SPRITES NUMBERED 0 - 31 )
6 DUP 4 * SATR + >R DUP >R SPRPAT R SPRCOL R> SPRPUT R> 4 +
7 SATR DO I VSBW D0 = IF C001 SP@ I 2 VMBW DROP ENDIF 4 +LOOP ;
8 : MOTION ( SPX SPY # --- )
9 4 * SMTN + >R 8 SLA SWAP 00FE AND OR SP@ R> 2 VMBW DROP ;
10 : #MOTION ( NO --- ) 837A C! ;
11 : SPRGET ( # --- DX DY )
12 4 * SATR + DUP VSBW 1+ 0FF AND SWAP 1+ VSBW SWAP ;
13 : DXY ( X2 Y2 X1 Y1 --- X^2 Y^2 )
14 ROT - ABS ROT ROT - ABS DUP * SWAP DUP * ;
15 R->BASE -->

```

## SCR #60

```

0 ( GRAPHICS PRIMITIVES 12JUL82 LCT)
1 BASE->R HEX : BEEP 34 GPLLNK ;      : HONK 36 GPLLNK ;
2 : SPRDIST ( #1 #2 --- DIST^2 ) ( DISTANCE BETWEEN 2 SPRITES )
3   SPRGET ROT SPRGET DXY OVER OVER
4   + DUP >R OR OR 8000 AND IF R> DROP 7FFF ELSE R> ENDIF ;
5 : SPRDISTXY ( X Y # --- DIST^2 ) SPRGET DXY OVER OVER
6   + DUP >R OR OR 8000 AND IF R> DROP 7FFF ELSE R> ENDIF ;
7 : MAGNIFY ( MAG-FACTOR --- )
8   83D4 C@ 0FC AND + DUP 83D4 C! 1 VWTR ;
9 : JOYST ( KEYBDNO --- ASCII XSTAT YSTAT ) 8374 C!
10 ?KEY DROP 8375 C@ DUP DUP 12 = IF DROP 0 0 ELSE 0FF =
11 IF 8377 C@ 8376 C@ ELSE 8375 C@
12 CASE 4 OF 0FC 4   ENDOF 5 OF 0 4 ENDOF 6 OF 4 4 ENDOF
13     2 OF 0FC 0   ENDOF 3 OF 4 0 ENDOF 0 OF 0 0FC ENDOF
14     0F OF 0FC 0FC ENDOF 0E OF 4 0FC ENDOF DROP DROP 0 0 0 0
15 ENDCASE ENDIF ENDIF 4 8374 C! ; R->BASE -->

```

## SCR #61

```

0 ( GRAPHICS PRIMITIVES 12JUL82 LCT) BASE->R HEX
1 : COINC ( #1 #2 TOL --- F ) ( 0= NO COINC 1= COINC )
2   DUP * DUP + >R SPRDIST R> > 0= ;
3 : COINCXY ( DX DY # TOL --- F )
4   DUP * DUP + >R SPRDISTXY R> > 0= ;
5 : COINCALL ( --- F ) ( BIT SET IF ANY TWO SPRITES OVERLAP )
6   8802 C@ 20 AND 20 = ;
7 : DELSPR ( # --- )
8   4 * DUP SATR + >R 0 C001 SP@ R> 4 VMBW DROP DROP
9   SMTN + >R 0 0 SP@ R> 4 VMBW DROP DROP ;
10 : DELALL ( --- )
11 0 #MOTION SATR 20 0 DO DUP D0 SWAP VSBW 4 + LOOP DROP
12 SMTN 80 0 VFILL ;
13
14
15 R->BASE -->

```

## SCR #62

```

0 ( GRAPHICS PRIMITIVES 24NOV82 LA0) BASE->R HEX 0 VARIABLE ADR
1 : MINIT 18 0 DO 0 I 4 / 20 * DUP 20 + SWAP
2   DO DUP J 1 I HCHAR 1+ LOOP DROP LOOP ;
3 : MCHAR ( COLOR C R --- ) DUP >R 2 / SWAP DUP >R 2 / SWAP
4   DUP >R GCHAR DUP 20 / 100 U* DROP 800 + >R 20 MOD
5   8 * R> + R> 4 MOD 2 * + ADR ! R> 2 MOD R> 2 MOD SWAP
6   IF IF 3 ELSE 1 ENDIF ELSE IF 2 ELSE 0 ENDIF ENDIF
7   DUP 2 MOD 0= IF SWAP 10 * SWAP ENDIF
8   CASE 0 OF ADR @ VSBW 0F ENDOF 1 OF ADR @ VSBW F0 ENDOF
9   2 OF 1 ADR +! ADR @ VSBW 0F ENDOF
10  3 OF 1 ADR +! ADR @ VSBW F0 ENDOF
11 ENDCASE AND + ADR @ VSBW ;
12 0 VARIABLE DMODE -1 VARIABLE DCOLOR
13 : DRAW 0 DMODE ! ; : UNDRAW 1 DMODE ! ; : DT0G 2 DMODE ! ;
14 8040 VARIABLE DTAB 2010 , 804 , 201 , 7FBF , DFEF , F7FB ,
15 FDFE , 8040 , 2010 , 804 , 201 , R->BASE -->

```

## SCR #63

```

0 ( GRAPHICS PRIMITIVES ) BASE->R HEX
1 CODE DDOT C079 ,
2 C0D9 , C081 , C103 , 0241 ,
3 0007 , 0243 , 0007 , 0242 ,
4 00F8 , 0244 , 00F8 , 0A52 ,
5 A042 , A044 , 0221 , 2000 ,
6 04C4 , D123 , DTAB , 06C4 ,
7 C644 , 0649 , C641 , 045F ,
8 : DOT ( X Y --- )
9 DDOT DUP 2000 - >R DMODE @
10 CASE 0 OF VOR ENDOF ( DRAW )
11 1 OF SWAP FF XOR SWAP VAND ENDOF ( UNDRAW )
12 2 OF VXOR ENDOF ( TOGGLE )
13 DROP DROP ENDCASE R>
14 DCOLOR @ 0 < IF DROP ELSE DCOLOR @ SWAP VSBW ENDIF ;
15 R->BASE -->

```

## SCR #64

```

0 ( GRAPHICS PRIMITIVES 12JUL82 LCT) BASE->R HEX
1 : SGN DUP IF DUP 0< IF -1 ELSE 1 ENDIF ELSE 0 ENDIF + ;
2 : LINE >R R ROT >R R - SGN SWAP >R R ROT >R R - SGN OVER ABS
3 OVER ABS < >R R 0= IF SWAP ENDOF 100 ROT ROT */ R>
4 IF ( X AXIS ) R> R> OVER OVER >
5 IF ( MAKE L TO R ) SWAP R> DROP R>
6 ELSE R> R> DROP
7 ENDOF 100 * ROT ROT 1+ SWAP
8 DO I OVER 0 100 M/ SWAP DROP DOT OVER + LOOP
9 ELSE ( Y AXIS ) R> R> R> ROT >R ROT >R OVER OVER >
10 IF ( MAKE T TO B ) SWAP R> DROP R>
11 ELSE R> R> DROP
12 ENDOF 100 * ROT ROT 1+ SWAP
13 DO DUP 0 100 M/ SWAP DROP I DOT OVER + LOOP
14 ENDOF DROP DROP ;
15 R->BASE

```

## SCR #65

