

# fbForth 2.0

## A File-Based Cartridge Implementation of TI Forth

...updated June 20, 2015

**Lee Stewart**

Based on the *TI Forth Instruction Manual* (1983) by Leslie  
O'Hagan, Leon Tietz and John T. Yantis  
and the **fbForth 1.0 Manual** (2014) by Lee Stewart

Copyright © 2014 – 2015 by Lee Stewart

Permission is hereby granted to copy and distribute this document at will.

# Original Dedication of TI Forth

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.

*—from original TI Forth Manual*

# Table of Contents

Original Dedication of TI Forth.....	iii
1 Introduction.....	1
1.1 Original Introduction to TI Forth.....	1
1.2 Author's Introduction.....	3
1.3 Starting fbForth 2.0.....	5
1.4 fbForth 2.0 Terminal Response.....	7
1.5 Changing How fbForth 2.0 Starts.....	7
1.6 Acknowledgments.....	7
2 Getting Started.....	8
2.1 Stack Manipulation.....	9
2.2 Arithmetic and Logical Operations .....	9
2.3 Comparison Operations.....	10
2.4 Memory Access Operations .....	11
2.5 Control Structures.....	11
2.5.1 IF ... THEN.....	11
2.5.2 IF ... ELSE ... THEN.....	12
2.5.3 BEGIN ... AGAIN.....	13
2.5.4 BEGIN ... UNTIL.....	13
2.5.5 BEGIN ... WHILE ... REPEAT.....	14
2.5.6 DO ... LOOP.....	15
2.5.7 DO ... +LOOP.....	16
2.5.8 CASE ... OF ... ENDOF ... ENDCASE.....	17
2.6 Input and Output to/from the Terminal .....	18
2.7 Numeric Formatting .....	19
2.8 Block-Related Words.....	20
2.9 Defining Words.....	21
2.10 Miscellaneous Words.....	21
3 How to Use the fbForth 2.0 Editors.....	23
3.1 Forth Block Layout Caveat.....	23
3.2 The Two fbForth Editors.....	24
3.3 Editing Instructions.....	26
3.4 Changing Foreground/Background Colors of 64-Col Editor.....	28
3.5 Block-Copying Utility.....	28
4 Memory Maps.....	30
4.1 VDP Memory Map.....	30
4.2 CPU Memory.....	31
4.3 CPU RAM Pad.....	32
4.4 Low Memory Expansion.....	33
4.5 High Memory Expansion.....	33
5 System Synonyms and Miscellaneous Utilities.....	34
5.1 System Synonyms.....	34

5.1.1	VDP RAM Read/Write.....	34
5.1.2	Extended Utilities: GPLLNK, XMLLNK and DSRLNK.....	36
5.1.3	VDP Write-Only Registers.....	37
5.1.4	VDP RAM Single-Byte Logical Operations.....	37
5.2	Disk Utilities.....	38
5.3	Listing Utilities.....	38
5.4	Debugging.....	39
5.4.1	Dump Information to Terminal.....	39
5.4.2	Tracing Word Execution.....	40
5.4.3	Recursion.....	40
5.5	Random Numbers.....	41
5.6	Miscellaneous Instructions.....	42
6	An Introduction to Graphics.....	43
6.1	Graphics Modes.....	43
6.2	fbForth 2.0 Graphics Words.....	44
6.3	Color Changes.....	45
6.4	Placing Characters on the Screen.....	46
6.5	Defining New Characters.....	46
6.6	Sprites.....	48
6.6.1	Magnification.....	48
6.6.2	Sprite Initialization.....	49
6.6.3	Using Sprites in Bitmap Mode.....	49
6.6.4	Creating Sprites.....	50
6.6.5	Sprite Automotion.....	51
6.6.6	Distance and Coincidences between Sprites.....	52
6.6.7	Deleting Sprites.....	53
6.7	Multicolor Graphics.....	54
6.8	Using Joysticks.....	55
6.9	Dot Graphics.....	58
6.10	Special Sounds.....	59
6.11	Constants and Variables Used in Graphics Programming.....	59
7	The Floating Point Support Package.....	60
7.1	Floating Point Stack Manipulation.....	60
7.2	Floating Point Defining Words.....	61
7.3	Floating Point Fetch and Store.....	61
7.4	Floating Point Conversion Words.....	61
7.5	Floating Point Number Manipulation.....	61
7.6	Floating Point Number Entry.....	62
7.7	Built-in Floating Point Constants.....	62
7.8	Floating Point Arithmetic.....	63
7.9	Floating Point Comparison Words.....	63
7.10	Formatting and Printing Floating Point Numbers.....	63
7.11	Transcendental Functions.....	65
7.12	Interface to the Floating Point Routines.....	66

7.13 Handling Floating Point Errors.....	66
7.14 Floating Point Error Codes.....	67
8 Access to File I/O Using TI-99/4A Device Service Routines.....	68
8.1 Switching VDP Modes After File Setup.....	68
8.2 The Peripheral Access Block (PAB).....	68
8.3 File Setup and I/O Variables.....	69
8.4 File Attribute Words.....	70
8.5 Words that Perform File I/O.....	71
8.6 Alternate Input and Output.....	74
8.7 File I/O Example 1: Relative Disk File.....	75
8.8 File I/O Example 2: Sequential RS232 File.....	76
8.9 Disk Catalog Utilities.....	77
8.9.1 DIR.....	77
8.9.2 CAT.....	78
9 The fbForth 2.0 TMS9900 Assembler.....	79
9.1 TMS9900 Assembly Mnemonics.....	79
9.2 fbForth 2.0's Workspace Registers.....	80
9.3 Loading and Using the Assembler.....	80
9.4 fbForth 2.0 Assembler Addressing Modes.....	81
9.4.1 Workspace Register Addressing.....	82
9.4.2 Symbolic Memory Addressing.....	82
9.4.3 Workspace Register Indirect Addressing.....	82
9.4.4 Workspace Register Indirect Auto-increment Addressing.....	83
9.4.5 Indexed Memory Addressing.....	83
9.4.6 Addressing Mode Words for Special Registers.....	83
9.5 Handling the fbForth 2.0 Stacks.....	84
9.6 Structured Assembler Constructs.....	84
9.7 Assembler Jump Tokens.....	85
9.8 Assembly Example for Structured Constructs.....	85
9.9 Assembly Example with DOES>ASM:.....	86
9.10 ASM: and DOES>ASM: without the Assembler.....	87
9.10.1 ASM: without the Assembler.....	88
9.10.2 DOES>ASM: without the Assembler.....	88
10 Interrupt Service Routines (ISRs).....	90
10.1 Installing an fbForth 2.0 Interrupt Service Routine.....	90
10.2 Example of an Interrupt Service Routine.....	91
10.3 Installing the ISR.....	91
10.4 Some Additional Thoughts Concerning the Use of ISRs.....	92
11 Potpourri.....	93
11.1 BSAVE and BLOAD.....	93
11.1.1 Using BSAVE to Customize How fbForth 2.0 Boots Up.....	94
11.1.2 An Overlay System with BSAVE and BLOAD.....	95
11.1.3 An Easier Overlay System in Source Code.....	96
11.2 Conditional Loads.....	97

11.3 CRU Words.....	98
11.4 Useful Additional Stack Words.....	99
12 fbForth 2.0 Dictionary Entry Structure.....	101
12.1 Link Field.....	101
12.2 Name Field.....	101
12.3 Code Field.....	102
12.4 Parameter Field.....	103
12.5 Notes on Resident Dictionary Words.....	103
13 Screen Fonts and the Font Editor.....	104
13.1 Loading the Default Screen Font.....	104
13.2 User Fonts.....	104
13.3 Using the Font Editor.....	105
13.4 Modifying the 64-Column Editor's Font.....	106
14 The Stack-based String Library.....	108
14.1 Introduction—The Concepts behind the Library.....	108
14.1.1 Coding Conventions.....	108
14.1.2 Stack Notation.....	109
14.1.3 Loading the String Stack Library.....	109
14.2 String Constant Words.....	109
14.3 String Stack Words.....	111
14.4 The String Stack.....	117
14.5 Error Checking.....	117
14.6 String Stack Format.....	117
14.7 String Constant Format.....	118
14.8 Throw Codes.....	118
14.9 Author Information.....	118
Appendix A ASCII Keycodes (Sequential Order).....	119
Appendix B ASCII Keycodes (Keyboard Order).....	121
Appendix C How fbForth 2.0 differs from Starting FORTH (1st Ed.).....	123
Appendix D The fbForth 2.0 Glossary.....	130
D.1 Explanation of Some Terms and Abbreviations.....	130
D.2 fbForth 2.0 Word Descriptions.....	131
Appendix E Differences: fbForth 2.0, fbForth 1.0 and TI Forth.....	207
E.1 TI Forth Words not in fbForth 2.0.....	207
E.2 fbForth 1.0 Words not in fbForth 2.0.....	210
E.3 New and Modified Words in fbForth 2.0.....	212
Appendix F User Variables in fbForth 2.0.....	218
F.1 fbForth 2.0 User Variables (Address Offset Order).....	218
F.2 fbForth 2.0 User Variables (Variable Name Order).....	220
Appendix G fbForth 2.0 Load Option Directory.....	222
G.1 Option: 64-Column Editor.....	222
G.2 Option: CPYBLK -- Block Copying Utility.....	222
G.3 Option: Memory Dump Utility.....	222
G.4 Option: TRACE -- Colon Definition Tracing.....	222

G.5 Option: Printing Routines.....	222
G.6 Option: TMS9900 Assembler.....	223
G.7 Option: CRU Words.....	223
G.8 Option: Stack-based String Library.....	223
G.9 Option: DIR -- Disk Catalog Utility.....	223
G.10 Option: CAT -- Disk Catalog Utility.....	223
Appendix H Assembly Source for CODEd Words.....	224
Appendix I Error Messages.....	230
Appendix J Contents of FBLOCKS.....	232
Appendix K Diskette Format Details.....	250
K.1 Volume Information Block (VIB).....	250
K.2 File Descriptor Index Record (FDIR).....	251
K.3 File Descriptor Record (FDR).....	251
Appendix L Notes on Radix-100 Notation.....	253

# 1 Introduction

## 1.1 Original Introduction to TI Forth

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 (1<sup>st</sup> Ed.)*  
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. [***Author'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.

[***Author's Note: This paragraph's use of DR0, DR1, etc. does not obtain for fbForth because those words have been eliminated from fbForth***] 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.

*—from original TI Forth Manual*

## 1.2 Author's Introduction

My source for the text of the original *TI Forth Instruction Manual*, much of which is included in 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 original printed document. I have taken the liberty of incorporating most of the original into this **fbForth 2.0**: *A File-Based Cartridge Implementation of TI Forth*.

Forth screens are now referred to as blocks, in line with the current Forth convention.

New to **fbForth 2.0** are

- The facility for loading your own font in place of the default font in the cartridge (see Chapter 13 “Screen Fonts and the Font Editor”). Chapter 13 also provides instruction for using the author’s Font Editor. New words are **SCRFONT**, **FNT**, **USEFFL** and **FONTE**.
- A stack-based string library has been added to FBLOCKS and is fully described in Chapter 14 “The Stack-based String Library”.
- Several software packages, previously requiring loading from FBLOCKS, are now part of the resident dictionary:
  - 40/80 Column Editor, rewritten in TMS9900 Assembly Language (ALC) for efficiency, now including an on-screen menu
  - Floating Point Math Library, which no longer uses the console GPL/XML-based library, and contains several new words, including **FFMT**. (includes a formatted print option for 3-digit E-notation)
  - File I/O Library, including new words, **>DEG**, **>RAD**, **CEIL**, **DEG/RAD**, **EXP10**, **EULER\_E**, **F>R**, **FABS**, **FCONSTANT**, **FMT.**, **FLOOR**, **FMINUS**, **FP1**, **FP10**, **FRAC**, **FROT**, **FVARIABLE**, **LN10INV**, **LOG10**, **R>F**, **RAD/DEG** and **TRUNC**
  - **BSAVE**—Binary Save Routine
  - All Graphics modes, including **VMODE**, an all-purpose mode-changing word
  - Graphics Primitives Library, much of it rewritten in ALC
- **.BASE**—a new word for displaying the current radix (number base) in decimal: This is useful because, regardless of the current radix, executing **BASE @ .** will always display **10**—not particularly useful! For example, **HEX .BASE** yields **16**, which is much more informative.
- **DIR**—new disk cataloging word that uses the DSR’s catalog “file” to get disk and file information: The actual byte size of PROGRAM files is unavailable with this word.
- **CAT**—new disk cataloging word that uses the disk’s VIB, FDIR and FDRs to get disk and file information: The actual byte size of PROGRAM files is displayed.
- Many new words have been added and many words have been removed (see Appendix E “Differences: fbForth 2.0, fbForth 1.0 and TI Forth”).

You will notice at startup that a revision number appears following a ‘.’ after the version number. The current version:revision is displayed as **fbForth 2.0:2**. Using a revision number allows for

minor builds that correct errors and fix bugs. Until a version-number change, this document will continue to use **fbForth 2.0** without the revision number in most instances.

Also, note that the title page of this document now displays the date it was updated, right-justified, just below the subtitle: "...updated June 20, 2015" in the present case.

Though, in coding **fbForth 2.0**, I have been careful with my modifications of TI Forth in converting it to use file I/O for reading and writing **fbForth** blocks, 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*  
*June, 2015*  
Silver Run, MD

### 1.3 Starting fbForth 2.0

To operate the **fbForth 2.0** System, you must have the following equipment or equivalent:

- TI-99/4A Console
- Monitor
- Memory Expansion
- Disk Controller
- 1 (or more) Disk Drives
- fbForth 2.0** Module
- RS232 Interface (optional)
- Printer (optional)

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

The **fbForth 2.0** system consists of the **fbForth 2.0** cartridge and the blocks file, FBLOCKS, on the system disk. **fbForth 2.0** will start perfectly well without FBLOCKS; however, there are a few useful utilities in FBLOCKS. The system will also complain if it does not find it.

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 **fbForth 2.0**. To use **fbForth 2.0**, select its menu option.

**fbForth 2.0** boots and displays the following welcome screen if DSK1.FBLOCKS is found (Note that the revision number now appears after the ':'):

```
fbForth 2.0:2      (c) 2014 Lee Stewart
...a file-based TI Forth implementation
FBLOCKS mod: 19JUN2015
Type MENU for load options.

fbForth 2.0
■
```

Note that the current modification date of FBLOCKS is displayed. Typing “MENU” per the startup instructions will load the following menu:

```

Load Options (19JUN2015)      fbForth 2.0
Description                    Load Block
-----
CPYBLK -- Block Copying Utility.....4
CRU Words.....5
64-Column Editor.....6
Memory Dump Utility.....16
TRACE -- Colon Definition Tracing....18
Printing Routines.....19
TMS9900 Assembler.....21
More Useful Stack Words etc.....41
Stack-based String Library.....42
DIR -- Disk Catalog Utility.....36
CAT -- Disk Catalog Utility.....58
TMS9900 Assembler BLOAD (clears RAM)..27
64-Column Editor BLOAD (clears RAM)..32
String Library BLOAD (clears RAM)....52

Type <block> LOAD to load.
ok:0

```

Loading a block in the “Load Block” column below loads all routines necessary to perform a particular group of tasks:

Start Block	Loads Forth Words Necessary to:	Chapter
4	Copy a range of blocks <sup>1</sup> to the same or another blocks file.	5
5	Access the <b>fbForth 2.0</b> equivalents of TMS9900 Assembler mnemonics for the CRU: LDCR, STCR, SBO, SBZ and TB.	11
6	Run the 64-column <b>fbForth 2.0</b> editor.	3
16	Execute <b>DUMP</b> and <b>VLIST</b> .	5
18	Trace the execution of Forth words.	5
19	Send output to an RS232 (or similar) device.	8
21	Write routines in <b>fbForth 2.0</b> TMS9900 Assembler.	9

To load a particular package, simply type its block number, exactly as it appears in the list, followed by **LOAD**. For example, to load the TMS9900 Assembler, type **21 LOAD** and press **<ENTER>**. You may load more than one package at a time.

<sup>1</sup> A Forth block (TI Forth uses ‘screen’) consists of 16 lines of 64 characters for a total of 1024 characters. When a Forth block is loaded from a blocks file, 1024 characters are copied from the file into a RAM block buffer. This is explained in more detail later in this document.

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

## 1.4 fbForth 2.0 Terminal Response

With few exceptions after typing **<ENTER>**, **fbForth** responds with:

ok:n

where the number *n* following **ok:** is the depth of the parameter stack, *i.e.*, the count of numbers or cells on the stack. For example, if the stack were empty and you typed three numbers followed by **<ENTER>**, the following would obtain:

2 4 6 ok:3

Note that above and elsewhere in this manual the computer's responses are underlined.

## 1.5 Changing How fbForth 2.0 Starts

When **fbForth 2.0** boots up, it always looks for DSK1.FBLOCKS and complains if it does not find it. Upon finding it, **fbForth 2.0** always loads block 1, the first block in the file. This provides you a way to change what happens at that point in the **fbForth** boot process. You can design your own blocks file that loads your favorite words, including those you create. All you need to do is to eventually rename the file "FBLOCKS" and place it in DSK1 when you want **fbForth 2.0** to load it after it boots up.

## 1.6 Acknowledgments

The author would like to thank the following for their help with the development of **fbForth 2.0**:

- Tim Tesch for the source code of the MDOS L10 Floating Point Library (FPL) and encouragement and permission in this effort to adapt it to do the heavy lifting for the **fbForth 2.0** Floating Point Library.
- Beery Miller for permission to use the MDOS L10 FPL.
- 9640News and all of the MDOS contributors who developed the TMS9900 code for the MDOS L10 FPL, which the author adapted for use with **fbForth 2.0** on the TI-99/4A.
- Mark Wills for his assistance, encouragement, permission to use code from TurboForth and porting his TurboForth String Stack Library to **fbForth 2.0**.

## 2 Getting Started

This chapter will familiarize you with the most common words (instructions, routines) 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 **fbForth 2.0**. Those with no Forth experience should begin by reading a book such as *Starting FORTH, (1<sup>st</sup> Ed.)* by Leo Brodie. Appendix C “How fbForth 2.0 differs from Starting FORTH (1st Ed.)” is designed to be used side by side with *Starting FORTH, (1<sup>st</sup> Ed.)* and lists the differences between the Forth language described in the book (poly-Forth) and **fbForth 2.0**.

A word in Forth is any sequence of characters delimited (set off) 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 ? ." ASM:

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 cell)
$c$	7-bit character (in right end of cell )
$flag$	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>	( $n \text{ --- } n \ n \mid n$ )	Duplicate only if non-zero
<b>.S</b>	( --- )	Non-destructively display stack contents
<b>&gt;R<sup>2</sup></b>	( $n \text{ ---}$ ) ( R: --- $n$ )	Move top item on stack to return stack
<b>DEPTH</b>	( --- <i>stack-depth</i> )	Number of cells on parameter stack
<b>DROP</b>	( $n \text{ ---}$ )	Discard top of stack
<b>DUP</b>	( $n \text{ --- } n \ n$ )	Duplicate top of stack
<b>OVER</b>	( $n_1 \ n_2 \text{ --- } n_1 \ n_2 \ n_1$ )	Make copy of second item on top
<b>R</b>	( --- $n$ ) ( R: $n \text{ --- } n$ )	Copy top item of return stack to stack
<b>R&gt;</b>	( --- $n$ ) ( R: $n \text{ ---}$ )	Move top item on return stack to stack
<b>ROT</b>	( $n_1 \ n_2 \ n_3 \text{ --- } n_2 \ n_3 \ n_1$ )	Rotate third item to top
<b>SP!</b>	( --- )	Clear stack, resetting it to its base <b>S0</b>
<b>SWAP</b>	( $n_1 \ n_2 \text{ --- } n_2 \ n_1$ )	Exchange top two stack items

## 2.2 Arithmetic and Logical Operations

The following are the most common arithmetic and logical operations:

<b>*</b>	( $n_1 \ n_2 \text{ --- } n_3$ )	Multiply
<b>*/</b>	( $n_1 \ n_2 \ n_3 \text{ --- } \textit{quot}$ )	Like <b>*/MOD</b> but giving <i>quot</i> only
<b>*/MOD</b>	( $n_1 \ n_2 \ n_3 \text{ --- } \textit{rem} \ \textit{quot}$ )	$n_1 * n_2 / n_3$ with 32 bit intermediate
<b>+</b>	( $n_1 \ n_2 \text{ --- } n_3$ )	Add
<b>-</b>	( $n_1 \ n_2 \text{ --- } n_3$ )	Subtract ( $n_1 - n_2$ )
<b>/</b>	( $n_1 \ n_2 \text{ --- } n_3$ )	Divide $n_1$ by $n_2$ and leave quotient $n_3$
<b>/MOD</b>	( $n_1 \ n_2 \text{ --- } \textit{rem} \ \textit{quot}$ )	Divide $n_1$ by $n_2$ giving remainder & quotient
<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>ABS</b>	( $n \text{ --- }  n $ )	Absolute value

<sup>2</sup> **>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>AND</b>	$(n_1 n_2 \text{ --- } n_3)$	Bitwise logical AND $n_3$
<b>D+</b>	$(d_1 d_2 \text{ --- } d_3)$	Add double precision numbers
<b>DABS</b>	$(d \text{ --- }  d )$	Absolute value of 32-bit number
<b>DMINUS</b>	$(d_1 \text{ --- } d_2)$	Leave two's complement of 32-bits
<b>MAX</b>	$(n_1 n_2 \text{ --- } n_1   n_2)$	Maximum
<b>MIN</b>	$(n_1 n_2 \text{ --- } n_1   n_2)$	Minimum
<b>MINUS</b>	$(n_1 \text{ --- } n_2)$	Leave two's complement
<b>MOD</b>	$(n_1 n_2 \text{ --- } n_3)$	Modulo ( remainder from $n_1 / n_2$ )
<b>OR</b>	$(n_1 n_2 \text{ --- } n_3)$	Bitwise logical OR $n_3$
<b>SGN</b>	$(n \text{ --- } -1   0   +1)$	Sign of $n$ as -1   0   +1
<b>SLA</b>	$(n_1 n_2 \text{ --- } n_3)$	Shift $n_1$ left arithmetic $n_2$ bits giving $n_3$
<b>SRA</b>	$(n_1 n_2 \text{ --- } n_3)$	Shift $n_1$ right arithmetic $n_2$ bits giving $n_3$
<b>SRC</b>	$(n_1 n_2 \text{ --- } n_3)$	Shift $n_1$ right circular $n_2$ bits giving $n_3$
<b>SRL</b>	$(n_1 n_2 \text{ --- } n_3)$	Shift $n_1$ right logical $n_2$ bits giving $n_3$
<b>SWPB</b>	$(n_1 \text{ --- } n_2)$	Swap the bytes of $n_1$ producing $n_2$
<b>XOR</b>	$(n_1 n_2 \text{ --- } n_3)$	Bitwise logical exclusive OR $n_3$
<b>U*</b>	$(u_1 u_1 \text{ --- } u_2)$	Unsigned * with double product
<b>U/</b>	$(u_1 u_2 \text{ --- } u_{rem} \text{ uquot} )$	Unsigned / with remainder

### 2.3 Comparison Operations

The following are the most common comparisons:

<b>&lt;</b>	$(n_1 n_2 \text{ --- } flag)$	True if $n_1$ less than $n_2$ (signed)
<b>=</b>	$(n_1 n_2 \text{ --- } flag)$	True if top two numbers are equal
<b>&gt;</b>	$(n_1 n_2 \text{ --- } flag)$	True if $n_1$ greater than $n_2$
<b>0&lt;</b>	$(n \text{ --- } flag)$	True if top number is negative
<b>0=</b>	$(n \text{ --- } flag)$	True if top number is 0 ( <i>i.e.</i> , NOT)
<b>0&gt;</b>	$(n \text{ --- } flag)$	True if top number is positive
<b>U&lt;</b>	$(u_1 u_2 \text{ --- } flag)$	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>	( <i>n addr ---</i> )	Store <i>n</i> at address (store a cell)
<b>+!</b>	( <i>n addr ---</i> )	Add <i>n</i> to contents of address
<b>?</b>	( <i>addr ---</i> )	Print the contents of address (same as <b>@ .</b> )
<b>@</b>	( <i>addr --- n</i> )	Replace word address by its contents
<b>C!</b>	( <i>b addr ---</i> )	Store <i>b</i> at address (store a byte)
<b>C@</b>	( <i>addr --- b</i> )	Fetch the byte at <i>addr</i>
<b>CMOVE</b>	( <i>from_addr to_addr u ---</i> )	Block move <i>u</i> bytes.
<b>BLANKS</b>	( <i>addr u ---</i> )	Fill <i>u</i> bytes with blanks beginning at <i>addr</i>
<b>ERASE</b>	( <i>addr u ---</i> )	Fill <i>u</i> bytes beginning at <i>addr</i> with 0s
<b>FILL</b>	( <i>addr u b ---</i> )	Fill <i>u</i> bytes with <i>b</i> beginning at <i>addr</i>
<b>MOVE</b>	( <i>from_addr to_addr u ---</i> )	Block move <i>u</i> cells.

## 2.5 Control Structures

The sets of words detailed in the following sections are used to implement control structures in **fbForth**. They are used to create all looping and conditional structures within the definitions of **fbForth** words. These structures may be nested to any depth that the return and parameter stacks can tolerate. If they are nested improperly an error message will be generated at compile time and the word definition will be aborted.

It can be very difficult for programmers new to Forth to understand how control structures work in Forth because of the stack-oriented nature of the language. Using these control structures will be a piece of cake once you understand that the value tested or otherwise consumed by **IF**, **UNTIL**, **WHILE**, **CASE**, **OF**, **ENDCASE** or **DO** must be on the stack *before* the word is executed rather than following the word inline as with most other programming languages. The sections that follow show details and examples of each control structure to give you a better idea of how they work. Some of the examples are taken from the resident dictionary of **fbForth 2.0** while others are from nonresident words that are part of the default system blocks file, FBLOCKS.

### 2.5.1 IF ... THEN

**IF ... THEN**

**IF** ( *flag ---* )

**ENDIF**

**IF** tests the top of stack and if non-zero (*true*), the words between **IF** and **THEN** are executed. Otherwise, they are skipped and execution resumes after **THEN**.

Synonym for **THEN**.

The words **IF** and **THEN** enclose code that will be executed when **IF** finds a nonzero value for *flag* on the stack. Consider the following example that simply takes the number on top of the stack and makes sure it is even, adding 1 if it is not:

```

: EVEN          « Define word EVEN to insure top of stack contains an even
                « number. Add 1 if not.
  ( n1 --- n1 | n1+1 ) « In: n1. Out: n1 or n1+1.
  DUP 1 AND     « Duplicate n1. Check if odd, i.e., LSb (least-significant bit)
                « set.
  IF           « Is n1 odd? ( IF tests the number left on the stack in the
                « above line).
    1+         « Yes. Add 1 to n1 to make it even.
  THEN
;

```

### 2.5.2 IF ... ELSE ... THEN

```

IF ... ELSE ... THEN      IF tests the top of stack and if non-zero (true), the
                            words between IF and ELSE are executed. If the top of
                            the stack is zero (false), the words between ELSE and
                            THEN are executed. Execution then continues after
                            THEN .
  IF      (flag ---)

```

The **IF ... ELSE ... THEN** structure causes execution of one of two alternatives. The following example is part of the **fbForth 2.0** resident dictionary. **CLOAD** loads a block from the current blocks file only if the word that follows **CLOAD** in the input stream cannot be found in the dictionary. It is a state-smart word that can be used in a word definition as well as on the command line. It is used in the following way:

```

20 CLOAD MYWORD ,

```

where **20** is the block that will be loaded from the current blocks file if **MYWORD** is not found in the dictionary.

```

: CLOAD          « Define CLOAD to conditionally load a block from blocks file.
  ( blk# --- )   « Load blk# if word after CLOAD not found.
  [COMPILE] WLITERAL « Force immediate word WLITERAL to compile into definition
                    « of CLOAD so it executes when CLOAD executes.
  STATE @       « Get compilation state for IF to test.
  IF           « Are we compiling?
    COMPILE <CLOAD> « Yes. Defer execution of runtime procedure <CLOAD> by
                    « compiling it into word invoking CLOAD in its definition.
  ELSE         « No. Execute it.
    <CLOAD>
  THEN
; IMMEDIATE    « Make CLOAD immediate, i.e., execute even if compiling.

```

### 2.5.3 BEGIN ... AGAIN

**BEGIN ... AGAIN** Creates an infinite loop, continually re-executing the words between **BEGIN** and **AGAIN**<sup>3</sup>.

The **BEGIN ... AGAIN** infinite loop is the simplest looping structure in **fbForth** because there are no tests—it just repeats forever the words between **BEGIN** and **AGAIN**. The only way the loop can be exited is if **QUIT** or **ABORT** gets executed within the loop or another word drops the top of the return stack.<sup>3</sup> Generally, however, if you wish to provide a normal exit from the loop, you should use one of the conditionally looping structures described in sections following this one.

The following example is the primary loop in **fbForth**. The last thing the **fbForth** boot process does is to execute **QUIT**. **QUIT** is an endless loop whose primary function is to repeatedly call the interpreter, which is itself an endless loop:

```

: QUIT ( --- )      « Define QUIT with no inputs or outputs.
  0 BLK !          « Store 0 in BLK to set up input from the terminal.
  [COMPILE] [      « Compile immediate word [ into QUIT 's definition; [ will
                   « set system to interpret state when QUIT executes.
  BEGIN           « Start infinite, top-level loop.
    RP! CR        « Clear return stack. Put screen cursor at start of next line.
    QUERY         « Get a line of text.
    INTERPRET     « Interpret input text.
    STATE @       « Get compilation state.
    0= IF         « Are we interpreting, i.e., STATE = 0?
      ." ok:" DEPTH . « Yes. Echo "ok:" to the terminal followed by stack depth.
    THEN
  AGAIN          « Repeat loop.
;

```

### 2.5.4 BEGIN ... UNTIL

**BEGIN ... UNTIL** Loop that executes the words between **BEGIN** and **UNTIL**, which must leave *flag* to be tested by **UNTIL**, until *flag* is non-zero (true).