```

0 ( COMPACT LIST )
1 0 CLOAD SMASH BASE->R DECIMAL 74 R->BASE CLOAD ;CODE
2 BASE->R DECIMAL 33 R->BASE CLOAD RANDOMIZE BASE->R DECIMAL
3 0 VARIABLE TCHAR 382 ALLOT 67 BLOCK TCHAR 384 CMOVE HEX
4 TCHAR 7C - CONSTANT TC 0 VARIABLE BADDR 0 VARIABLE INDX
5 ( SMASH EXPECTS ADDR #CHAR LINE# --- LB VADDR CNT )
6 0 VARIABLE LB FE ALLOT
7 CODE SMASH
8 C079 , C0B9 , C0D9 , 0204 , LB , C644 , 0649 , 06C1 ,
9 0221 , 2000 , C641 , C042 , 0581 , 0241 , FFFE , 0649 ,
10 0A21 , C641 , A083 , 80C2 , 1501 , 1020 , 04C5 , 04C6 ,
11 D173 , D1B3 , 0965 , 0966 , C025 , TC , C066 , TC ,
12 0B41 , 020C , 0004 , C2C0 , 024B , F000 , C1C1 , 0247 ,
13 0F00 , E1CB , DD07 , 0BC0 , 0BC1 , 060C , 16F4 , 05C5 ,
14 05C6 , C305 , 024C , 0002 , 16E7 , 10DD , 045F ,
15 R->BASE -->

```

## SCR #66

```

0 ( COMPACT LIST ) BASE->R DECIMAL
1 : CLINE LB 100 ERASE SMASH VMBW ;
2 : CLOOP DO I 64 * OVER + 64 I CLINE LOOP DROP ;
3
4 : CLIST BLOCK 16 0 CLOOP ;
5
6
7
8
9
10
11
12
13
14 R->BASE
15

```

## SCR #68

```

0 ( FILE I/O ROUTINES 12JUL82 LCT)
1 0 CLOAD STAT BASE->R DECIMAL 33 R->BASE CLOAD RANDOMIZE
2 BASE->R HEX
3 0 VARIABLE PAB-ADDR
4 0 VARIABLE PAB-BUF
5 0 VARIABLE PAB-VBUF
6 : FILE <BUILDS , , , DOES> DUP @ PAB-VBUF ! 2+ DUP @ PAB-BUF !
7   2+ @ PAB-ADDR ! ;
8 : GET-FLAG PAB-ADDR @ 1+ VSBW ;
9 : PUT-FLAG PAB-ADDR @ 1+ VSBW ;
10 : SET-PAB PAB-ADDR @ DUP 0A 0 VFILL 2+ PAB-VBUF SWAP 2 VMBW ;
11 : CLR-STAT GET-FLAG 1F AND PUT-FLAG ;
12 : CHK-STAT GET-FLAG 0E0 AND
13   837C C@ 20 AND OR 9 ?ERROR ;
14 : FXD GET-FLAG 0EF AND PUT-FLAG ;
15 : VRBL GET-FLAG 10 OR PUT-FLAG ; R->BASE -->

```

## SCR #69

```

0 ( FILE I/O ROUTINES 12JUL82 LCT) BASE->R HEX
1 : DSPLY GET-FLAG 0F7 AND PUT-FLAG ;
2 : INTRNL GET-FLAG 8 OR PUT-FLAG ;
3 : I/OMD GET-FLAG 0F9 AND ;
4 : INPT I/OMD 4 OR PUT-FLAG ;
5 : OUTPT I/OMD 2 OR PUT-FLAG ;
6 : UPDT I/OMD PUT-FLAG ;
7 : APPND I/OMD 6 OR PUT-FLAG ;
8 : SQNTL GET-FLAG 0FE AND PUT-FLAG ;
9 : RLTV GET-FLAG 1 OR PUT-FLAG ;
10 : REC-LEN PAB-ADDR @ 4 + VSBW ;
11 : CHAR-CNT! PAB-ADDR @ 5 + VSBW ;
12 : CHAR-CNT@ PAB-ADDR @ 5 + VSBR ;
13 : REC-NO DUP SWPB PAB-ADDR @ 6 + VSBW PAB-ADDR @ 7 + VSBW ;
14 : N-LEN! PAB-ADDR @ 9 + VSBW ;
15 R->BASE -->

```

## SCR #70

```

0 ( FILE I/O ROUTINES 12JUL82 LCT) BASE->R HEX
1 ( COMPILE A STRING WHICH IS MOVED TO VDP-ADDR AT EXECUTION)
2
3 : (F-D")
4   PAB-ADDR @ 0A + R COUNT DUP 1+ =CELLS R> +
5   >R >R SWAP R VMBW R> N-LEN! ;
6 : F-D" 22 STATE @
7   IF
8     COMPILE (F-D") WORD HERE C@
9     1+ =CELLS ALLLOT
10  ELSE
11    PAB-ADDR @ 0A + SWAP WORD HERE COUNT >R SWAP R
12    VMBW R> N-LEN!
13    ENDIF ; IMMEDIATE
14
15 R->BASE -->

```

## SCR #71

```

0 ( FILE I/O ROUTINES 12JUL82 LCT)
1 BASE->R HEX
2 : DOI/O CLR-STAT PAB-ADDR @ VSBW PAB-ADDR @ 9 + 8356 !
3   0 837C C! DSRLNK CHK-STAT ;
4 : OPN 0 DOI/O ;
5 : CLSE 1 DOI/O ;
6 : RD 2 DOI/O PAB-VBUF @ PAB-BUF @ CHAR-CNT@ VMBR CHAR-CNT@ ;
7 : WRT >R PAB-BUF @ PAB-VBUF @ R VMBW R> CHAR-CNT! 3 DOI/O ;
8 : RSTR REC-NO 4 DOI/O ;
9 : LD REC-NO 5 DOI/O ;
10 : SV REC-NO 6 DOI/O ;
11 : DLT 7 DOI/O ;
12 : SCRTCH REC-NO 8 DOI/O ;
13 : STAT 9 DOI/O PAB-ADDR @ 8 + VSBR ;
14
15 R->BASE

```

## SCR #72