**END** Synonym for **UNTIL**.

The following example from FBLOCKS is from block 16 of the memory dump utility. **VLIST** lists words in the **CONTEXT** vocabulary starting with the last defined word pointed to by **CONTEXT** and following the linked list of words and vocabularies until it finds the first word at the top of the chain that has a pointer (link field address or *lfa*) of 0. This topmost word will always be **EXECUTE** in **fbForth**. See Chapter 12 “fbForth 2.0 Dictionary Entry Structure” for an explanation of **fbForth** word fields and their abbreviations (*lfa*, *nfa*, *cfa* and *pfa*). If you know the *pfa*, you can get the other three field addresses for a given word. You can get the *pfa* if you know the *nfa*. These facts are used in the following example:

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

<b>: VLIST</b>	« Define <b>VLIST</b> to list the <b>CONTEXT</b> vocabulary.
<b>( --- )</b>	« Takes no parameters and leaves none.
<b>80 OUT !</b>	« Store maximum expected character count in <b>OUT</b> .
<b>CONTEXT @ @</b>	« Get <i>nfa</i> of last defined word in <b>CONTEXT</b> vocabulary.
<b>0 SWAP</b>	« Start word counter at 0 and swap <i>nfa</i> to top of stack.
<b>BEGIN</b>	« Start indefinite loop.
<b>DUP C@ 3F AND</b>	« Dup <i>nfa</i> . Get length byte's least-significant 5 bits.
<b>OUT @ +</b>	« Add name length to <b>OUT</b> .
<b>SCRN_WIDTH @ 3 -</b>	« Get screen width – 3 for spaces and end of line.
<b>&gt; IF</b>	« Will line be too long?
<b>CR 0 OUT !</b>	« Yes. Go to next line and zero character count.
<b>THEN</b>	
<b>DUP ID.</b>	« Dup <i>nfa</i> . Display name.
<b>SWAP 1+ SWAP</b>	« Get word count to top. Increment it. Swap <i>nfa</i> back.
<b>PFA LFA @</b>	« Get <i>lfa</i> from <i>pfa</i> . Get next word's <i>nfa</i> from <i>lfa</i> .
<b>SPACE</b>	« Emit a space (updates <b>OUT</b> in the process).
<b>DUP 0=</b>	« Dup new <i>nfa</i> . Leave <i>true</i> if 0, else <i>false</i> .
<b>PAUSE</b>	« Pause if keystroke. Return <i>true</i> if < <b>BREAK</b> >, else <i>false</i> .
<b>OR UNTIL</b>	« <b>OR</b> above flags. Exit loop if <i>true</i> , else repeat.
<b>DROP CR . ." words listed"</b>	« Drop leftover <i>nfa</i> . Display word count on next line.
<b>;</b>	

### 2.5.5 BEGIN ... WHILE ... REPEAT

<b>BEGIN ... WHILE ... REPEAT</b>	Executes words between <b>BEGIN</b> and <b>WHILE</b> , which must leave <i>flag</i> to be tested by <b>WHILE</b> . If <i>flag</i> is non-zero ( <i>true</i> ), executes words between <b>WHILE</b> and <b>REPEAT</b> , then jumps back to <b>BEGIN</b> . If <i>flag</i> is zero ( <i>false</i> ), continues execution after the <b>REPEAT</b> .
<b>WHILE (flag ---)</b>	

The following example starts with a **BEGIN ... UNTIL** loop that waits for the left joystick's fire button to be depressed, after which it starts a counter and enters the **BEGIN ... WHILE ... REPEAT** loop. That loop waits for the fire button to be released, counting the number of times through the loop while that is not happening. After the fire button is released, the **WHILE** clause is not executed and the loop exits. **FIREDOWN** finishes with the display of the number of iterations through the **BEGIN ... WHILE ... REPEAT** loop:

<b>: FIREDOWN</b>	« Define <b>FIREDOWN</b> to display loop iterations between press and release of left joystick's fire button.
<b>( --- )</b>	« No parameters in or out.
<b>BEGIN</b>	« Start indefinite loop awaiting fire button press.
<b>1 JOYST DROP DROP</b>	« Get state of joystick/keyboard #1. Save only char value.
<b>18 = UNTIL</b>	« Repeat loop until char is fire-button value (18).
<b>0</b>	« Initialize counter on stack.
<b>BEGIN</b>	« Start indefinite loop awaiting release of fire button.
<b>1 JOYST DROP DROP</b>	« Get state of joystick/keyboard #1. Save only char value.
<b>18 = WHILE</b>	« Continue with loop while char value = 18, else exit.
<b>1+</b>	« Increment loop counter on stack.

```

REPEAT                « Repeat loop.
CR . ." iterations." « Display # of iterations on next screen line.
;

```

### 2.5.6 DO ... LOOP

<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>lim strt ---</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>	( <i>---</i> )	Causes loop to terminate at next <b>LOOP</b> or <b>+LOOP</b> .

The following example could have been written more efficiently; but, this version makes use of all of the above words. The word **8X8SRCH** defined below looks on the stack for the address of an 8x8 array *addr* of numbers to search and a number *n* to match. The result will be only a *false* flag if there is no match, but a *true* flag, row *r* and column *c* of the array if there is a match.

You will notice that the stack depth is stored on the return stack before entering the outer **DO** loop and moved to the parameter stack when that loop is exited to then calculate the difference. The reason for this maneuver is that there is no way for **8X8SRCH** to anticipate how many cells there may be on the stack below *n* before **8X8SRCH** executes:

```

: 8X8SRCH                « Define 8X8SRCH to search an 8x8, row-major array for a
                          number.
  ( n addr --- F | c r T ) « In: n = number to match; addr = array address. Out:
                          false (0), if not found—or c = column; r = row; true
                          (non-zero), if found.
  DEPTH >R                « Store stack depth to return stack to check at end.
  8 0 DO                  « Array row loop.
    8 0 DO                « Array column loop.
      OVER OVER           « Copy n and addr to top of stack.
      J 8 * I +           « Convert row r and column c to address offset into array.
      + @                 « Add offset to addr and get value at that location.
      = IF                « Do we have a match to n?
        DROP DROP        « Yes. DROP top 2 numbers from the stack.
        I J 1 LEAVE      « Leave column c, row r and 1 for outer loop test. Leave
                          inner loop when we next get to LOOP .
      ELSE                «
        0                 « No. Leave 0 for outer loop test.
      THEN
    LOOP                  « Inner loop end.
  IF                      « Did we have a match?

```

```

      1
      LEAVE
    THEN
  LOOP
  DEPTH R> -
  2 = IF
    DROP DROP 0
  THEN
;

```

« Yes. Leave *true* (1) [stack now: *c r 1*].  
« Leave outer loop at **LOOP** .  
« Outer loop end.  
« Get current stack depth, previous depth and difference.  
« # cells on stack out of loops = 2?  
« Yes. Loop exhausted with no match. **DROP** everything and leave only *false* (0).  
« Outer loop end.

The following example from the graphics primitives of the resident dictionary uses decimal numbers instead of hexadecimal. It initializes the screen in multicolor graphics mode.

Note that **I** (containing loop's index) on the fourth line is the same index as **J** (next outer loop's index) on the eighth line and *not* the same as **I** on the eighth line. The definitions of **I** and **J** are not equivalent; but, in this situation they reach the same cell on the return stack to get the index of the outer loop:

```

: MINIT ( --- )
  24 0 DO
    0
    I 4 / 32 *
    DUP 32 +
    SWAP
    DO
      DUP J 1 I HCHAR
      1+
    LOOP
    DROP
  LOOP
;

```

« Define **MINIT** to initialize multicolor mode. It takes no parameters and leaves none.  
« Row loop: 24 = loop limit; 0 = index start.  
« Initialize column counter on stack for use in inner loop.  
« Calculate inner loop index start from current value of outer loop's index **I** .  
« **DUP** it and add 32 to get inner loop limit.  
« Now, inner loop index start is on top of stack.  
« Char# loop.  
« Get 4 values to stack for use by **HCHAR** : **DUP** column counter, get row from index **J** of outer loop; **1** char; char# **I** .  
« Increment column counter left on stack.  
« Inner loop end.  
« **DROP** column counter still on stack.  
« Outer loop end.

### 2.5.7 DO ... +LOOP

```

DO ... +LOOP
  DO ( lim strt --- )
  +LOOP ( n --- )

```

**DO** as above. **+LOOP** adds top stack value to loop counter (index).

There may be times you will want your loop index to step by more than 1 or to step down instead of up. For that, you need **+LOOP** .

The following example is the definition of the **fbForth** word **.S** , which nondestructively displays the stack contents. **.S** starts by displaying “| ” to indicate the bottom of the stack. It then displays the numbers starting at the bottom of the stack, which is marked by the value in user variable **S0** .

The reason we need **+LOOP** is that, though we say that **S0** marks the bottom of the stack, in actuality it is a roof because the stack grows downward from high memory. The first cell on the stack is the first step below this roof. If there is at least one number on the stack and you want to read it, you would need to *subtract* 2 from the value in **S0** to get its address. The upshot of all this is that we need a loop that decrements the stack address by 2:

```

: .S ( --- )    « Define .S to nondestructively display the stack contents. It takes no
                « parameters and leaves none.
CR              « Start display on new line.
SP@ 2-         « Get address of top of stack and go 1 cell beyond, which will be the
                « loop limit.
S0 @ 2-       « Get address of stack base and adjust to address of first cell, which will
                « be the loop index start.
." | "        « Display "|".
OVER OVER     « Duplicate loop limit and start.
= 0= IF       « Are they =? If they are, the stack is empty and we don't want to go
                « through the loop, so we test that result for falsity with 0=. Now the
                « question for IF is, "Are they ≠?"
DO            « Yes—they are ≠.
  I @ U.      « The index I is the address of the current stack cell. Get its contents
                « and display it as an unsigned number in the current radix.
  -2 +LOOP    « Loop end. Add -2 to the loop index to get the next stack cell's address
ELSE          « No—we have an empty stack.
  DROP DROP   « DROP the 2 numbers DO didn't get to use so we don't pollute the stack.
THEN
;

```

### 2.5.8 CASE ... OF ... ENDOF ... ENDCASE

<pre> CASE   n<sub>1</sub> OF ... ENDOF   n<sub>2</sub> OF ... ENDOF   ...   n<sub>m</sub> OF ... ENDOF   ... ENDCASE CASE ( n --- ) </pre>	<p>Looks for a number (<math>n_1, n_2, \dots, n_m</math>) matching <math>n</math>. 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 <math>n</math> from the stack. Execution then continues after <b>ENDCASE</b>. Code after the last <b>ENDOF</b> may use <math>n</math>, which is still available; but, it must not consume <math>n</math>. Otherwise, <b>ENDCASE</b> will drop whatever was under <math>n</math>, adversely affecting program logic and possibly causing a stack underflow.</p>
---	---

The **CASE** structure allows you to select one of many courses of action based on a single value. It is much neater and easier to read than what would result if you attempted the same thing with a series of **IF** and **ELSE** clauses. It is also much less prone to error.

The following example is from the graphics primitives that are now part of the resident dictionary. It uses the console's keyboard scanning routine **KSCAN** to check for joystick and fire-button status of left and right joysticks or corresponding keys on left and right sides of the keyboard:

```

HEX
: JKBD
  ( kbd --- chr xst yst )
      8374 C!
      ?KEY DROP 8375 C@
      DUP 12 =
      OVER 0FF =
      OR IF
        8377 C@ 8376 C@
      ELSE
        DUP
        CASE
          04 OF 0FC 4 ENDOF « chr = 4 (NW)? xst = FCh, yst = 4
          05 OF 0 4 ENDOF « chr = 5 (N)? xst = 0, yst = 4
          06 OF 4 4 ENDOF « chr = 6 (NE)? xst = 4, yst = 4
          02 OF 0FC 0 ENDOF « chr = 2 (W)? xst = FCh, yst = 0
          03 OF 4 0 ENDOF « chr = 3 (E)? xst = 4, yst = 0
          0F OF 0FC 0FC ENDOF « chr = Fh (SW)? xst = FCh, yst = FCh
          00 OF 0 0FC ENDOF « chr = 0 (S)? xst = 0, yst = FCh
          0E OF 4 0FC ENDOF « chr = Eh (SE)? xst = 4, yst = FCh
          DROP DROP 0 0 0 0 « Illegal chr: Drop both copies and leave four 0s.
        ENDCASE
      THEN
        0 8374 C! « Restore previous keyboard #.
;

```

A more extensive example of the **CASE** structure appears in FBLOCKS in the 64-column editor (**EDT** in block 12). **EDT** is set up with an infinite **BEGIN ... AGAIN** loop that continuously monitors the keyboard until the exit key, **<FCTN+9>**, is struck. **<FCTN+9>**'s ASCII value is **0Fh**, so the **OF** clause that follows **0Fh** executes its contents, ultimately executing **QUIT** to get back to the terminal command line interpreter.

## 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. **.BASE** is the best way for the user to determine the current radix because **BASE @** will always display **10**.

.	( <i>n</i> --- )	Print a signed number
."	( --- )	Print a string terminated by "
.BASE	( --- <i>n</i> )	Print the decimal value of the current radix (number base)
.R	( <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>
?KEY	( --- <i>n</i> )	Read keyboard. No key? <i>n</i> = 0. Key? <i>n</i> = ASCII keycode.

<b>?TERMINAL</b>	( --- <i>flag</i> )	Test if <b>&lt;BREAK&gt;</b> ( <b>&lt;CLEAR&gt;</b> on TI-99/4A) pressed
<b>BASE</b>	( --- <i>addr</i> )	System variable containing number base. To set some base (e.g., Octal) use the following sequence from any base above Octal: <b>8 BASE !</b>
<b>COUNT</b>	( <i>addr</i> --- <i>addr+1</i> <i>n</i> )	Move length byte from a packed character string <sup>4</sup> at <i>addr</i> to stack and increment <i>addr</i> —suitable for <b>TYPE</b>
<b>CR</b>	( --- )	Perform a Carriage Return + Line Feed
<b>D.</b>	( <i>d</i> --- )	Print double-precision (DP) number
<b>D.R</b>	( <i>d n</i> --- )	Print DP number right-justified in field of width <i>n</i>
<b>DECIMAL</b>	( --- )	Sets the base to Decimal (Base 10)
<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>HEX</b>	( --- )	Sets the base to Hexadecimal (Base 16)
<b>KEY</b>	( --- <i>c</i> )	Wait for a keystroke and put its ASCII value on the stack.
<b>SPACE</b>	( --- )	Type 1 space
<b>SPACES</b>	( <i>n</i> --- )	Type <i>n</i> spaces
<b>TYPE</b>	( <i>addr n</i> --- )	Type <i>n</i> characters from <i>addr</i> to terminal
<b>U.</b>	( <i>u</i> --- )	Print an unsigned number
<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>HOLD</b>	( <i>c</i> --- )	Insert ASCII character <i>c</i> into string
<b>#&gt;</b>	( <i>d</i> --- <i>addr u</i> )	Terminate conversion, ready for <b>TYPE</b>

Formatting is always right to left. Consider that you wish to display a formatted Social Security Number that is on the stack as the double number, 123456789. The following would do the trick:

```
<# # # # # 45 HOLD # # 45 HOLD # # # #> CR TYPE
123-45-6789 ok:0
```

<sup>4</sup> A packed character string is a string of characters with a leading length byte. Several **fbForth** words expect or produce such strings.

Note that the format as you read the Forth code is the reverse of what is displayed and that 45 is the decimal value for the ASCII character ‘-’. See the individual definitions, especially **<#**, in Appendix D “The fbForth 2.0 Glossary” for more information.