```

0 ( ALTERNATE I/O SUPPORT FOR RS232 PNTR 12JUL82 LCT)
1 0 CLOAD INDEX      BASE->R DECIMAL 68 R->BASE CLOAD STAT
2 0 0 0 FILE >RS232  BASE->R HEX
3 : SWCH >RS232 PABS @ 10 + DUP PAB-ADDR ! 1- PAB-VBUF !
4 SET-PAB OUTPT F-D" RS232.BA=9600"          OPN 3
5 PAB-ADDR @ VSBW 1 PAB-ADDR @ 5 + VSBW PAB-ADDR @ ALTOUT ! ;
6 : UNSWCH 0 ALTOUT ! CLSE ;
7 : ?ASCII ( BLOCK# --- FLAG )
8     BLOCK 0 SWAP DUP 400 + SWAP
9     DO I C@ 20 > + I C@ DUP 20 < SWAP 7F > OR
10    IF DROP 0 LEAVE ENDIF LOOP ;
11 : TRIAD 0 SWAP SWCH 3 / 3 * DUP 3 + SWAP
12 DO I ?ASCII IF 1+ I LIST CR ENDIF LOOP
13 -DUP IF 3 SWAP - 14 * 0 DO CR LOOP
14 0F MESSAGE 0C EMIT  ENDIF UNSWCH ;
15 R->BASE  -->

```

## SCR #73

```

0 ( SMART TRIADS AND INDEX 15SEP82 LA0 ) BASE->R DECIMAL
1 : TRIADS ( FROM TO --- )
2 3 / 3 * 1 + SWAP 3 / 3 * DO I TRIAD 3 +LOOP ;
3 : INDEX ( FROM TO --- ) 1+ SWAP
4 DO I DUP ?ASCII IF CR 4 .R 2 SPACES I BLOCK 64 TYPE ELSE DROP
5     ENDIF PAUSE IF LEAVE ENDIF LOOP ;
6
7
8
9
10
11
12
13
14
15 R->BASE

```

## SCR #74

```

0 ( ASSEMBLER 12JUL82 LCT)
1 FORTH DEFINITIONS
2 0 CLOAD CODE
3
4 VOCABULARY ASSEMBLER IMMEDIATE
5
6 : CODE
7     ?EXEC CREATE SMUDGE LATEST PFA DUP CFA !
8     [COMPILE] ASSEMBLER ;
9
10 : ;CODE
11     ?CSP COMPILE ( ;CODE) SMUDGE
12     [COMPILE] [ [COMPILE] ASSEMBLER ;
13
14
15

```



## SCR #75

```

0 ( ASSEMBLER 12JUL82 LCT)  0 CLOAD A$$M
1 BASE->R DECIMAL 74 R->BASE CLOAD ;CODE
2 BASE->R HEX
3 ASSEMBLER DEFINITIONS
4 : GOP' OVER DUP 1F > SWAP 30 < AND
5     IF + , , ELSE + , ENDIF ;
6 : GOP <BUILDS , DOES> @ GOP' ;
7 0440 GOP B,      0680 GOP BL,      0400 GOP BLWP,
8 04C0 GOP CLR,    0700 GOP SET0,    0540 GOP INV,
9 0500 GOP NEG,    0740 GOP ABS,     06C0 GOP SWPB,
10 0580 GOP INC,   05C0 GOP INCT,    0600 GOP DEC,
11 0640 GOP DECT,  0480 GOP X,
12 : GROU <BUILDS , DOES> @ SWAP 40 * + GOP' ;
13 2000 GROU COC,  2400 GROU CZC,    2800 GROU XOR,
14 3800 GROU MPY,  3C00 GROU DIV,    2C00 GROU XOP,
15 -->

```

## SCR #76

```

0 ( ASSEMBLER 12JUL82 LCT)
1 : GGOP <BUILDS ,
2     DOES> @ SWAP DUP DUP 1F > SWAP 30 < AND
3     IF 40 * + SWAP >R GOP' R> ,
4     ELSE 40 * + GOP' ENDIF ;
5 A000 GGOP A,    B000 GGOP AB,
6 8000 GGOP C,    9000 GGOP CB,
7 6000 GGOP S,    7000 GGOP SB,
8 E000 GGOP SOC,  F000 GGOP SOCB,
9 4000 GGOP SZC,  5000 GGOP SZCB,
10 C000 GGOP MOV, D000 GGOP MOVB,
11
12 : 00P <BUILDS , DOES> @ , ;
13 0340 00P IDLE, 0360 00P RSET, 03C0 00P CKOF,
14 03A0 00P CKON, 03E0 00P LREX, 0380 00P RTWP,
15 -->

```

## SCR #77

```

0 ( ASSEMBLER 12JUL82 LCT)
1
2 : ROP <BUILDS , DOES> @ + , ;
3
4 02C0 ROP STST, 02A0 ROP STWP,
5
6 : IOP <BUILDS , DOES> @ , , ;
7
8 02E0 IOP LWPI, 0300 IOP LIM1,
9
10 : RIOP <BUILDS , DOES> @ ROT + , , ;
11
12 0220 RIOP AI,   0240 RIOP ANDI,
13 0280 RIOP CI,   0200 RIOP LI,
14 0260 RIOP ORI,
15 -->

```

## SCR #78

```

0 ( ASSEMBLER 12JUL82 LCT)
1 : RCOP <BUILDS , DOES> @ SWAP 10 * + + , ;
2 0A00 RCOP SLA, 0800 RCOP SRA,
3 0B00 RCOP SRC, 0900 RCOP SRL,
4 : DOP <BUILDS , DOES> @ SWAP 00FF AND OR , ;
5 1300 DOP JEQ, 1500 DOP JGT,
6 1B00 DOP JH, 1400 DOP JHE,
7 1A00 DOP JL, 1200 DOP JLE,
8 1100 DOP JLT, 1000 DOP JMP,
9 1700 DOP JNC, 1600 DOP JNE,
10 1900 DOP JNO, 1800 DOP JOC,
11 1C00 DOP JOP, 1D00 DOP SBO,
12 1E00 DOP SBZ, 1F00 DOP TB,
13 : GCOP <BUILDS , DOES> @ SWAP 000F AND 040 * + GOP' ;
14 3000 GCOP LDCR, 3400 GCOP STCR,
15 -->

```

## SCR #79

```

0 ( ASSEMBLER 12JUL82 LCT)
1 : @() 020 ; : *? 010 + ;
2 : *?+ 030 + ; : @(?) 020 + ;
3 : W 0A ; : @ (W) W @ (?) ;
4 : *W W *? ; : *W+ W *?+ ;
5 : RP 0E ; : @ (RP) RP @ (?) ;
6 : *RP RP *? ; : *RP+ RP *?+ ;
7 : IP 0D ; : @ (IP) IP @ (?) ;
8 : *IP IP *? ; : *IP+ IP *?+ ;
9 : SP 09 ; : @ (SP) SP @ (?) ;
10 : *SP SP *? ; : *SP+ SP *?+ ;
11 : UP 08 ; : @ (UP) UP @ (?) ;
12 : *UP UP *? ; : *UP+ UP *?+ ;
13 : NEXT 0F ; : *NEXT+ NEXT *?+ ;
14 : *NEXT NEXT *? ; : @ (NEXT) NEXT @ (?) ;
15 -->

```

## SCR #80