## 2.8 Block-Related Words

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

<b>B/BUF</b>	( --- <i>n</i> )	Constant: Block size in bytes (always 1024 in <b>fbForth</b> )
<b>BLK</b>	( --- <i>addr</i> )	User variable containing current block number (contains 0 for terminal input)
<b>BLOCK</b>	( <i>n</i> --- <i>addr</i> )	Leave address of block <i>n</i> , reading it from the current blocks file if necessary
<b>CLEAR</b>	( <i>n</i> --- )	Fill block <i>n</i> with blanks
<b>CLR_BLKs</b>	( <i>n</i> <sub>1</sub> <i>n</i> <sub>2</sub> --- )	<b>CLEAR</b> a range of blocks from block <i>n</i> <sub>1</sub> to block <i>n</i> <sub>2</sub>
<b>CPYBLK</b>	( --- )	Copy a range of blocks from one blocks file to the same or a different blocks file from information in input stream
<b>EMPTY-BUFFERS</b>	( --- )	Erase all buffers
<b>FLUSH</b>	( --- )	Write all updated (dirty) buffers to disk
<b>LIST</b>	( <i>n</i> --- )	List block <i>n</i> to terminal
<b>LOAD</b>	( <i>n</i> --- )	Interpret block <i>n</i>
<b>MKBFL</b>	( --- )	Create a blocks file from string and number in input stream
<b>SCR</b> <sup>5</sup>	( --- <i>addr</i> )	User variable containing block number most recently referenced by <b>LIST</b> or <b>EDIT</b>
<b>UPDATE</b>	( --- )	Mark last buffer accessed as updated (dirty)
<b>USEBFL</b>	( --- )	Select a different blocks file from input stream

<sup>5</sup> The name of the word **SCR** is a throwback to Forth systems like TI Forth that used low-level disk block I/O for Forth blocks/screens. It is so named to refer to an editable Forth screen because a screen was not required to be equivalent to a block in figForth. A block was defined as the chunk (block) of disk space read/written in the process of accessing Forth screens and was not required to be as large as a screen. A screen was composed of one or more disk blocks. For **fbForth**, ‘block’ is synonymous with ‘screen’ and contains exactly 1024 bytes regardless of the chunk (now a 128-byte file record instead of a disk block) read/written from/to a blocks file. Each **fbForth** block access processes 8 records/block. **SCR** was retained simply because it made coding **fbForth** easier.

## 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 **<BUILDS ... DOES>ASM:** , to create new defining words.

<b>:</b> <b>xxx</b>	( --- )	Begin colon definition of <b>xxx</b> <sup>6</sup>
<b>;</b>	( --- )	End colon definition
<b>VARIABLE 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 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 xxx ... NEXT,</b>	( --- )	Define assembly language primitive named <b>xxx</b>
<b>ASM: xxx ... ;ASM</b>	( --- )	Ditto: <b>ASM:</b> ≡ <b>CODE</b> and <b>;ASM</b> ≡ <b>NEXT,</b>
<b>: xxx &lt;BUILDS ...     ;CODE ... NEXT,</b>		Create new defining word <b>xxx</b> with execution-time assembly/machine code routine
<b>: xxx &lt;BUILDS ...     DOES&gt;ASM: ... ;ASM</b>		Ditto: <b>DOES&gt;ASM:</b> ≡ <b>;CODE</b> and <b>;ASM</b> ≡ <b>NEXT,</b>
<b>: xxx &lt;BUILDS ...     DOES&gt; ... ;</b>		Create new defining word <b>xxx</b> with execution-time high level Forth routine

## 2.10 Miscellaneous Words

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

<b>' xxx</b>	( --- <i>addr</i> )	Leave parameter field address ( <i>pfa</i> ) of <b>xxx</b> . If compiling, compile address. (tick)
<b>(</b>	( --- )	Begin comment. Terminated by <b>)</b>
<b>,</b>	( <i>n</i> --- )	Compile <i>n</i> into the dictionary (comma)
<b>ABORT</b>	( --- )	Error termination
<b>ALLOT</b>	( <i>n</i> --- )	Leave <i>n</i> -byte gap in dictionary
<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>DEFINITIONS</b>	( --- )	Set <b>CURRENT</b> to <b>CONTEXT</b>
<b>FORGET xxx</b>	( --- )	Forget all definitions back to and including <b>xxx</b> <sup>6</sup>
<b>FORTH</b>	( --- )	Set <b>CONTEXT</b> to main Forth vocabulary

<sup>6</sup> If you wish to **FORGET** an unfinished definition, the word likely will not be found. If it is the last definition attempted, you can make it findable by executing **SMUDGE** and then **FORGET**ting it.

<b>HERE</b>	( --- <i>addr</i> )	Leaves address of next unused byte in the dictionary
<b>IN</b>	( --- <i>addr</i> )	User variable containing offset into input buffer
<b>PAD</b>	( --- <i>addr</i> )	Leaves address of scratch area (68 bytes above <b>HERE</b> )
<b>SP@</b>	( --- <i>addr</i> )	Leaves address of top stack item
<b>VOCABULARY xxx</b>	( --- )	Define new vocabulary

Many additional words are available in **fbForth 2.0**. The user should consult the remaining chapters in this manual as well as the glossary ( Appendix D ) and Appendix G for a complete description. Many of these words are defined in FBLOCKS 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 fbForth 2.0 Editors

Words introduced in this chapter:

<b>CLEAR</b>	<b>EDIT</b>	<b>TEXT</b>
<b>CLR_BLKs</b>	<b>EMPTY-BUFFERS</b>	<b>TEXT80</b>
<b>CPYBLK</b>	<b>FLUSH</b>	<b>USEBFL</b>
<b>ED@</b>	<b>MKBFL</b>	<b>WHERE</b>

In the Forth language, programs are divided into blocks. Each Forth block is 16 lines of 64 characters and has a number associated with it. A single-sided single-density (SSSD) TI-99/4A disk that contains a single DF128<sup>7</sup> blocks file that fills the disk can hold 89 Forth blocks (numbered 1<sup>8</sup> – 89). There will actually be one sector (256 bytes) left because disk and file overhead occupy 3 sectors and the blocks file occupies 356 sectors (89 · 4), which leaves one sector of a possible 360 unoccupied. A program may occupy as many Forth blocks as necessary.

If you plan to edit the system blocks file, FBLOCKS, you should back it up with a suitable disk manager program or a combination of **MKBFL** (see below) and **CPYBLK** (see § 3.5 “Block-Copying Utility”) before modifying it.

The editor uses the current blocks file, which is DSK1.FBLOCKS at system startup. You can change the current blocks file to one of your choosing, *e.g.*, DSK2.MYBLOCKS, with **USEBFL** by typing on the terminal:

```
USEBFL DSK2.MYBLOCKS ok:0
```

If DSK2.MYBLOCKS does not exist, you must first create it with an appropriate number of blocks by executing **MKBFL**, being careful not to exceed the capacity of the disk, followed by **USEBFL**:

```
MKBFL DSK2.MYBLOCKS 80 ok:0  
USEBFL DSK2.MYBLOCKS ok:0
```

Now you are ready to begin editing the selected blocks file.

### 3.1 Forth Block Layout Caveat

As indicated above, Forth blocks 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-block 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 block 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**.

<sup>7</sup> DF128 refers to the file format: **D**isplay data type, **F**ixed record length, **128**-byte logical record length

<sup>8</sup> For **fbForth**, the first block of a blocks file is always numbered 1. This is different from most figForth systems, including TI Forth, which start at block number 0.



The full block is displayed in **TEXT80** mode.

```

BLOCK #2
0 0 0 0 0 0 0 0
0 CLS 0 0 GOTOXY ." Load Options (29NOV2014) fbForth 2.0" CR CR
1 0 ." Description Load Block" CR
2 0 -----" CR
3 ." CPYBLK -- Block Copying Utility.....4" CR
4 ." CRU Words.....5" CR
5 ." 64-Column Editor.....6" CR
6 ." Memory Dump Utility.....16" CR
7 ." TRACE -- Colon Definition Tracing...18" CR
8 ." Printing Routines.....19" CR
9 ." TMS9900 Assembler.....21" CR
10 ." More Useful Stack Words etc.....41" CR
11 ." Stack-based String Library.....42" CR
12 ." Disk Catalog Utility.....36" CR
13 ." TMS9900 Assembler BLOAD (clears RAM)..27" CR
14 ." 64-Column Editor BLOAD (clears RAM)..32" CR
15 ." String Library BLOAD (clears RAM)....52" CR CR -->

F1:Del F2:Ins F3:Del Line F4:Nxt Block F5:30 spcs L/R F6:Prv Block F7:Del EOL
F8:Ins Line F9:Exit ESDX:Cursor ^V:Tab ENT:Nxt Line ^8:Ins Blank Line FV:-Tab

```

The second, which is loaded by **6 LOAD**, operates in **SPLIT** mode, a modified bitmap mode. It allows you to view an entire block at once on a 40-column screen; however, the characters are very small. It will be referred to as the 64-column editor.

```

CLS 0 0 GOTOXY ." Load Options (29NOV2014) fbForth 2.0" CR CR
0 ." Description Load Block" CR
1 -----" CR
2 ." CPYBLK -- Block Copying Utility.....4" CR
3 ." CRU Words.....5" CR
4 ." 64-Column Editor.....6" CR
5 ." Memory Dump Utility.....16" CR
6 ." TRACE -- Colon Definition Tracing...18" CR
7 ." Printing Routines.....19" CR
8 ." TMS9900 Assembler.....21" CR
9 ." More Useful Stack Words etc.....41" CR
10 ." Stack-based String Library.....42" CR
11 ." Disk Catalog Utility.....36" CR
12 ." TMS9900 Assembler BLOAD (clears RAM)..27" CR
13 ." 64-Column Editor BLOAD (clears RAM)..32" CR
14 ." String Library BLOAD (clears RAM)....52" CR CR -->
BLOCK #2

```

If you load the 64-column editor, that is the only one you will be able to use. If, after you load it, you wish to use the 40/80-column editor, you will need to remove the 64-column editor with

**FORGET TCHAR ok:0**

or by restarting with **COLD** or **MON**. Use whichever editor you prefer. Editing instructions are identical for each.

### 3.3 Editing Instructions

You should insure that the blocks you are editing are filled with only displayable characters (blanks, if starting from scratch). If you just created the file you are editing with **MKBFL**, all blocks have already been filled with blanks. A single block may be filled with blanks before it is edited by typing a block number and **CLEAR**:

**1 CLEAR ok:0**

will prepare block 1 for use by the editor.

A range of blocks may be cleared to blanks by executing **CLR\_BLKs** with the first and last blocks of the range on the stack:

**1 5 CLR\_BLKs ok:0**

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

**1 EDIT**

The above instruction will bring the contents of block 1 into view. If you did not **CLEAR** the block before entering the editor and the block contains non-displayable characters or other undesirable information, it may be easier to simply exit the editor temporarily and clear the block before writing to it. To exit the editor, press the **<BACK>** (**<FCTN+9>**) function key on your keyboard. To clear the block, type the block number and the word **CLEAR** as above.

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

**ED@**

will return you to the last block you were editing.

Upon entering the editor, the cursor is located in column 0 of line 0. It is customary to use line 0 for a comment describing the contents of that block. Type a comment that says “**PRACTICE BLOCK**” or something to that effect. Do not forget that all comments must begin with a ‘ ( ’<sup>11</sup> and end with a ‘ ) ’.

If you are using the 40/80-column editor in **TEXT** mode, you have probably noticed that only 35 columns (0–34) of the 64 available columns are visible on your terminal. To see the rest of the block, type any characters on line 1 until you reach the right margin. Now type a few more characters. Notice that the block is now displaying columns 29 – 63. Press **<ENTER>** to move to the beginning of the next line.

In the 40/80-column editor, you will notice that a keystroke menu is displayed at the bottom of the screen just below the editing window. Though it is cryptic, it should aid in remembering the keystrokes for the editing commands. This feature was inspired by Mark Wills’ *TurboForth* (see [turboforth.net](http://turboforth.net)) and the idea and some code was used with his permission.

---

<sup>11</sup> The left parenthesis *must* be followed by at least 1 space. Press **<ENTER>** to move to the next line.

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

key	function
<FCTN+S>, (←)	moves the cursor one position to the left.
<FCTN+D>, (→)	moves the cursor one position to the right.
<FCTN+E>, (↑)	moves the cursor up one position.
<FCTN+X>, (↓)	moves the cursor down one position.
<DELETE> (<FCTN+1>)	deletes the character on which the cursor is placed.
<INSERT> (<FCTN+2>)	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.
<AID> (<FCTN+7>)	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 <REDO>. <REDO> inserts a line just above where the cursor is and places the contents of <b>PAD</b> there.
<BEGIN> (<FCTN+5>)	<b>40/80-column editor:</b> in <b>TEXT</b> mode, moves the cursor 29 positions to the right if the cursor is on the left half of a block. Otherwise, it moves the cursor 29 positions to the left. This key can be used to toggle between the left and right half of a block. In <b>TEXT80</b> mode, places the cursor in the upper left corner. <b>64-column editor:</b> places the cursor in the upper left corner
<ERASE> (<FCTN+3>)	are used in combination to pick up lines and move them elsewhere on the screen. <ERASE> picks up one line while erasing it from view.
<REDO> (<FCTN+8>)	<REDO> inserts this line just above the line on which the cursor is placed. Both <ERASE> and <REDO> may be used repeatedly to erase several lines from view or to insert multiple copies of a line.
<CTRL+8>	will insert a blank line just above the line the cursor is on.
<CTRL+V>	will tab forward by words.
<FCTN+V>	will tab backwards by words.

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 block without leaving the editor. Pressing <CLEAR> (<FCTN+4>) will read in the succeeding block. Pressing <PROCEED> (<FCTN+6>) will read in the preceding block.

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 in the blocks file to be written to the file. Use this word at the end of an editing session to be certain your changes are written to the disk. The word **EMPTY-BUFFERS** can be used to clear all Forth buffers and thereby undo any unsaved changes. This is not guaranteed to work except on the current block due to how the editors function when acquiring buffer space.

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

### 3.4 Changing Foreground/Background Colors of 64-Col Editor

The black-on-gray color scheme of the 64-column editor and the white-on-dark-blue colors of the 8-line text area at the bottom of the screen can be changed to whatever foreground/background combinations you would like by making minimal changes to Forth code on block 12 of FBLOCKS. There are three chunks of commented-out code on lines 1 and 2 (see following) that offer templates for changing the editor's colors, the screen background color and the 8-line text area, in that order:

```
1: : EDT   VDPMDE @ >R SPLIT ( 0 1000 040 VFILL) ( 0F 7 VWTR)
2: ( 1000 800 01B VFILL) CINIT !CUR R/C CGOTOXY
```

If you want to change the editor's colors to dark blue on transparent, un-comment the first chunk of code,

```
( 0 1000 040 VFILL)
```

by removing the parentheses:

```
0 1000 040 VFILL
```

For some other combination, change **040** to **0XY**, where **X** is the desired hexadecimal digit for the foreground color and **Y** is the desired background color digit.

To change the screen color (including the border color), un-comment the second chunk of code,

```
( 0F 7 VWTR)
```

which will change the screen color to white:

```
0F 7 VWTR
```

If you do not want white, change the **0F** to the screen color of your choice.

The final chunk of commented code,

```
( 1000 800 01B VFILL)
```

only requires un-commenting to get black on yellow for the bottom 8-line section:

```
1000 800 01B VFILL
```

Change the **01B** to any desired combination of colors as described above for the editor's colors.

You may also want to change the color of the 64-column editor's cursor from white to some other color that makes sense with your new color scheme. If so, you will need to change the color of the cursor sprite in the word **CINIT** (block 7) from **0 1 F 5 0 SPRITE** to **0 1 new\_color 5 0 SPRITE**, where *new\_color* is your new color (see § 6.3 "Color Changes").

### 3.5 Block-Copying Utility

You can copy a range of blocks to the same or another blocks file with **CPYBLK**. This utility is not part of the resident dictionary, so you will need to load block 19 (**19 LOAD**) from

FBLOCKS. Typing **MENU** will show you this option as well as ensure that FBLOCKS is the current blocks file. Usage instructions are displayed after **CPYBLK** is loaded:

**19 LOAD**

**CPYBLK copies a range of blocks to the same or another file, e.g.,**

**CPYBLK 5 8 DSK1.F1 9 DSK2.F2**  
**will copy blocks 5-8 from DSK1.F1 to DSK2.F2 starting at block 9.**

**ok:0**

It should be noted that **CPYBLK** will safely copy overlapping source and destination block ranges when the source and destination files are the same. First, **CPYBLK** checks to see whether the source and destination files are the same. If they are, it next checks to see whether the ranges overlap. If they do, it checks to see whether the number of blocks to be copied exceeds the distance between start blocks of source and destination. If it does, then, and only then, it will change the direction of copying to be end to start blocks. It will also reverse the start and end block numbers if you enter a larger number for the start block than for the end block.