```

0 ( ASSEMBLER 12JUL82 LCT)
1 ( DEFINE JUMP TOKENS )
2 : GTE 1 ; : H 2 ; : NE 3 ;
3 : L 4 ; : LTE 5 ; : EQ 6 ;
4 : OC 7 ; : NC 8 ; : OO 9 ;
5 : HE 0A ; : LE 0B ; : NP 0C ;
6 : LT 0D ; : GT 0E ; : NO 0F ;
7 : OP 10 ;
8 : CJMP ?EXEC
9 CASE LT OF 1101 , 0 ENDOF
10 GT OF 1501 , 0 ENDOF
11 NO OF 1901 , 0 ENDOF
12 OP OF 1C01 , 0 ENDOF
13 DUP 0< OVER 10 > OR IF 19 ERROR ENDIF DUP
14 ENDCASE 100 * 1000 + , ;
15 -->

```

## SCR #81

```

0 ( ASSEMBLER 12JUL82 LCT)
1 : IF,      ?EXEC
2   [COMPILE] CJMP HERE 2- 42 ; IMMEDIATE
3 : ENDIF,   ?EXEC
4   42 ?PAIRS HERE OVER - 2- 2 / SWAP 1+ C! ; IMMEDIATE
5 : ELSE,    ?EXEC
6   42 ?PAIRS 0 [COMPILE] CJMP HERE 2- SWAP 42 [COMPILE]
7   ENDIF, 42 ; IMMEDIATE
8 : BEGIN,   ?EXEC
9   HERE 41 ; IMMEDIATE
10 : UNTIL,   ?EXEC
11   SWAP 41 ?PAIRS [COMPILE] CJMP HERE - 2 / 00FF AND
12   HERE 1- C! ; IMMEDIATE
13 : AGAIN,   ?EXEC
14   0 [COMPILE] UNTIL, ; IMMEDIATE
15 -->

```

## SCR #82

```

0 ( ASSEMBLER 12JUL82 LCT)
1 : REPEAT,   ?EXEC
2   >R >R [COMPILE] AGAIN, R> R> 2- [COMPILE] ENDIF,
3   ; IMMEDIATE
4 : WHILE,    ?EXEC
5   [COMPILE] IF, 2+ ; IMMEDIATE
6
7
8
9
10 : NEXT, *NEXT B, ;
11
12 FORTH DEFINITIONS
13
14 : A$$M ;          R->BASE
15

```

## SCR #83

```

0 ( BSAVE -- BINARY SAVER FOR FORTH OVERLAYS      LCT 14SEP82 )
1 0 CLOAD BSAVE    BASE->R DECIMAL
2 : BSAVE ( from scrn-no --- ) FLUSH
3   BEGIN
4     SWAP >R DUP 1+ SWAP
5     OFFSET @ + BUFFER UPDATE DUP B/BUF ERASE
6     R OVER ! 2+ HERE OVER ! 2+
7     CURRENT @ OVER ! 2+          LATEST      OVER ! 2+
8     CONTEXT @ OVER ! 2+          CONTEXT @ @ OVER ! 2+
9     VOC-LINK @ OVER ! 2 +    29801 OVER ! 10 +
10    HERE R -
11    R> DUP 1000 + >R SWAP >R SWAP R>
12    1000 MIN CMOVE
13    R SWAP HERE R> <
14    UNTIL
15    SWAP DROP FLUSH    ; R->BASE

```

## SCR #84

```

0 ( NEW MESSAGE ROUTINE 13SEP82 LCT )    BASE->R DECIMAL
1
2 ( THIS VERSION OF MESSAGE HAS THE SCREEN 4 AND 5 MESSAGES
3   INCLUDED IN THIS ROUTINE. )
4
5 FLUSH EMPTY-BUFFERS HERE LIMIT$ @ B/BUF 4 + - DUP LIMIT$ !
6 DP ! ( PLACES message WHERE 5TH DISK BUF IS.  NOW HAVE 4 BUFS )
7 : message
8     WARNING @
9     IF
10    -DUP
11    IF ( NON-ZERO MESSAGE NUMBER )
12        DUP 26 <
13        IF ( MESSAGE NEED NOT BE RETRIEVED FROM DISK )
14            CASE ( FOLLOWING CASES FOR MESSAGE NUMBERS )
15 -->

```

## SCR #85

```

0 ( NEW MESSAGE CONTINUED )
1 01 OF ." empty stack"                ENDOF
2 02 OF ." dictionary full"            ENDOF
3 03 OF ." has incorrect address mode"  ENDOF
4 04 OF ." isn't unique."              ENDOF
5
6 06 OF ." disk error"                 ENDOF
7 07 OF ." full stack"                 ENDOF
8
9 09 OF ." file i/o error"             ENDOF
10 10 OF ." floating point error"       ENDOF
11 11 OF ." disk fence violation"       ENDOF
12 12 OF ." can't load from screen zero" ENDOF
13
14
15 15 OF ." TI FORTH --- a fig-FORTH extension" ENDOF -->

```

## SCR #86

```

0 ( NEW MESSAGE CONTINUED )
1 17 OF ." compilation only, use in definition" ENDOF
2 18 OF ." execution only"                    ENDOF
3 19 OF ." conditionals not paired"           ENDOF
4 20 OF ." definition not finished"           ENDOF
5 21 OF ." in protected dictionary"           ENDOF
6 22 OF ." use only when loading"             ENDOF
7
8 24 OF ." declare vocabulary"                ENDOF
9 25 OF ." bad jump token"                    ENDOF
10
11 ENDCASE
12
13 -->
14
15

```

## SCR #87

```

0 ( NEW MESSAGE CONTINUED )
1
2         ELSE
3         4 OFFSET @ B/SCR / - .LINE
4         ENDIF
5         ENDIF
6         ELSE
7         ." MSG # " .
8         ENDIF
9 ;
10
11 DP ! ( RESTORE DP TO POSITION PRIOR TO message )
12 ( INSTALL NEW MESSAGE )
13 ' BRANCH CFA      ' MESSAGE
14 ' message OVER - 2- OVER 2+ ! !
15 R->BASE

```

## SCR #88