If something goes wrong, you may need to restore to current status the blocks file you were using before you invoked **CPYBLK** . See **USEBFL** in Appendix D .

## 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>12</sup>

The VDP memory can be configured in many ways by the user. The **fbForth 2.0** system provides the ability to set up this memory for each of the VDP's 5 modes of operation (Text80, Text, Graphics, Multicolor and Graphics2). The allocation of memory for these modes is shown on the VDP Memory Map. The first four modes are shown on the left side of the figure, the Graphics2 mode on the right side. The area at **03C0h** is used by the GPL transcendental functions in all modes for a rollout area, which was a problem for TI Forth and **fbForth 1.0**. Fortunately, you do not need to worry about this now because **fbForth 2.0**'s floating point math package does not use them (see Chapter 7 The Floating Point Support Package). 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 **fbForth 2.0** system. The use of the remainder of the RAM in this area is dictated by the TI-99/4A's resident operating system.

The 32KB memory expansion is divided into an 8KB piece at **2000h** and a 24KB piece at **A000h**. The small piece contains BIOS and utility support for **fbForth 2.0** as well as 4KB of disk buffers, the Return Stack and the User Variable area. The large piece of this RAM contains the user dictionary, the Parameter Stack and the Terminal Input Buffer.

### 4.1 VDP Memory Map

Address				Address
<b>0000h</b>	Graphics & Multicolor Screen Image Table <i>bytes: 300h</i>	Text Screen Table	Mode Image Table	<b>1800h</b>
<b>0300h</b>	Sprite Attribute List <b>80h</b>	40 Columns <b>TEXT</b> <b>3C0h</b>	80 Columns <b>TEXT80</b> <b>780h</b>	<b>0000h</b>
<b>0380h</b>	Color Table <b>20h</b>			
<b>03A0h</b>	Unused <b>20h</b>			
<b>03C0h</b>	VDP Rollout Area <b>20h</b>			
<b>03E0h</b>	Value Stack <b>80h</b>			
<b>0460h</b>	PABS etc. <b>320h</b>			
<b>0780h</b>	Sprite Motion Table <b>80h</b> [ <i>Value Stack for TEXT80</i> ]			

<sup>12</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 blocks. 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>0800h</b>	Pattern & Sprite Descriptor Tables 0 – 127 <b>400h</b>	
<b>0C00h</b>	128 – 255 <b>400h</b>	
<b>1000h</b>	<b>fbForth</b> 's Disk Buffer <b>80h</b>	
<b>1080h</b>	PAB: User Screen Font File <b>46h</b>	
<b>10C6h</b>	PAB: Current Blocks File <b>46h</b>	
<b>110Ch</b>	PAB: Second Blocks File <b>46h</b>	
<b>1152h</b>	Unused <i>[PABS points here for <b>TEXT80</b>]</i>  <b>2686h</b>	
	Bitmap Screen Image Tab. <b>300h</b>	<b>1800h</b>
	Sprite Attribute List <b>80h</b>	<b>1B00h</b>
	<b>fbForth</b> 's Disk Buffer <b>80h</b>	<b>1B80h</b>
	PAB: User Screen Font File <b>46h</b>	<b>1C00h</b>
	PAB: Current Blocks File <b>46h</b>	<b>1C46h</b>
	PAB: Second Blocks File <b>46h</b>	<b>1C8Ch</b>
	User PABs, etc. <b>2EEh</b>	<b>1CD2h</b>
	Stack for VSPTR <b>40h</b>	<b>1FC0h</b>
	Bitmap Pattern Descriptor Table <b>1800h</b>	<b>2000h</b>
<b>37D8h</b>	Disk Buffer Region for 3 Simultaneous Disk Files <b>828h</b>	
	Sprite Descriptor Table <b>1DEh</b>	<b>3800h</b>
<b>3FFFh</b>	Disk Buffer Region: 2 Files <b>622h</b>	<b>39DEh</b> <b>3FFFh</b>

### 4.2 CPU Memory

Address	
<b>0000h</b>	Console ROM
<b>2000h</b>	Low Memory Expansion Loader, Your Program, REF/DEF Table
<b>4000h</b>	Peripheral ROMs for DSRs
<b>6000h</b>	<b>fbForth 2.0</b> ROMs (including Resident Dictionary) in Command Module
<b>8000h</b>	Memory-mapped Devices for VDP, GROM, SOUND, SPEECH. CPU RAM at <b>8300h – 83FFh</b>
<b>A000h</b> <b>FFFFh</b>	High Memory Expansion Your Program (up to parameter stack & TIB at high end)

### 4.3 CPU RAM Pad

Address <sup>13</sup>	
8300h	<b>fbForth</b> 's Workspace (see § 9.2)
831Fh	
8320h	–FREE– <b>Eh</b>
832Dh	
832Eh	<b>fbForth</b> 's Inner Interpreter, etc.
8347h	
8348h	–FREE– <b>2</b>
8349h	
834Ah	FAC (Floating Point Accumulator)
8351h	
8354h	Floating Point Error
8355h	Floating Point String↔Number Conversion Options
8356h	Subroutine Pointer for DSRs    use these 3 bytes
8357h	
835Ch	ARG (Floating Point Argument Register)
8363h	
836Eh	VSPTR (Value Stack Pointer)
836Fh	
8370h	Highest Available Address of VDP RAM
8371h	
8372h	Least Significant Byte of Data Stack Pointer
8373h	Least Significant Byte of Subroutine Stack Pointer
8374h	Keyboard Number to be Scanned
8375h	ASCII Keycode Detected by Scan Routine
8376h	Joystick Y-status
8377h	Joystick X-status
8379h	VDP Interrupt Timer
837Ah	Number of Sprites that can be in Automotion
837Bh	VDP Status Byte    Bit 0 <sup>14</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
837Ch	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
837Dh	VDP Character Buffer
837Eh	Current Screen Row Pointer
837Fh	Current Screen Column Pointer
8380h	Default Subroutine Stack
83A0h	Default Data Stack
83C0h	Random Number Seed (Begin Interrupt Workspace)
83C2h	Flag    Bit 0    Disable All of the Following Bit 1    Disable Sprite Motion

<sup>13</sup> Locations omitted are not used by **fbForth**, but may be used by system routines.

<sup>14</sup> Bit 0 = high order bit.

Address			
	Bit 2	Disable Auto Sound	
	Bit 3	Disable System Reset Key (Quit)	
<b>83C4h</b>	Link to ISR Hook		
<b>83C6h</b>	Default keyboard argument – 3 ( <i>i.e.</i> , 0 – 2)		
<b>83C7h</b>	Keyboard column 0 (special keys)		
<b>83C8h</b>	Scan code of current key, whatever keyboard type		
<b>83C9h</b>	Ditto for keyboard type 4 (Pascal)		
<b>83CAh</b>	Ditto for keyboard type 5(Standard) [Keyboard Debounce?]		
<b>83CCh</b>	Sound List Pointer (VDP RAM)		
<b>83CEh</b>	Sound List Initiation (set to <b>01h</b> ) & Countdown Byte		
<b>83D0h</b>	Search Pointers for GROM & ROM		
<b>83D4h</b>	Contents of VDP Register 1		
<b>83D6h</b>	Screen Timeout Counter		
<b>83D8h</b>	Return Address Saved by Scan Routine		
<b>83DAh</b>	Player Number Used by Scan Routine		
<b>83E0h</b>	G	R0	«Data (Src)
<b>83E2h</b>	P	R1	«Address (Src)
<b>83E4h</b>	L	R2	«Data (Dst)
<b>83E6h</b>	W	R3	«Address (Dst)
<b>83E8h</b>	o	R4	«MSB: (Src Flag) LSB: (Dst Flag)
<b>83EAh</b>	r	R5	«MSB: Word Command Flag
<b>83ECh</b>	k	R6 – R8	
<b>83F2h</b>	s	R9	«MSB: GPL Code
<b>83F4h</b>	p	R10 – R12	
<b>83FAh</b>	a	R13	«Current GROM Port ( <b>9800h</b> )
<b>83FCh</b>	c	R14	«Timer Tick & Flags
<b>83FEh</b>	e	R15	«VDPWA ( <b>8C02h</b> )

#### 4.4 Low Memory Expansion

<b>2000h</b>	XML Vectors	<b>0010h</b> bytes
<b>2010h</b>	<b>fbForth</b> Block Buffers (4)	<b>1010h</b>
<b>3020h</b>	99/4 Support for <b>fbForth</b>	<b>0640h</b>
<b>3660h</b>	User Variable Area	<b>0080h</b>
<b>36E0h</b>	Assembler Support, Trampoline Code, ...	<b>0322h</b>
<b>3A04h</b>	↑	<b>05FCh</b>
<b>3FFFh</b>	Return Stack	

#### 4.5 High Memory Expansion

<b>A000h</b>	End of Resident <b>fbForth</b> Vocabulary	<b>0030h</b>
<b>A030h</b>	User Dictionary Space ↓ ↑ Parameter Stack	<b>5F70h</b>
<b>FFA0h</b> <b>FFF1h</b>	Terminal Input Buffer	<b>0052h</b>

## 5 System Synonyms and Miscellaneous Utilities

Words introduced in this chapter:

'	<b>RANDOMIZE</b>	<b>VFILL</b>
,	<b>RND</b>	<b>VLIST</b>
.S	<b>RNDW</b>	<b>VMBR</b>
: (traceable)	<b>SEED</b>	<b>VMBW</b>
C,	<b>TRACE</b>	<b>VMOVE</b>
CLS	<b>TRIAD</b>	<b>VOR</b>
DSRLNK	<b>TRIADS</b>	<b>VSBR</b>
DUMP	<b>TROFF</b>	<b>VSBW</b>
GPLLNK	<b>TRON</b>	<b>VWTR</b>
INDEX	<b>UNTRACE</b>	<b>VXOR</b>
MYSELF	<b>VAND</b>	<b>XMLLNK</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 block-listing 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 part of the resident dictionary in **fbForth 2.0**. 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 **fbForth 2.0** words implements the Editor/Assembler (E/A) utility with the same name.

Two words have no E/A equivalent: **VFILL** was introduced in TI Forth and **VMOVE** was new in **fbForth 1.0**.

**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	<i>byte</i>	<i>vaddr</i>	instr
<b>HEX</b>	<b>A3</b>	<b>380</b>	<b>VSBW</b>

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	<i>addr</i>	<i>vaddr</i>	<i>count</i>	instr
<b>HEX</b>	<b>PAD</b>	<b>808</b>	<b>4</b>	<b>VMBW</b>

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	<i>vaddr</i>	instr
<b>HEX</b>	<b>781</b>	<b>VSBR</b>

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	<i>vaddr</i>	<i>addr</i>	<i>count</i>	instr
<b>HEX</b>	<b>300</b>	<b>PAD</b>	<b>20</b>	<b>VMBR</b>

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	<i>vaddr</i>	<i>count</i>	<i>byte</i>	<i>instr</i>
	HEX	300	20	0	VFILL

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

**VMOVE** (*vaddr<sub>1</sub> vaddr<sub>2</sub> count ---*)

Copies *count* bytes from one location (*vaddr<sub>1</sub>*) in VDP RAM to another (*vaddr<sub>2</sub>*).

	base	<i>vaddr<sub>1</sub></i>	<i>vaddr<sub>2</sub></i>	<i>count</i>	<i>instr</i>
	HEX	1500	1640	100	VMOVE

copies 256 (100h) bytes from *vaddr<sub>1</sub>* to *vaddr<sub>2</sub>*. If the ranges overlap, it is only safe to copy from a higher address to a lower address because the copy proceeds from the lowest address of the source block to the highest. If the copy were in the other direction, all the bytes in the overlapping region would be trashed before they could be copied.

### 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*, § 16.2.2 – § 16.2.4 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 as well as provide what additional information that routine may require.

	base	set up FAC for call	<i>addr</i>	<i>instr</i>
	HEX	900 834A !	16	GPLLNK

branches to the GPL routine located at 16h which loads the TI-99/4A standard character set into VDP RAM. It then returns to your program.

**XMLLNK** (*addr ---*)

Allows you to link a Forth program to any executable machine-code routine with vectors in ROM or low-RAM (2000h) or to branch to a routine located in high RAM (8000h – FFFFh). The instruction expects to find the address of and offset into a ROM or low-RAM table or a high-RAM address on the stack.

	base	<i>addr</i>	<i>instr</i>
	HEX	800	XMLLNK

accesses the floating-point (FP) multiplication routine, located in console ROM. The *addr* value (800h) in this case is a reference to offset 10h into the console-ROM table for FP routines that starts at 0D1Ah. 0D1Ah is the first table pointed to in the XML jump table (0CFAh) in console ROM. Offset 10h (0D2Ah) of the FP table contains the address in

console ROM of said FP multiplication routine, which executes and returns to your program.

*Note:* The above FP multiplication routine requires the FP multiplier in FAC and the FP multiplicand in ARG. The product is returned in FAC. The **fbForth 2.0** FP library (Chapter 7) *no longer* uses the code in the above example for FP multiplication.

**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 **fbForth 2.0** 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	VWTR

The above instruction writes **F5h** into VDP write only register number 7. This particular register controls the foreground and background colors in text and text80 modes. The foreground color is ignored in other modes. 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 each 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	VAND
	HEX	F0	804	VOR
	HEX	F0	804	VXOR

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

**FORTH-COPY**, **DTEST**, **DISK-HEAD** and **FORMAT-DISK** are not supported in **fbForth 2.0**. If you need the functionality of these words, use one of the various disk manager cartridges or programs available such as TI's Disk Manager 2 cartridge, CorComp's Disk Manager, Quality 99 Software's Disk Manager III or Fred Kaal's Disk Manager 2000. You can, of course, use the above words in TI Forth.

**SCOPY** and **SMOVE** have been replaced by **CPYBLK**, which is described in § 3.5 "Block-Copying Utility".

## 5.3 Listing Utilities

There are three words defined in **fbForth 2.0** starting in block 19 of FBLOCKS, which make listing information from a Forth blocks file very simple. The following descriptions refer to FBLOCKS dated 01SEP2014 or later to insure that you can print the first 3 blocks. If the file contains a number of blocks not evenly divisible by 3, printing the last 1 or 2 blocks will cause a file error message to be printed when **TRIAD** tries to read past the end of the blocks file.

**TRIAD**           (*blk* ---)

The first, called **TRIAD**, requires a block number on the stack. When executed, it will end with a block number evenly divisible by three. Blocks 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**       (*blk*<sub>1</sub> *blk*<sub>2</sub> ---)

The second instruction, called **TRIADS**, may be thought of as a multiple **TRIAD**. It expects start and end block numbers on the stack. **TRIADS** executes **TRIAD** as many times as necessary to cover the specified range of blocks.

**INDEX**       (*blk*<sub>1</sub> *blk*<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 blocks. **INDEX** expects start and end block numbers 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

Loading block 16 loads two useful **fbForth** words for getting information for debugging purposes. Both **VLIST** and **DUMP** are 80-column aware if you have successfully executed **TEXT80** (see Chapter 3 “How to Use the fbForth 2.0 Editors” for some discussion of 80-column text mode).

**VLIST** ( --- )

The **fbForth** 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. When finished or aborted with **<BREAK>**, **VLIST** displays the number of words listed.

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

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

base	<i>addr</i>	<i>count</i>	instr
HEX	2010	20	DUMP

will list 32 (**20h**) bytes of memory beginning at address **2010h** to your terminal:

```

2010: 0001 2820 6662 466F ..( fbFo
2018: 7274 6820 5745 4C43 rth WELC
2020: 4F4D 4520 5343 5245 OME SCRE
2028: 454E 2D2D 2D4C 4553 EN---LES
ok:0

```

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>**.