```

0 ( CRU WORDS 120CT82 LA0 ) 0 CLOAD STCR
1 BASE->R DECIMAL 74 R->BASE CLOAD ;CODE
2 BASE->R HEX
3 CODE SB0 C339 , A30C , 1D00 , 045F ,
4 CODE SBZ C339 , A30C , 1E00 , 045F ,
5 CODE TB C319 , A30C , 04D9 , 1F00 , 1601 , 0599 , 045F ,
6
7 CODE LDCR C339 , A30C , C079 , C039 , 0241 , 000F , 1304 ,
8          0281 , 0008 , 1501 , 06C0 , 0A61 , 0261 , 3000 ,
9          0481 , 045F ,
10
11 CODE STCR C339 , A30C , C059 , 04C0 , 0241 , 000F , C081 ,
12          0A61 , 0261 , 3400 , 0481 , C082 , 1304 , 0282 ,
13          0008 , 1501 , 06C0 , C640 , 045F ,
14
15 R->BASE

```

## Appendix J TI Forth Bugs

TI Forth Bugs found as of the May, 1985 issue of HOCUS (Milwaukee TI Users Group):

Jeff Stanford—

Screen 22, Line 5:

**BASE->R HEX ( 3800 ' SATR ! )**

Screen 23, Line 2:

**: CINIT 3800 DUP ' SPDTAB ! 800 / 6 VWTR 3800 ' SATR !**

Screen 28, Line 1

**: EDT VDPMD @ 5 = 0= IF SPLIT ENDIF CINIT !CUR R/C CGOTOXY**

Screen 28, Line 11:

**0F 0F 5 0 SPRPAT CLS SCRNO DROP 300 ' SATR ! QUIT ENDOF**

Tom Freeman—

Screens 53 – 55, Line 1 on each screen:

Change **VDPSET2** to **SETVDP2**

Screen 58:

Switch Lines 9 & 10

In new Line 9, change **300 ! SATR** to **300 ' SATR**

Screen 59, Line 9, between **SWAP** and **AND**

Change **00FF** to **00FE**

Everybody & his brother—

Screen 72 Line 5:

**PAB\_ADDR** to **PAB-ADDR**

Jim Vincent—

Original Manual, Chapter 6, Page 10, Line 1 (see this manual, footnote 10, page 37):

**HEX 3800 ' SATR !**

Original Manual, Chapter 10, Page 3, Line 20 (see this manual, footnote 17, page 70):

**: DOWN -100 ALLOT DROP ;**

See also the corrections and clarifications in the Editor's Notes in Appendix G .

## Appendix K Diskette Format Details

The information in this section is based on TI's *Software Specifications for the 99/4 Disk Peripheral (March 28, 1983)*.

The original disk drives supplied by TI supported only single-sided, single-density (SSSD), 90-KB diskettes. The TI Forth system was designed around and supplied in this disk format. Though different formats are possible, we will consider the usual format of 256 bytes per sector and 40 tracks per side. The following table shows possible formats with 256 bytes/sector and 40 tracks/side:

Disk Type	Sides	Density	Sectors/Track	Total Sectors	Capacity
SSSD	1	single	9	360	90 KB
DSSD	2	single	9	720	180 KB
SSDD	1	double	18	720	180 KB
DSDD	2	double	18	1440	360 KB
Compact Flash <sup>19</sup>	2	double	20	1600 <sup>20</sup>	400 KB

The information in the following sections accrues to all the above formats:

### K.1 Volume Information Block (VIB)

Byte #	1 <sup>st</sup> Byte	2 <sup>nd</sup> Byte	Byte #
0	Disk Volume Name (10 characters padded on the right with blanks)		1
8			9
10	Total Number of Sectors		11
12	Sectors/Track	"D"	13
14	"S"	"K"	15
16	Protection ("P" or "")	Tracks/Side	17
18	# of Sides	Density	19
20	Reserved		21
54			55
56			57
254	Allocation Bit Map (room for 1600 sectors)		255

<sup>19</sup> This is a third-party peripheral expansion device with 400 KB virtual disks using Compact Flash memory on devices named nanoPEB and CF7+ (see website: <http://webpages.charter.net/nanopeb/>)

<sup>20</sup> 1600 sectors is the maximum possible number of sectors that can be managed by the current specification.

Sector 0 contains the volume information block (VIB). The layout is shown in the above table.

## K.2 File Descriptor Index Record (FDIR)

Sector 1 contains the file descriptor index record (FDIR). It can hold up to 127 2-byte entries, each pointing to a file descriptor record (FDR—see next section). These pointers are alphabetically sorted by the file names to which they point. This list of pointers starts at the beginning of sector 1 and ends with a pointer value of 0.

## K.3 File Descriptor Record (FDR)

Byte #	1 <sup>st</sup> Byte	2 <sup>nd</sup> Byte	Byte #
0	File Name (10 characters padded on the right with blanks)		1
8			9
10	Reserved		11
12	File Status Flags	# of Records/Sector (0 for program)	13
14	# of Sectors currently allocated (not counting this FDR)		15
16	EOF Offset (bytes in last Sector)	Bytes/Record	17
18	# of Records (Fixed) or # of Sectors (Variable)—bytes are in reverse order		19
20	Reserved		21
26			27
28	Data Chain Pointer Blocks (3 bytes/block encoding two 12-bit numbers that indicate cluster start and highest, cumulative sector offset)		29
254			255

There can be as many as 127 file descriptor records (FDRs) laid out as in the above table. There are no subdirectories. FDRs will start in sector 2 and continue, at least, until sector 33, unless a file allocation requires more space than is available in sectors 34 – end-of-disk, in which case the system will begin allocating space for the file in the first available sector in sectors 3 – 33. This is done “to obtain faster directory search response times”<sup>21</sup>. Each FDR beyond 32 files will be placed in the first available sector.

Byte 12 contains file status flags defined as follows, with bit 0 as the least significant bit:

Bit #	Description
0	Program or Data file (0 = Data; 1 = Program)
1	Binary or ASCII data (0 = ASCII, DISPLAY file; 1 = Binary, INTERNAL or program file)
2	Reserved
3	PROTECT flag (0 = not protected; 1 = protected)
4–6	Reserved
7	FIXED/VARIABLE flag (0 = fixed-length records; 1 = variable-length records)

<sup>21</sup> *Software Specifications for the 99/4 Disk Peripheral (March 28, 1983)*, p. 19.



The cluster blocks listed in bytes 28 – 255 of the FDR each contain 2 12-bit (3-nibble<sup>22</sup>) numbers. The first points to the beginning sector of that cluster of contiguous sectors and the second is the sector offset reached by that cluster. If we label the 3 nibbles of the cluster pointer as  $n_1 - n_3$  and the 3 nibbles of the cumulative sector offset as  $m_1 - m_3$ , with the subscripts indicating the significance of the nibble, then the 3 bytes are laid out as follows:

Byte 1:  $n_2n_1$     Byte 2:  $m_1n_3$     Byte 3:  $m_3m_2$

The actual 12-bit numbers, then, are

Cluster Pointer:  $n_3n_2n_1$                       Sector Offset:  $m_3m_2m_1$

For example, the following represents 2 blocks in the FDR for a file with 2 clusters allocated:

Actual layout in the FDR: **4D20h 5F05h F060h**

1<sup>st</sup> Cluster Pointer: **04Dh** ( $77_{10}$ )<sup>23</sup>                      Record Offset: **5F2h** ( $1522_{10}$ )

2<sup>nd</sup> Cluster Pointer: **005h** ( $5_{10}$ )                      Record Offset: **60Fh** ( $1551_{10}$ )