A third word, **.S**, is part of **fbForth 2.0**'s resident dictionary and available at any time.

**.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 on the command line 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, *i.e.*, top of the stack:

```

.S
| 1 8 189 ok:3

```

### 5.4.2 Tracing Word Execution

This section is based on the following article available at [www.forth.org](http://www.forth.org) :

Paul van der Eijk. 1981. Tracing Colon-Definitions. *Forth Dimensions* 3:2, p. 58.

A special set of instructions in block 18 of FBLOCKS 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 ok:0
TRACE ok:0 (compile next definition with TRACE option)
: CUBE DUP DUP * * ; ok:0 (routine to be traced)
UNTRACE ok:0 (don't compile next definition with TRACE option)
: TEST CUBE ROT CUBE ROT CUBE ; ok:0
TRON ok:0 (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:3
.S (check final stack contents)
| 343 125 216 ok:3 (stack contents after final CUBE)

```

### 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, which prevents the word from being found during compilation. To allow recursion, **fbForth 2.0** 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:0
TRACE ok:0                (compile following definition under TRACE option)
: FACT DUP 1 > IF DUP 1 - MYSELF * ENDIF ; ok:0
UNTRACE ok:0
TRON ok:0
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:1
.S                        (check final stack contents)
↓ 120 ok:1

```

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

## 5.5 Random Numbers

Two different random number functions are available in **fbForth**. They are part of **fbForth**'s resident dictionary.

**RNDW** ( ---  $n$  )

The first random number function, **RNDW**, generates a random word (2 bytes). No range is specified for **RNDW**. The 16-bit (LSW) result of  $(6FE5h * seed + 7AB9h)$  is shifted circularly right 5 bits before being stored as the new value for *seed* (located at **83C0h**) and returned as  $n$  on the stack such that  $0 \leq n \leq FFFFh$ .

**RNDW ok:1**

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

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

The second, **RND**, generates a positive random integer between 0 and a specified range  $n_1$  by taking the absolute value of the result for **RNDW** above, dividing it by  $n_1$  and leaving the remainder on the stack as  $n_2$ .

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

will place on the stack an integer  $n_2$  such that  $0 \leq n_2 < 13$ .

**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. The seed is calculated by clearing the VDP status register by reading it at **8802h** and entering a counter loop that increments the counter and checks the VDP status register for the next VDP interrupt, at which point it exits the loop and stores the counter in the seed location **83C0h**.

**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 ok:0**

will place the value 4 into the random number generator seed location **83C0h**. This is particularly useful during testing because **RND** and **RNDW** will generate the same series of pseudo-random numbers every time they are started with the same seed.

## 5.6 Miscellaneous Instructions

' ( *--- pfa* )

' (tick) searches the **CONTEXT** vocabulary and then the **CURRENT** vocabulary in the dictionary for the next word in the input stream. If it is found, ' pushes the word's parameter field address *pfa* onto the stack. Otherwise, an error message is displayed and the contents of **IN** and **BLK** are left on the stack.

, ( *n* --- )

, (comma) stores *n* at **HERE** on an even address boundary in the dictionary, which includes the current value of **HERE** , and advances **HERE** one cell to the next even address. Comma is the primary compiling word in Forth.

**C,** ( *b* --- )

**C,** stores *b* at **HERE** . **C,** is the byte equivalent of , . Care must be taken when using **C,** to compile bytes into the dictionary because most storage to the dictionary is cell-oriented. If **HERE** is left on an odd address, a word like , will overwrite the previously stored byte!

**CLS** ( --- )

**CLS** is part of **fbForth**'s resident dictionary. 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 bitmap displays or sprites.

## 6 An Introduction to Graphics

Words introduced in this chapter:

<b>#MOTION</b>	<b>HCHAR</b>	<b>SPRCOL</b>
<b>BEEP</b>	<b>HONK</b>	<b>SPRDIST</b>
<b>CHAR</b>	<b>JCRU</b>	<b>SPRDISTXY</b>
<b>CHARPAT</b>	<b>JKBD</b>	<b>SPRGET</b>
<b>COINC</b>	<b>JMODE</b>	<b>SPRITE</b>
<b>COINCALL</b>	<b>JOYST</b>	<b>SPRPAT</b>
<b>COINCXY</b>	<b>LINE</b>	<b>SPRPUT</b>
<b>COLOR</b>	<b>MAGNIFY</b>	<b>SSDT</b>
<b>DELALL</b>	<b>MCHAR</b>	<b>TEXT</b>
<b>DELSPR</b>	<b>MINIT</b>	<b>TEXT80</b>
<b>DOT</b>	<b>MOTION</b>	<b>UNDRAW</b>
<b>DRAW</b>	<b>MULTI</b>	<b>VCHAR</b>
<b>DTOG</b>	<b>SCREEN</b>	<b>VDPMDE</b>
<b>GCHAR</b>	<b>SPCHAR</b>	<b>VMODE</b>
<b>GRAPHICS</b>	<b>SPLIT</b>	
<b>GRAPHICS2</b>	<b>SPLIT2</b>	

### 6.1 Graphics Modes

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

- 0) **Text80 Mode**—This is the same as text mode described below except that, in text80 mode, the screen is 80 columns by 24 lines. The user should insure that the system in use is capable of displaying 80-columns before invoking it, *i.e.*, it should be equipped with an F18A VDP (available at <http://codehackcreate.com/>) or similar device.
- 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 40/80-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) **Bitmap Mode (Graphics2)**—This mode is available only on the TI-99/4A. Bitmap mode allows you to set any pixel on the screen and to change its color within the limits permitted by the TMS9918a. The screen is 256 columns by 192 rows.

- 5) **Split Mode**—This mode is one of two unique graphics modes created by using graphics2 mode in a non-standard way. Split2 [see (6)] is the other non-standard variation of graphics2 mode. Split and split2 modes allow you to display text while creating bitmap graphics. Split mode sets the top two thirds of the screen in graphics2 mode and places text on the last third. Split mode is used by the 64-column editor.
- 6) **Split2 Mode**—This mode is the other of the two unique graphics modes created by using graphics2 mode in a non-standard way [see (5)]. Split2 sets the top one sixth of the screen as a text window and the rest in graphics2 mode.

Split and split2 modes provide an interactive bitmap graphics setting. That is, you can type bitmap instructions and watch them execute without changing modes.

Sprites (moving graphics) are available in all modes except text and text80. The sprite automotion feature is not available in graphics2, split, or split2 modes.

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

```

TEXT80      ( --- )
TEXT        ( --- )
GRAPHICS    ( --- )
MULTI       ( --- )
GRAPHICS2   ( --- )
SPLIT       ( --- )
SPLIT2      ( --- )
VMODE       ( n --- )      where n is one of the VDP mode numbers (0 – 6).
```

The following resident user variable holds a number corresponding to one of the above modes as enumerated above. It can be useful for programmatically determining the graphics mode:

```
VDPMDE      ( --- addr )
```

Executing one of the mode-setting words puts the corresponding number into **VDPMDE** as can be seen in the following:

```

GRAPHICS VDPMDE @ .
2 ok:0
```

## 6.2 fbForth 2.0 Graphics Words

Many **fbForth** 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 bitmapped (graphics2, split, split2) modes, respectively—‘T’ includes text80.

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

(T G M B)

For text modes, 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 ok:0**

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

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

base	<i>fg</i>	<i>bg</i>	<i>charset</i>	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 columns and rows are numbered from zero!!!*

For example,

base	<i>col</i>	<i>row</i>	<i>count</i>	<i>char</i>	<i>instr</i>	
<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)

will print a stream of 91 \*s, starting at column 10, row 17, that will wrap from right to left on the screen.

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

To print a vertical stream of characters, the word **VCHAR** is used in the same format as **HCHAR**. These characters will wrap from the bottom of the screen to the top.

**GCHAR** ( *col row --- char* )

The **fbForth** word **GCHAR** will return on the stack the ASCII code of the character currently at the specified position on the screen. If the above **HCHAR** instruction were executed and followed by

base	<i>col</i>	<i>row</i>	<i>instr</i>	
<b>HEX</b>	<b>F</b>	<b>11</b>	<b>GCHAR</b>	or
<b>DECIMAL</b>	<b>15</b>	<b>17</b>	<b>GCHAR</b>	

(T G)

**2Ah** or **42** would be left on the stack.

## 6.5 Defining New Characters

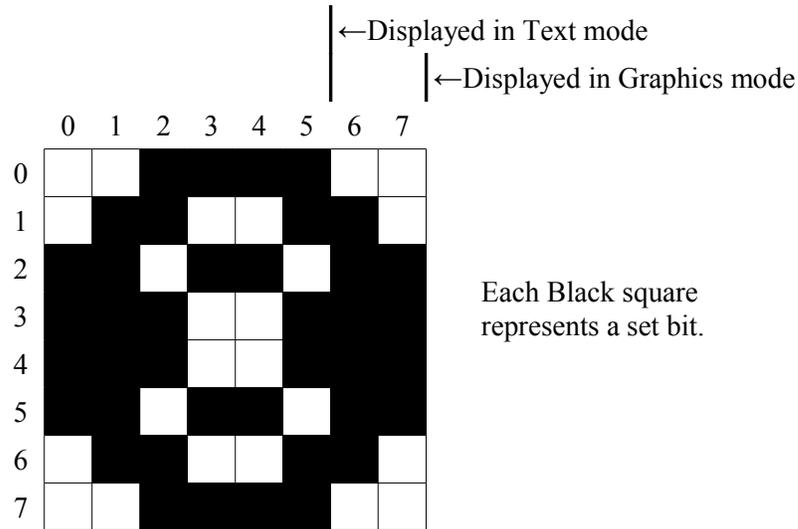
Each character in graphics mode is 8 x 8 pixels in size. Each row makes up one byte of the 8-byte character definition. Each set bit (1) takes on the foreground color while the others remain the background color.

In text mode, characters are defined in the same way, but only the left 6 bits of each row are displayed on the screen.

For example, these 8 bytes:

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

define this character:



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

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

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

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.
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 **fbForth** 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

**DELALL** ( --- )

**DELALL**

(G M B)

should be used to initialize all sprites. It removes all sprites from the screen and from memory. It also zeroes the Sprite Motion Table, except in bitmap modes. **DELALL** takes no parameters. Only the Sprite Descriptor Table will remain intact after this instruction is executed. The VDP mode must be set before using this word.

If the user wants to change the Sprite Descriptor Table to a location different from the default for a given VDP mode, use **SSDT** , which follows, before **DELALL** .

**SSDT** ( *vaddr* --- )

\*\*\*This word is optional in **fbForth 2.0**\*\*\*

As of **fbForth 2.0**, **SSDT** no longer initializes sprites. It merely changes the location of the Sprite Descriptor Table, which is now already set to a default location with the VDP mode changing words. If you *do* use this word, the computer must be set into the VDP mode you wish to use with sprites before executing it. Sprites should be initialized with **DELALL** after executing **SSDT** . Recall that *sprites are not available in text mode*.

With this word, 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 (except in bitmap modes) to avoid confusion. **2000h** is usually a good location, but any available 2KB (**800h**) boundary will do.

base	<i>vaddr</i>	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**.

### 6.6.3 Using Sprites in Bitmap Mode

**SATR** ( --- *vaddr* )

When using sprites in any of the bitmap modes (graphics2, split, split2) and after entering the desired VDP mode, the location of the Sprite Attribute List will have already been changed to **1B00h**. This can be verified in split or split2 mode as follows:

**HEX SATR U. 1B00 ok:0**

The base address of the Sprite Descriptor Table will also have been changed to the required **3800h**, which can be verified in split or split2 mode with

**HEX SPDTAB U. 3800 ok:0**

Only 59 character numbers will be available for sprite patterns in the bitmap modes because otherwise you will interfere with the disk buffering region at the top of VRAM. **SPCHAR** may

only be used to define patterns 0 – 58. (See the following section for information on **SPCHAR**.) If you really need more than 59 sprite patterns available and you don't need to open any files other than blocks files like FBLOCKS, you can change from 2 simultaneous files to 1 with **1 FILES** after changing the VDP mode because **fbForth 2.0** only opens one blocks file at a time, and then, only to read or write a single block. This will allow 65 more patterns (0 – 123).

*Note:* If you have mass storage in addition to diskettes (hard disk, nanoPEB, CF7+, *etc.*), it is possible that more than you expect of upper VRAM is used for buffering. In this case, check location **8370h** for the highest VRAM address available, subtract **3800h** from it, divide by 8 and truncate the quotient to get the number of sprite patterns available.

<b>3800h</b>	Sprite Patterns 0-58
<b>39DDh</b>	<b>01DEh</b>
<b>39DEh</b>	Start of Disk Buffer Region for 2 files

#### 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<sub>1</sub> n<sub>2</sub> n<sub>3</sub> n<sub>4</sub> 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.

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>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** (*dotcol dotrow 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	<i>dotcol</i>	<i>dotrow</i>	<i>color</i>	<i>char</i>	<i>spr</i>	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	<i>char</i>	<i>spr</i>	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	<i>color</i>	<i>spr</i>	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>	<i>instr</i>
-----	
<b>6</b>	<b>#MOTION</b>
(G M)	

will set sprites #0 – #5 in motion.

<i>n</i>	<i>instr</i>
-----	
<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 (*dotcol*) and vertical (*dotrow*) positions of sprite #2. 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 **fbForth** 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* )

	base	<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 upper left corners of the two sprites 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 **fbForth** 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>	<i>instr</i>
<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 takes 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 ---* )

$$\frac{spr \quad instr}{2 \quad \mathbf{DELSPR}}$$

(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 sets sprite #2 to sprite pattern #0 and sets the sprite off screen at  $x = 1$ ,  $y = 192$ . It zeroes the velocity of sprite #2 in the Sprite Motion Table, but does not alter the number of sprites the computer thinks are defined by virtue of not setting  $y = \mathbf{D0h}$ , the  $y$ -value that undefines all sprites with numbers greater than or equal to the lowest-numbered sprite with that value.

**DELALL** ( --- )

**DELALL**  
(G M B)

on the other hand, will remove all sprites from the screen and from memory. See § 6.6.2 Sprite Initialization above for more details.

## 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 takes 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>	<i>instr</i>	
<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 with **MCHAR** with the same position. In other words, cover the existing character.

## 6.8 Using Joysticks

**JOYST** (  $n_1$  --- [*char*  $n_2$   $n_3$ ] |  $n_2$  )

The **JOYST** instruction allows you to use joysticks in your **fbForth** program. **JOYST** accepts input from joystick #1 and the left side of the keyboard ( $n_1 = 1$ ) or from joystick #2 and the right side of the keyboard ( $n_1 = 2$ ). Return values depend on the value in **JMODE** (see below). If **JMODE** = 0 (default), **JOYST** executes **JKBD** (see below for more detail), which returns the character code *char* of the key pressed, the *x* status  $n_2$  and the *y* status  $n_3$ . If **JMODE**  $\neq$  0, **JOYST** executes **JCRU**, which checks only the joysticks and returns a single value with 0 or more of the 5 least significant bits set. See **JCRU** below for their meaning.

**JMODE** ( --- *addr* )

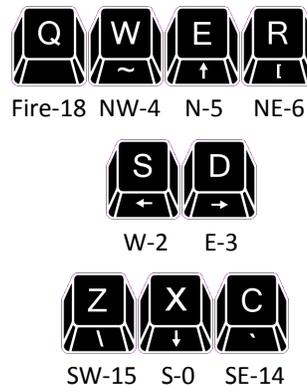
**JMODE** is a user variable that uses offset **26h** of the user variable table. It is used by **JOYST** to determine whether to execute **JKBD** (= 0) or **JCRU** ( $\neq$  0). The default value is 0. See **JOYST**, **JKBD** and **JCRU** in this section.