The above example represents a file, the data for which occupies 1552 sectors on the disk. If we assume that no files have been deleted in this case, you should also be able to deduce that there are only 3 files on the disk because the second cluster starts in sector 5 and occupies all sectors from 5 – 33, which should tell you there are 3 FDRs before this cluster was allocated: Sector 0 (VIB), sector 1 (FDIR), sector 2 (FDR of first file), sector 3 (FDR of second file), sector 4 (FDR of third file and sector 5 (second cluster start of the third file, the first two occupying sectors 34 – 76 by inference). Furthermore, the disk contains 1600 sectors because that is the maximum and the first cluster ended in the 1600<sup>th</sup> sector of the disk (1<sup>st</sup> cluster starts in sector 77 and ends 1522 sectors later in sector 1599).<sup>24</sup>

## ***K.4 Comparison of TI Forth and TI File System Layouts on the Same Disk***

The TI file system layout has been detailed earlier in this appendix. The TI Forth system is based on 1-KB blocks or screens that each consist of 16 lines of 64 characters. TI Forth screens start in the first sector of the disk and contiguously occupy the entire disk with each screen consuming 4 contiguous sectors ( $4 \cdot 256 = 1024$  bytes/screen). The TI Forth system reads and writes screens using direct sector access, thus making it possible to easily destroy the normal file-system layout of the system disk and, less so, work disks that have been set up by **DISK-HEAD** or the method of § L.2 if you are not careful. The sections that follow show the two layouts side by side to make it easier to understand the relationship of the 128-byte file records with where they appear on TI Forth screens.

<sup>22</sup> A nibble (also nibble) is half of one byte (8 bits) and is equal to 4 bits. The editor prefers “nibble” to “nibble” because of its obvious relationship to “byte”. 2 nibbles = 1 byte.

<sup>23</sup> The subscript, 10, indicates base 10 (decimal).

<sup>24</sup> This example is taken from one of my (Lee Stewart’s) Compact Flash volumes.

### K.4.1 TI Forth System Disk

The TI Forth system disk of the original 90-KB disk is shown in the following table [Note: A bold line in the body of this table and the next represents the beginning line of a file.]:

Sector	Record Bytes	Record	File Name	Contents (Characters as DOS/IBM ASCII)	Forth Screen	Line
0	256	0	{VIB}	TI/FORTH ©hoDSK .....	0	0
:	:	:	:	:	:	:
1	256	0	{FDIR}	.©.♥.♦.....	0	4
:	:	:	:	:	:	:
2	256	0	{FDR} FORTH	FORTH ...♥.♣.P♪....."@.....	0	8
:	:	:	:	:	:	:
3	256	0	{FDR} FORTHSAVE	FORTHSAVE ..©..&{(.....'P©.....	0	12
:	:	:	:	:	:	:
4	256	0	{FDR} SYS-SCRNS	SYS-SCRNS ...©©8.Çp©.....Má←p!!.....	1	0
:	:	:	:	:	:	:
5	128	566	SYS-SCRNS	.....	1	4
:	:	:	:	:	:	:
8	128	572	SYS-SCRNS	T I F O R T H	2	0
			SYS-SCRNS		2	1
	128	573	SYS-SCRNS	THIS VERSION OF THE FORTH LANGUAGE	2	2
			SYS-SCRNS	IS BASED ON THE fig-FORTH MODEL	2	3
9	128	574	SYS-SCRNS		2	4
			SYS-SCRNS	THE ADDRESS OF THE FORTH INTEREST GROUP IS:	2	5
	128	575	SYS-SCRNS		2	6
			SYS-SCRNS	FORTH INTEREST GROUP	2	7
10	128	576	SYS-SCRNS	P.O. BOX 1105	2	8
			SYS-SCRNS	SAN CARLOS, CA 94070	2	9
	128	577	SYS-SCRNS		2	10
			SYS-SCRNS	TEXAS INSTRUMENTS PERSONNEL WITH SIGNIFICANT	2	11
11	128	578	SYS-SCRNS	INPUT TO THIS VERSION INCLUDE:	2	12
			SYS-SCRNS	LEON TIETZ	2	13
	128	579	SYS-SCRNS	LESLIE O'HAGAN	2	14
			SYS-SCRNS	EDWARD E. FERGUSON	2	15
12	128	580	SYS-SCRNS	( WELCOME SCREEN ) 0 0 GOTOXY ." BOOTING..." CR	3	0
			SYS-SCRNS	BASE->R HEX 10 83C2 C! ( QUIT OFF! )	3	1
	128	581	SYS-SCRNS	DECIMAL ( 84 LOAD ) 20 LOAD 16 SYSTEM MENU	3	2
			SYS-SCRNS	HEX 68 USER VDPME 1 VDPME ! DECIMAL	3	3
13	128	582	SYS-SCRNS	: -SYNONYMS 33 LOAD ; : -EDITOR 34 LOAD ; : -COPY 39 LOAD ;	3	4
			SYS-SCRNS	: -DUMP 42 LOAD ; : -TRACE 44 LOAD ; : -FLOAT 45 LOAD ;	3	5
	128	583	SYS-SCRNS	: -TEXT 51 LOAD ; : -GRAPH1 52 LOAD ; : -MULTI 53 LOAD ;	3	6
			SYS-SCRNS	: -GRAPH2 54 LOAD ; : -SPLIT 55 LOAD ; : -GRAPH 57 LOAD ;	3	7
14	128	584	SYS-SCRNS	: -FILE 68 LOAD ; : -PRINT 72 LOAD ; : -CODE 74 LOAD ;	3	8

Sector	Record Bytes	Record	File Name	Contents (Characters as DOS/IBM ASCII)	Forth Screen	Line
			SYS-SCRNS	: -ASSEMBLER 75 LOAD ; : -64SUPPORT 22 LOAD ;	3	9
	128	585	SYS-SCRNS	: -VDPMODES -TEXT -GRAPH1 -MULTI -GRAPH2 -SPLIT ;	3	10
			SYS-SCRNS	: -BSAVE 83 LOAD ; : -CRU 88 LOAD ;	3	11
15	128	586	SYS-SCRNS		3	12
			SYS-SCRNS		3	13
	128	587	SYS-SCRNS		3	14
			SYS-SCRNS	R->BASE	3	15
:	:	:	:	:	:	:
33	128	622	SYS-SCRNS	oo	8	4
				oo	8	5
		623		oo	8	6
				oo	8	7
34			FORTH	...BOOT A..B..B..B..B%(B..BDSBK1B.FBORBTHBSABVE9..B.IB.JB..B.	8	8
:	:	:	:	:	:	:
39			FORTHSAVE	..42.....@... :"....4`...6.`.<4b4d	9	12
:	:	:	:	:	:	:
77	128	0	SYS-SCRNS	oo	19	4
			SYS-SCRNS	oo	19	5
:	:	:	:	:	:	:
81	128	8	SYS-SCRNS	( CONDITIONAL LOAD )	20	0
			SYS-SCRNS	: MENU CR 272 265 DO I MESSAGE CR LOOP CR CR CR ;	20	1
	128	9	SYS-SCRNS	: SLIT ( --- ADDR OF STRING LITERAL )	20	2
			SYS-SCRNS	R> DUP C@ 1+ =CELLS OVER + >R ;	20	3
82	128	10	SYS-SCRNS		20	4
			SYS-SCRNS	: WLITERAL ( WLITERAL word )	20	5
	128	11	SYS-SCRNS	BL STATE @	20	6
			SYS-SCRNS	IF COMPILE SLIT WORD HERE C@ 1+ =CELLS ALLOT	20	7
83	128	12	SYS-SCRNS	ELSE WORD HERE ENDIF ; IMMEDIATE -->	20	8
			SYS-SCRNS	-SYNONYMS -EDITOR -COPY	20	9
	128	13	SYS-SCRNS	-DUMP -TRACE -FLOAT	20	10
			SYS-SCRNS	-TEXT -GRAPH1 -MULTI	20	11
84	128	14	SYS-SCRNS	-GRAPH2 -SPLIT -VDPMODES	20	12
			SYS-SCRNS	-GRAPH -FILE -PRINT	20	13
	128	15	SYS-SCRNS	-CODE -ASSEMBLER -64SUPPORT	20	14
			SYS-SCRNS	-BSAVE -CRU	20	15
:	:	:	:	:	:	:
359	128	564	SYS-SCRNS		89	12
			SYS-SCRNS		89	13
	128	565	SYS-SCRNS		89	14
			SYS-SCRNS		89	15

## K.4.2 TI Forth Work Disk

The TI Forth original, 90-KB format disk written by **DISK-HEAD** is shown in the following table:

[illegible]

## Appendix L TI Forth System for Larger Disks

Most users of TI Forth these days are using disk sizes that are larger than the original 90 KB disks on which TI supplied TI Forth to TI-99/4A users groups at the end of 1983. This appendix will show you how to put TI Forth on a larger system disk and how to create larger, non-system TI Forth work disks.

### L.1 Larger System Disk

With the following procedure, you can make a TI Forth system disk in a larger disk format:

1. Make a backup copy of the original system disk and use that below where “original system disk” is indicated.
2. Format a disk with Disk Manager or a third-party disk manager to the desired size.
3. With the same disk manager program, copy “FORTH” from the original system disk to the newly formatted disk.
4. Repeat (3) for “FORTHSAVE”.
5. Repeat (3) for “SYS-SCRNS”.
6. Put the following screen on an available blank screen on the original system disk (screens 30 – 32 should be available) and load it:

```
( Swell TI FORTH SYS-SCRNS file to fill disk      07SEP11 LES)
BASE->R : LSYS ; DECIMAL 68 CLOAD STAT 33 CLOAD RANDOMIZE
HEX 0 VARIABLE LESBUF 7E ALLOT 0 VARIABLE LASTREC
PABS @ A + LESBUF 1700 FILE SCRFIL
: FORTHSYS ( size_KB drive_no --- )
  SCRFIL SET-PAB RLTV DSPLY 80 REC-LEN
  F-D" DSK .SYS-SCRNS" ( filename to PAB--space for drive#)
  31 + PAB-ADDR @ D + VSBW ( drive# + 1 to ASCII & put in PAB)
  OPN LESBUF 80 BLANKS ( open file & blank fill buffer)
  4 * 30 - 2 * 1- LASTREC ! LASTREC @ REC-NO ( Set last rec#)
  80 WRT                      ( Write last record)
  25C 23C D0                  ( Restore screens 2-5)
  I REC-NO RD LASTREC @ I + 26F - REC-NO WRT LOOP
  CLSE ;                      ( Close file) R->BASE
```

7. Type the size in KB of the new system disk, the zero-based drive number and the word **FORTHSYS** . If your new system disk is 360 KB and the drive number is 1 (DSK2), type the following on the keyboard:

```
360 1 FORTHSYS
```

To accommodate your larger disk, you now need to add to line 12 of Forth screen 3:

```
360 DISK_SIZE !
```

Depending on whether you use 2 or 3 disk drives, you might also want to follow that with:

**720 DISK\_HI ! or 1080 DISK\_HI !**

When you are done using **FORTH SYS** , you can get rid of its part of the dictionary by executing

**FORGET LSYS**

## ***L.2 Larger Work Disk***

With the procedure delineated below (an alternative to **DISK-HEAD** ), you can make a TI Forth work disk in a larger disk format. If you study it, you will see at a higher level what it is that **DISK-HEAD** is actually doing at a lower level. An updated, more general-purpose **DISK-HEAD** (as **DSK-HD** ) follows in § L.3 .

1. Format a disk with Disk Manager or a third-party disk manager to the desired size.
2. Put the following screen on an available blank screen on your new system disk (there are now plenty of empty screens beyond screen 89) and load it:

```
( Create TI FORTH work disk larger than 90 KB      07SEP11 LES)
BASE->R DECIMAL : LWRK ; 68 CLOAD STAT 33 CLOAD RANDOMIZE
39 CLOAD DISK-HEAD
HEX 0 VARIABLE LESBUF 7E ALL0T
PABS @ A + LESBUF 1700 FILE SCRFIL
: FORTHWRK ( size_KB drive_no --- ) OVER OVER
  SCRFIL SET-PAB RLTV DSPLY 80 REC-LEN
  F-D" DSK .SCREENS" ( filename to PAB--space for drive#)
  31 + PAB-ADDR @ D + VSBW ( drive# + 1 to ASCII & put in PAB)
  OPN LESBUF 80 BLANKS ( open file & blank fill buffer)
  4 * 3 - 2 * 1- REC-NO ( Calculate last record # & set it)
  80 WRT CLSE          ( Write last record & close file)
  * BLOCK !" FORTH    " UPDATE FLUSH ( write disk name->VIB)
; R->BASE
```

3. Ensure that **DISK\_SIZE** and **DISK\_HI** are properly set before executing **FORTHWRK** .
4. If the work disk is in drive 0, be sure to set **DISK\_LO** to 0 before executing **FORTHWRK** .
5. Type the size in KB of the new work disk, the zero-based drive number and the word **FORTHWRK** . If your new work disk is 360 KB and the drive number is 1 (DSK2), type the following on the keyboard:

**360 1 FORTHWRK**

When you are done using **FORTHWRK** , you can get rid of its part of the dictionary by executing

**FORGET LWRK**

## ***L.3 Updating Disk Utilities for Larger Disks***

With disks larger than 90 KB, you will need either to update several disk utility words on the system disk or avoid using them with any but 90 KB disks. These words are

Word	Screen	Lines
<b>DTEST</b>	39	8
<b>FORTH-COPY</b>	39	14
<b>DISK-HEAD</b>	40	0 – 15
<b>FORMAT-DISK</b>	33	7

The above words are redefined in this section to remove hardwired disk sizes from the definitions. The words may then be used for any size disk. You can replace the original definitions of these words on the Forth system screens indicated above except for **FORMAT-DISK**, which will require a bit more work. *Always remember to keep backups (more than one!) of the original disks!!* Please note, also, that the stack effects are the same for only two of these redefined words ( **DTEST** , **FORTH-COPY** ) as for the originals. The other two words ( **DSK-HD** , **FMT-DSK** ) are actually renamed from the originals because their stack effects are different. If you wish to also use the original names with the new stack effects, simply uncomment the definitions that follow each new definition. If you do uncomment the definition of **FORMAT-DISK**, be sure to remove the original definition on screen 33 and, perhaps, add a conditional load for the screen where you put **FMT-DSK** on the bottom line of screen 33. With **FMT-DSK** on screen 100, the new bottom line for screen 33 would be

```
DECIMAL 100 CLOAD FMT-DSK      R->BASE
```

You should probably have a definition for **DISK-HEAD** on screen 40 because other system screens use it for conditional loads. Either uncomment the definition after **DSK-HD**, make it a null definition ( **: DISK-HEAD ;** ) or just bite the bullet by changing the name of **DSK-HD** to **DISK-HEAD** in the new definition and try your best to remember the new stack effects.

Before executing any of these words, be sure that **DISK\_SIZE**, **DISK\_LO** and **DISK\_HI** are properly set.

```
DTEST      ( --- )
```

```
      : DTEST DISK_SIZE @ 0 DO I DUP . BLOCK DROP LOOP ;
```

```
FORTH-COPY ( --- )
```

```
      : FORTH-COPY DISK_SIZE @ 0 DO I DUP . DISK_SIZE @ + I  
        SCOPY LOOP ;
```

```

DSK-HD      ( drive sides density --- )

( WRITE A HEAD COMPATABLE WITH THE DISK MANAGER    07SEP11 LES)
BASE->R HEX : DSK-HD SWAP SWPB + >R DISK_SIZE * DUP
              CLEAR BLOCK ( START SECTOR 0)
              DUP !" FORTH      " DUP A + DISK_SIZE @ 4 * SWAP !
              DUP C + 944 SWAP ! DUP E + 534B SWAP ! DUP 10 + 2028 SWAP !
              DUP 12 + R> SWAP ! DUP 14 + 24 0 FILL DUP 38 + C8 FF FILL
              100 + ( START SECTOR 1) DUP 2 SWAP ! DUP 2+ FE 00 FILL
              100 + ( START SECTOR 2) DUP !" SCREENS    " DUP A + 0 SWAP !
              DUP C + 2 SWAP ! DUP E + DISK_SIZE @ 4 * 3 - DUP >R SWAP !
              DUP 10 + 80 SWAP ! DUP 12 + R> 2 * SWPB SWAP !
              DUP 14 + 8 0 FILL >R 22 R 1C + C! DISK_SIZE @ 4 * 1- DUP 34 -
              DUP F AND 4 SLA R 1D + C! 4 SRA R 1E + C!
              03 R 1F + C! DUP 3 - F AND 4 SLA R 20 + C! 4 SRA R 21 + C!
              R> 22 + 0DE 0 FILL UPDATE FLUSH
; ( : DISK-HEAD DSK-HD ; )          R->BASE

```

Be advised that **DSK-HD** *does* **CLEAR** sectors 0 – 3 (screen 0 if disk is in drive 0) of the disk as does the original **DISK-HEAD**. Also, note that unlike **DISK-HEAD**, **DSK-HD** requires three numbers on the stack, *viz.*, the drive number, the number of sides (1 or 2) and the density (1 or 2). To invoke it for a DSDD disk in drive 1 (DSK2), you would type

```
1 2 2 DSK-HD
```

```

FMT-DSK      ( drive sides density --- sectors )

( Format Disk, given drive #, sides & density 07SEP11 LES)
BASE->R DECIMAL 33 CLOAD RANDOMIZE 0 CLOAD FMT-DSK HEX
: FMT-DSK ( drive sides density --- sectors )
  1 PABS @ VSBW 11 PABS @ 1+ VSBW ( subroutine 11h)
  8350 C! ( density)
  8351 C! ( sides)
  1+ 834C C! ( drive)
  28 834D C! ( 40 tracks)
  DISK_BUF @ 834E ! ( VDP buffer)
  PABS @ 8356 ! 0A 0E SYSTEM ( call DSRLNK subroutine)
  834A @ ( leave sectors formatted)
;          ( : FORMAT-DISK FMT-DSK ; )          R->BASE

```

This disk-formatting word requires on the stack the drive number, number of sides and density of the disk to be formatted. To format a 360-KB DSDD diskette in drive 2 (DSK3), you would type

```
2 2 2 FMT-DSK
```

The following two lines will each format a 90-KB SSSD diskette:

```
2 1 1 FMT-DSK
```

```
2 FORMAT-DISK      (if you keep the original definition)
```



If you were to store the above definition of **FMT-DSK** on screen 100 and would like it to load when the system-synonyms screen loads (the screen with **FORMAT-DISK** on it), then replace line 15 on screen 33 with

**DECIMAL 100 CLOAD FMT-DSK R->BASE**

You might also want to consider deleting the definition of **FORMAT-DISK** from screen 33 because you will not need it with the above word. It is probably not a good idea to rename the new definition **FORMAT-DISK** because its stack effects are so different from the old definition and could be confusing.

As with **FORMAT-DISK**, **FMT-DSK** creates a disk that can only be used by TI Forth. There is no information written to sectors 0 and 1 that will allow file-access words to work with the disk. If you run **DSK-HD** after **FMT-DSK**, you can then use the disk for file access from TI Forth, TI BASIC, etc. A clunky way to create a blank disk would be to delete the file "SCREENS" from the disk after running **DSK-HD** by using file-access words described in Chapter 8. You can (*carefully!*) change the name of the disk by editing the first 10 bytes of Forth screen 0 for a disk in drive 0 and after setting **DISK\_LO** to 0. Just remember to use names of no more than 10 characters that contain no spaces or periods. Spaces *should* be used *after* the name to fill out the 10 characters in this field.