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

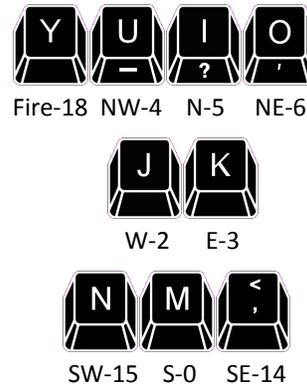
Executed by **JOYST** when **JMODE** = 0, **JKBD** allows input from joystick #1 and the left side of the keyboard ( $n_1 = 1$ ) or from joystick #2 and the right side of the keyboard ( $n_1 = 2$ ). Values returned are the character code *char* of the key pressed, the *x* status  $n_2$  and the *y* status  $n_3$ . 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 **JKBD** instruction (or **JOYST** with **JMODE** = 0) 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 on page 57.

**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 while using the keyboard, a character code of 18 (**12h**) along with two zeroes will be returned. If the fire button is pressed while using a joystick, a character code of 18 (**12h**) along with the current *x*- and *y*-joystick positions will be returned.

If you are using **JKBD** (or **JOYST** with **JMODE** = 0) in a loop, do not forget to **DROP** or otherwise use the three numbers left on the stack before calling **JKBD** or **JOYST** again. A stack overflow will likely result if you do not.

You will notice that the *x* and *y* values left by **JKBD** (or **JOYST** with **JMODE** = 0) for joystick status use **FCh** for left and down as described on page 250 of the *Editor/Assembler Manual*. If you are used to the value -4, which is the value returned for the same directions in TI Basic and TI Extended Basic, you can change **JKBD**’s return of **FCh** to -4 in block 39, where it is defined. You will need to change every instance of ‘**0FC**’ to ‘-4’ in the definition of **JKBD**—there are six of them.

The reason, of course, that **FCh** is used in **fbForth** (and TI Forth before it) is that **FCh** is how -4 is represented in a single byte in the byte-oriented GROM joystick table where it is stored.

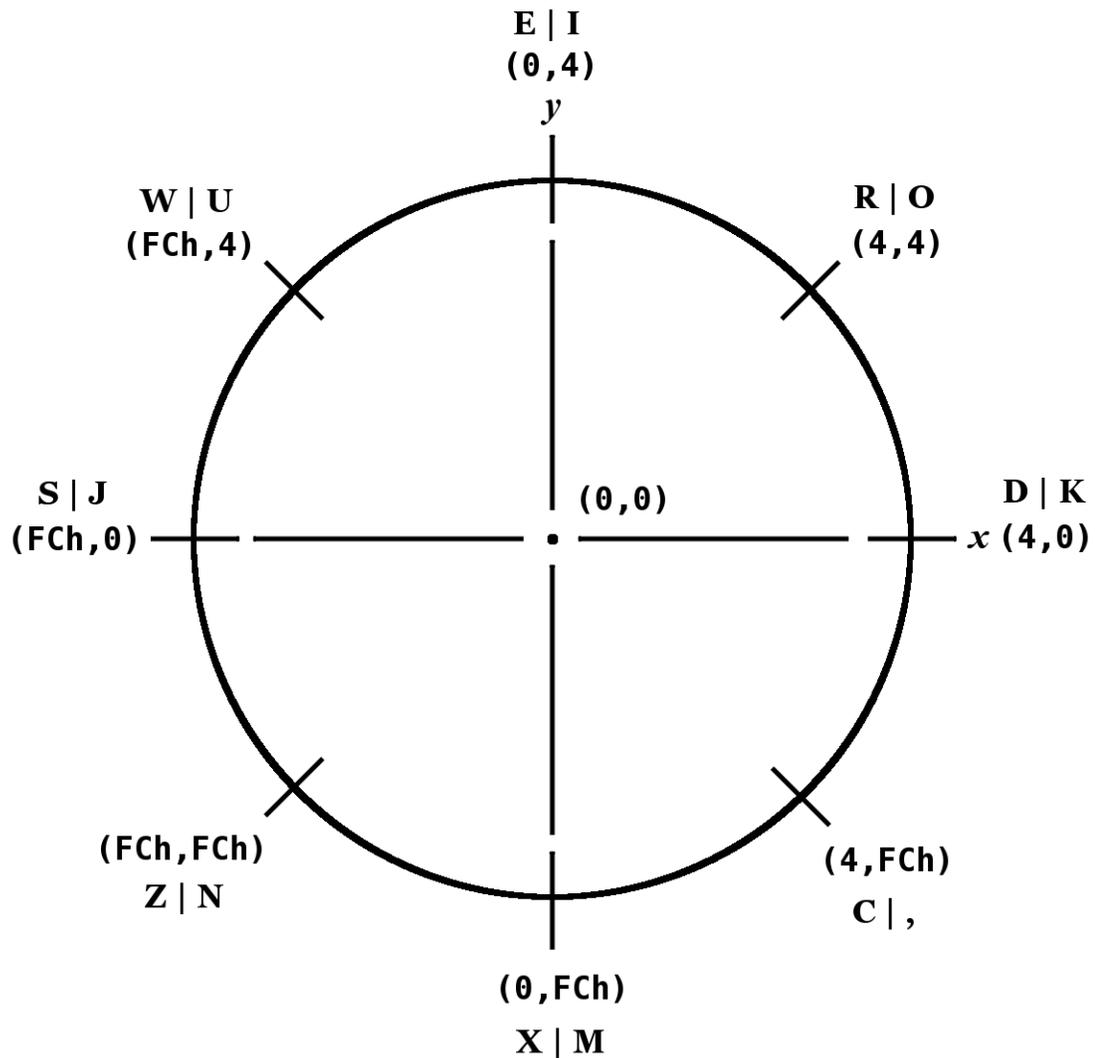
**JCRU** ( $n_1$  ---  $n_2$ )

Executed by **JOYST** when **JMODE**  $\neq$  0, **JCRU** allows input from joystick #1 ( $n_1 = 1$ ) or #2 ( $n_1 = 2$ ). The value  $n_2$  returned will have 0 or more of the 5 least significant bits set for direction and fire-button status. Bit values are 1 = Fire, 2 = W, 4 = E, 8 = S and 16 = N.

Two-bit directional combinations are 18 = NW (N + W or 16 + 2), 20 = NE, 10 = SW and 12 = SE.

If you are using **JCRU** (or **JOYST** with **JMODE**  $\neq$  0) in a loop, do not forget to **DROP** or otherwise use the number left on the stack before calling **JCRU** or **JOYST** again. A stack overflow will likely result if you do not.

*Note:* Be sure you have FBLOCKS dated 22DEC2013 or later before you attempt to use the words (**JOYST** , **JMODE** , **JKBD** and **JCRU** ) described in this section.



*Joystick positions and values left by JKBD (or JOYST with JMODE = 0)*

## 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** ( --- )

stores 0 in **DMODE** , which causes **DOT** to plot dots in the ‘on’ state.

**UNDRAW** ( --- )

stores 1 in **DMODE** , which causes **DOT** to plot dots in the ‘off’ state.

**DTOG** ( --- )

stores 2 in **DMODE** , which causes **DOT** to toggle dots between the ‘on’ and ‘off’ state. If the dot is ‘on’, **DOT** will turn it ‘off’ and vice versa.

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

The value of a variable called **DMODE** controls which drawing mode **DOT** is in. If **DMODE** contains 0, **DOT** is in **DRAW** mode. If **DMODE** contains 1, **DOT** is in **UNDRAW** mode, and if **DMODE** contains 2, **DOT** is 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 a dot at position (34,12), depending on the value of **DMODE** .

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

**DCOLOR** is short for “dot color” and should contain either one byte of foreground-background (FG-BG) color information or -1. The default is -1, which means that **DOT** will use the FG and BG colors of the byte in the Bitmap Color Table where the dot will be plotted/unplotted. These colors are black on transparent when the bitmap graphics modes are initialized. The screen color default is gray. To alter the FG and BG colors 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 FG color and the second specifies a BG color. Why do you need a BG color for a dot? There is a simple explanation: Each dot represents one bit of a byte in memory. Any ‘on’ bit in that byte displays the FG color while the others take on the BG color. Usually, you would specify the background color to be transparent so that all ‘off’ dots will have the screen’s color.

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

The **fbForth** instruction **LINE** allows you to easily plot a line between *any* two points on the bitmap 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 set **DMODE** and **DCOLOR** before using **LINE** if you do not want different plotting mode and FG-BG dot colors.

## 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. In **fbForth 2.0**, the values of **COLTAB**, **PDT**, **SATR** and **SPDTAB** are now changed by the mode changing words and do not require intervention by the user.

name	type	description	default	bitmap modes		
				4	5	6
<b>DMODE</b>	variable	Dot graphics drawing mode	<b>0</b>	<b>0   1   2</b>		
<b>SMTN</b>	constant	VDP address of Sprite Motion Table	<b>780h</b>	N/A	N/A	N/A
<b>COLTAB</b>	constant	VDP address of Color Table	<b>380h</b>	<b>0</b>	<b>0</b>	<b>0</b>
<b>PDT</b>	constant	VDP address of Pattern Descriptor Table	<b>800h</b>	<b>2000h</b>	<b>3000h</b>	<b>2000h</b>
<b>SATR</b>	constant	VDP address of Sprite Attribute Table	<b>300h</b>	<b>1B00h</b>	<b>1B00h</b>	<b>1B00h</b>
<b>SPDTAB</b>	constant	VDP address of Sprite Descriptor Table	<b>800h</b>	<b>3800h</b>	<b>3800h</b>	<b>3800h</b>

## 7 The Floating Point Support Package

Words introduced in this chapter:

>DEG	F!	F>	FOVER	LOG10
>F	F*	F>R	FP1	PI
>RAD	F+	F@	FP10	R>F
?FLERR	F-	FABS	FRAC	RAD/DEG
ATN	F->S	FCONSTANT	FROT	S->F
CEIL	F.	FDROP	FSWAP	SIN
COS	F/	FDUP	FVARIABLE	SQR
DEG/RAD	F0<	FFMT.	INT	TAN
EXP	F0=	FLERR	LN10INV	TRUNC
EXP10	F<	FLOOR	LOG	^
EULER_E	F=	FMINUS		

The floating point package is designed to make it easy to use the Radix 100 floating point package available in ROM in the **fbForth 2.0** cartridge. Normal use of these routines does not require the user to understand the implementation. You should consult Appendix L Notes on Radix-100 Notation to get a better understanding of how floating point numbers are managed on the TI-994A computer by the routines discussed in this chapter.

All floating point operations that have results that exceed the maximum or minimum representable floating point numbers convert the result to the maximum representable floating point number ( $\pm 9.999999999999999 \cdot 10^{127}$ ) for numbers that are too large and to 0 for numbers that are too small.

### 7.1 Floating Point Stack Manipulation

The floating point numbers in the TI-99/4A occupy 4 16-bit cells (8 bytes) each. In order to simplify stack manipulations with these numbers, the following stack manipulation words are presented. They have the same functions as their 16-bit, 1-cell counterparts that appear in this manual without the 'F' in their names:

<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>	$(f_1 f_2 \text{---} f_2 f_1)$
<b>FROT</b>	$(f_1 f_2 f_3 \text{---} f_2 f_3 f_1)$
<b>F&gt;R</b>	$(f \text{---} ) (R: \text{---} f)$
<b>R&gt;F</b>	$(\text{---} f) (R: f \text{---})$

## 7.2 Floating Point Defining Words

The following words create new floating point variables and constants. They both require an 8-byte floating-point number on the stack to place in the parameter field of the newly defined variable or constant:

<b>FVARIABLE</b>	<b>xxx</b>	( $f$ --- )	Create variable with initial value $f$
	<b>xxx</b>	( --- $addr$ )	Returns address when executed
<b>FCONSTANT</b>	<b>xxx</b>	( $f$ --- )	Create constant with value $f$
	<b>xxx</b>	( --- $f$ )	Returns $f$ when executed

## 7.3 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.*, define a floating point variable: **>F 0 FVARIABLE nnnn** could be used. **FVARIABLE** allots 8 bytes in the variable **nnnn**'s parameter field.)

## 7.4 Floating Point Conversion Words

The following words convert numbers on the stack to and from floating point numbers:

**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.

## 7.5 Floating Point Number Manipulation

**FABS** (  $f$  ---  $|f|$  )

converts  $f$  to its absolute value.

**FMINUS** (  $f$  ---  $-f$  )

negates  $f$  by negating the most significant word (topmost cell on the stack).

**FLOOR** (  $f_1$  ---  $f_2$  )

finds the closest integer  $f_2$  less than or equal to  $f_1$ .

**CEIL**            ( $f_1$  ---  $f_2$ )

finds the closest integer  $f_2$  greater than or equal to  $f_1$ .

**TRUNC**           ( $f_1$  ---  $f_2$ )

truncates  $f_1$ , leaving the integer portion  $f_2$  of  $f_1$  on the stack.

**FRAC**            ( $f_1$  ---  $f_2$ )

truncates  $f_1$ , leaving the fractional portion  $f_2$  of  $f_1$  on the stack.

## 7.6 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

```

## 7.7 Built-in Floating Point Constants

**DEG/RAD**        ( $f$ )

pushes the constant 57.295779513082 (degrees/radian) to the stack.

**EULER\_E**        ( $f$ )

pushes the constant  $e = 2.718281828459$  to the stack.

**FP1**              ( $f$ )

pushes the constant 1 to the stack as a floating point number. It is equivalent to **>F 1**.

**FP10**            ( $f$ )

pushes the constant 10 to the stack as a floating point number. It is equivalent to **>F 10**.

**LN10INV**        ( $f$ )

pushes the constant  $1/\ln(10) = 1/2.302\dots = 0.43429448190325$  to the stack.

**PI** ( ---  $f$  )

pushes the constant  $\pi = 3.141592653590$  to the stack.

**RAD/DEG** ( ---  $f$  )

pushes the constant 0.01745329251994 (radians/degree) to the stack.

## 7.8 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+** (  $f_1$   $f_2$  ---  $f_3$  )

puts on the stack the result ( $f_3$ ) of  $f_1 + f_2$ .

**F-** (  $f_1$   $f_2$  ---  $f_3$  )

puts on the stack the result ( $f_3$ ) of  $f_1 - f_2$ .

**F\*** (  $f_1$   $f_2$  ---  $f_3$  )

puts on the stack the result ( $f_3$ ) of  $f_1 \times f_2$ .

**F/** (  $f_1$   $f_2$  ---  $f_3$  )

puts on the stack the result ( $f_3$ ) of  $f_1 / f_2$ .

## 7.9 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.

**F0<** (  $f$  ---  $flag$  )  $flag$  is true if  $f$  on stack is negative

**F0=** (  $f$  ---  $flag$  )  $flag$  is true if  $f$  on stack is zero

**F>** (  $f_1$   $f_2$  ---  $flag$  )  $flag$  is true if  $f_1 > f_2$

**F=** (  $f_1$   $f_2$  ---  $flag$  )  $flag$  is true if  $f_1 = f_2$

**F<** (  $f_1$   $f_2$  ---  $flag$  )  $flag$  is true if  $f_1 < f_2$

### 7.10 Formatting and Printing Floating Point Numbers

**F.** (  $f$  --- )

The word **F.** is used to print the floating point number on top of the stack to the terminal in TI Basic free format. **F.** is the simplest printing word for floating point numbers. It is exactly the same as **0 FFMT.** (see next definition) and is the only floating point print word retained from TI Forth and **fbForth 1.0** because of its likely common use and to maintain some backward compatibility:

- 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.<sup>15</sup>

The following screen shows examples of all the above situations:

```

>F 8 ok:4
F. 8 ok:0
>F 123.45690123456 ok:4
F. 123.4569012 ok:0
>F -123.456789 ok:4
F. -123.456789 ok:0
>F 12.34567E89 ok:4
F. 1.23457E+90 ok:0
>F 12.3456789E-56 ok:4
F. 1.23457E-55 ok:0
>F 12.3456789E120 ok:4
F. 1.23457E+*** ok:0
>F 12.3456789E127 ok:4
F. 9.99999E+*** ok:0

```

The words **F.R**, **FF.** and **FF.R** are no longer defined in **fbForth 2.0**, but can be defined in terms of **FFMT.** below.

**FFMT.** (*f* [*intLen fracLen*] *optMask* ---)

**FFMT.** handles all of the formatting and printing of floating point numbers. The integer length *intLen* and the fraction length *fracLen* are required if and only if the options mask *optMask*  $\neq 0$ . To avoid confusion, you should always use **F.** (*optMask* = 0) for free format output and limit use of **FFMT.** to fixed-format output (*optMask*  $\neq 0$ ). Then you may think of this word as *requiring* all four stack entries in the above stack signature.

The output field width consists of *intLen* + *fracLen* (the significand field width) plus the exponent field width (always 4 characters for E-notation or 5 characters for extended E-notation). The significand field width cannot exceed 16 characters or an error message that the field is too big will be printed. If the number cannot be formatted for the requested output field width and  $intLen + fracLen \leq 16$ , the field will be filled with asterisks.

The following table details the various formats, free and fixed, that are possible and the input stack entry parameters required for **FFMT.** :

<sup>15</sup> The exponential format of the output string provided by TI in its GPL routine allows for just two digits for the power of ten. It is puzzling that TI did this because the exponent can be as high as 127 and as low as -128. This means that perfectly legitimate three-digit exponents appear as "\*\*\*" in the output! This was one of the reasons for the Author's adaptation of the Geneve MDOS L10 Floating Point Math Library (with permission from Beery Miller of 9640 News) to run on the TI-99/4A in cartridge ROM space.

Parameter	Description
$f$	Floating point number to be formatted
$intLen$	Number of places before decimal point, including sign
$fracLen$	Number of places after decimal point, including decimal point
$optMask$	Output options mask—
bit 0:	0 = Free form TI Basic style. No other bits should be set. There should be <b>no</b> other numbers on the stack except $f$ and $optMask = 0$ .
	1 = Fixed format, requiring four parameters on the stack, $f$ , $intLen$ , $fracLen$ and $optMask$
bit 1:	2 = Explicit sign
bit 2:	4 = Show '+' for positive number instead of space. Bit 1 must also be set.
bit 3:	8 = E-notation. There will be 4 additional places in the output not accounted for by $intLen$ and $fracLen$ .
bit 4:	16 = Extended E-notation. Bit 3 must also be set. There will be 5 additional places in the output not accounted for by $intLen$ and $fracLen$ .

Several examples of the output possible with **FFMT** . are shown in the following screen:

```

>F 123 ok:4
2 15 1 FFMT. field too big! ok:0
>F 123 ok:4
2 14 1 FFMT. ***** ok:0
>F 123 ok:4
4 12 1 FFMT. 123.000000000000 ok:0
>F 123.45678901234 ok:4
2 14 9 FFMT. 1.2345678901230E+02 ok:0
>F -1.2345678901234E120 FDUP ok:8
2 14 9 FFMT. -1.2345678901230E+** ok:4
2 14 25 FFMT. -1.2345678901230E+120 ok:0

```

## 7.11 Transcendental Functions

The following transcendental functions are also available:

**>DEG** ( $f_1 \text{ --- } f_2$ ) is the conversion of  $f_1$  radians to  $f_2$  degrees

<b>&gt;RAD</b>	$(f_1 \text{ --- } f_2)$	is the conversion of $f_1$ degrees to $f_2$ radians
<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>EXP10</b>	$(f_1 \text{ --- } f_2)$	$f_2$ is 10 raised to the $f_1$ power
<b>LOG</b>	$(f_1 \text{ --- } f_2)$	$f_2$ is the natural log of $f_1$
<b>LOG10</b>	$(f_1 \text{ --- } f_2)$	$f_2$ is the common log ( $\log_{10}$ ) 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 $f_1$

## 7.12 Interface to the Floating Point Routines

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). Though **fbForth 2.0** uses them for floating point calculations, the following words are no longer defined in **fbForth 2.0**:

<b>&gt;ARG</b>	<b>FAC</b>	<b>FAC&gt;ARG</b>	<b>FMUL</b>	<b>SETFL</b>
<b>&gt;FAC</b>	<b>FAC-&gt;S</b>	<b>FADD</b>	<b>FSUB</b>	<b>VAL</b>
<b>ARG</b>	<b>FAC&gt;</b>	<b>FDIV</b>	<b>S-&gt;FAC</b>	

## 7.13 Handling Floating Point Errors

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

**FLERR** is used to fetch the contents of the floating point error register (**8354h**) to the stack. It can be used to get more specific information about the error than you get with **?FLERR** below. See the next section for error codes. The *Editor/Assembler Manual* may also be helpful because, even though the console routines it describes are no longer used in **fbForth 2.0**, they were the basis of the current routines and still use location **8354h** for storing the error.

**?FLERR** ( $\text{---}$ )

**?FLERR** issues the following error message if the last floating point operation resulted in an error:

**?FLERR ? floating point error**

*Note:* All of the floating point operations in **fbForth 2.0** reset the floating point error location, **8354h**, before they run. You no longer need to insure it is reset yourself as with the GPL routines. You do, however, need to check for an error before another floating point operation clears it. Also, the message is meaningless if no floating point operation has yet occurred.

### 7.14 Floating Point Error Codes

The following table lists the possible error codes reported in the byte at location **8354h** after floating-point operations:

Code	Error Description
01	Overflow
02	Syntax
03	Integer overflow on conversion
04	Square root of a negative number
05	Negative number to non-integer power
06	Logarithm of a negative number
07	Invalid argument in a trigonometric function

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

Words introduced in this chapter:

APPND	INPT	PABS	SQNTL
CLSE	INTRNL	RD	STAT
DIR	LD	REC-LEN	SV
DLT	OPN	REC-NO	SWCH
DSPLY	OUTPT	RLTV	UNSWCH
F-D"	PAB-ADDR	RSTR	UPDT
FILE	PAB-BUF	SCRTCH <sup>16</sup>	VRBL
FXD	PAB-VBUF	SET-PAB	WRT

This chapter will explain the means by which different types of data files native to the TI-99/4A are accessed with **fbForth 2.0**. To further illustrate the material, two commented examples have been included in this chapter. The first (§ 8.7 ) demonstrates the use of a relative disk file and the second (§ 8.8 ) a sequential RS232 file.

A group of Forth words has been included in the resident dictionary of **fbForth 2.0** to permit a Forth program to reference common data with Basic or Assembly Language programs. These words implement the file system described in the *User's Reference Guide* and the *Editor/Assembler Manual*. Note that the **fbForth 2.0** system (as opposed to TI Forth) uses only normally formatted disks for the system blocks file (FBLOCKS) and that you may perform file I/O to/from any disks, including the system disks, as long as they are properly initialized by a Disk Manager and there is enough room. You should avoid writing to TI Forth disks that contain TI Forth blocks (screens) because you will likely destroy them.

### 8.1 Switching VDP Modes After File Setup

You must be careful switching VDP modes after you set up access to a file (discussed in following sections) because switching to/from bitmap and 80-column text modes moves the PAB and file-setup areas (PABS) in VRAM. *This will destroy access to the file!* You can, however, switch safely among graphics, text and multicolor modes without losing access to your file information.

### 8.2 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.

<sup>16</sup> **SCRTCH** , is *not* part of **fbForth**. It is mentioned because it was defined in TI Forth. TI, however, never implemented **SCRTCH** in any DSR for the TI-99/4A. Its use always resulted in a file I/O error.

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. Adequate space is provided for many PABs in this area. More information on the details of a PAB are available in 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 (Status)	Byte 9 Name Length
Byte 10+ File Descriptor • • •	

### 8.3 File Setup and I/O Variables

All Device Service Routines (DSRs) on the TI-99/4A expect to perform data transfers to/from VDP RAM. Since **fbForth 2.0** 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** ( --- *addr* )

Holds address in VDP RAM of first byte of the PAB.

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

Holds address in CPU RAM of first byte in **fbForth**'s memory where allocation has been made for this buffer.

**PAB-VBUF** ( --- *addr* )

Holds address in VDP RAM of the first byte of a region of adequate length to store data temporarily while it is transferred between the file and **fbForth**. 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 bitmap mode, be cautious where **PAB-VBUF** is placed.

There is practically no available space in bitmap mode. There are a couple of things you can do. You can set simultaneous files to 1 with **1 FILES** to free up 518 bytes between the old value in **8370h** and the new value put there after executing **1 FILES**. This should be safe as long as you do not read/write blocks because **fbForth** only opens a file to read/write one block. The blocks file is closed the rest of the time.

The other thing you can do is to temporarily use the bitmap color and/or screen image tables by saving and restoring the area you want to use. It might even be rather entertaining to watch your file I/O happen on the screen!

**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 RAM buffer)
<b>PABS @ 10 +</b>	(PAB starts 10 bytes into VRAM region for PABS and this address will be stored in <b>PAB-ADDR</b> )
<b>MY-BUF</b>	(RAM address to be stored in <b>PAB-BUF</b> )
<b>6000</b>	(A free area at <b>1770h</b> in VRAM to be stored in <b>PAB-VBUF</b> )
<b>FILE JOE</b>	(Whenever the word <b>JOE</b> is executed, the file I/O variables, <b>PAB-ADDR</b> , <b>PAB-BUF</b> and <b>PAB-VBUF</b> , will be set as defined here.)
<b>JOE</b>	(Use the file's identifying word (FID) 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 the first ten bytes. It then places the contents of the variable **PAB-VBUF** into its PAB location at bytes 2 and 3. Obviously, **PAB-ADDR** and **PAB-VBUF** must be set up before **SET-PAB** is invoked, which is done by executing the file identifying word ( **JOE** , in the above example) before **SET-PAB**. **SET-PAB** should be executed only once for each file and should immediately follow the first invocation of the file ID word.

## 8.4 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
	TI Basic	fbForth	
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
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 <i>and</i> read from
	APPEND	<b>APPND</b>	Data may be added to the end of the 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" ok:0**

**F-D" DSK2.FILE-ABC" ok:0**

## 8.5 Words that Perform File I/O

The actual I/O operations are performed by the following words. The table gives the usual TI Basic keyword associated with the corresponding **fbForth** word. Here, as in the previous table,

the **fbForth** words are spelled differently than the TI Basic words to avoid conflict with one or more existing **fbForth** words.

From TI Basic	From <b>fbForth</b>	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
DELETE	<b>DLT</b>	7
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** and **WRT** instructions with a relative file, you must place the desired, zero-based 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** via your **PAB-VBUF** 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 your **PAB-BUF** via your **PAB-VBUF** 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.

**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** .

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

returns the status byte *b* (PAB+8, labeled “Screen Offset” in the PAB diagram above) of the current device/file from the PAB pointed to by **PAB-ADDR** after calling the DSR’s STATUS opcode (9), which actually gets the status and writes it to PAB+8. Incidentally, the term “Screen Offset” for PAB+8 is from its use by the cassette interface, which must put prompts on the screen, to get the offset of screen characters with respect to their normal ASCII values. 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.

Almost all of the file I/O support words of TI Forth and **fbForth 1.0** are no longer available in **fbForth 2.0** as high-level Forth definitions. They have been defined in TMS9900 Assembly Language as part of the resident dictionary and are no longer directly executable. If the user desires more information from a file I/O error condition than the fact that a file I/O error occurred, the following definition can be used to retrieve the flag/status byte from the file’s PAB:

```
: GET-FLAG PAB-ADDR @ 1+ VSBR ;
```

The following describes in detail the information that can be obtained with **GET-FLAG** :

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

retrieves to the stack the flag/status byte *b* from byte 1 the current PAB. The high-order 3 bits are used for DSR error return, except for “bad device name”. With the “bad device name” error, this error return will be 0; but, the GPL status byte (**837Ch**) will have the COND bit set (**20h**). The low-order 5 bits are set by routines that set the file type prior to

calling **OPN** , which reads these bits. See table below for the meaning of each bit of the flag/status byte:

### 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>

Examples of file I/O in use are shown in § 8.7 , § 8.8 and block 19ff in FBLOCKS (dated 01SEP2014 or later, which has definitions of the alternate I/O capabilities for printing to the RS232 interface).

## 8.6 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 serial printer. For example, the **LIST** instruction normally outputs to the monitor. By typing

**SWCH 45 LIST UNSWCH**

you can list block 45 of the current blocks file to the printer. If your RS232 printer is not on port 1 and set at 9600 baud or you would rather print via the parallel port, you must modify the word **SWCH** in block 19 of FBLOCKS.

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. Otherwise, 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.

### 8.7 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>	<b>PAB</b> 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 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 in the default (UPDATE) mode. 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 in a Forth block beforehand and loaded at this time.

```

: FIL-WRT TESTDATA      TESTDATA is assumed to be the beginning memory address
                          of the information to be written to the file
  10 0 DO                Want to write 16 (10h) 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 from block 16 of FBLOCKS)
<b>CLSE</b>	Close the file

### 8.8 File I/O Example 2: Sequential RS232 File

Instruction	Comment
<b>HEX</b>	Change number base to hexadecimal
<b>0 VARIABLE MY-BUF 4E ALLOT</b>	Create an 80-character <b>PAB-BUF</b>
<b>PABS @ 30 +</b>	Skip all previous PABs. This will be the <b>PAB-ADDR</b>
<b>MY-BUF 1900</b>	Place the <b>PAB-BUF</b> and <b>PAB-VBUF</b> on stack in preparation for <b>FILE</b>
<b>FILE PRNTR</b>	Associates the name <b>PRNTR</b> with these three parameters
<b>PRNTR</b>	File name must be executed before using any other File I/O words
<b>SET-PAB</b>	Create a PAB skeleton
<b>DSPLY</b>	<b>PRNTR</b> will contain printable information
<b>SQNTL</b>	<b>PRNTR</b> may be accessed only in sequential order
<b>VRBL</b>	Records may have variable lengths
<b>50 REC-LEN</b>	Maximum record length is 80 char.
<b>F-D" RS232.BA=9600" or</b>	<b>PRNTR</b> will be an RS232 serial "file" with baud rate = 9600 or a parallel printer "file".
<b>F-D" PIO"</b>	
<b>OPN</b>	Open the file
. . . . .	

A procedure is necessary to write more than one record to a file. A file-write routine may be composed in a Forth block beforehand and loaded at this time. The following is a simple example:

<b>: PRNT FILE-INFO</b>	<b>FILE-INFO</b> is assumed to be the beginning memory address of the information to be sent to the printer
<b>20 0 DO</b>	Will write 32 records

<b>DUP</b>	Duplicate address
<b>MYBUF 50 CMOVE</b>	Move 80 characters from <b>FILE-INFO</b> to <b>MY-BUF</b>
<b>50 WRT</b>	Write one record to printer
<b>50 +</b>	Increment address on stack
<b>LOOP DROP</b>	Clear stack
<b>;</b>	End definition
. . . . .	
<b>PRNT</b>	Execute write program
<b>CLSE</b>	Close the file called <b>PRNTR</b>

## 8.9 Disk Catalog Utilities

Two different disk cataloging utilities are available in FBLOCKS dated 19JUN2015 and later.

### 8.9.1 DIR

**DIR** is adapted, with permission, from Mark Wills' TurboForth and, though available in FBLOCKS dated as early as 17OCT2014, only the version that first appears in the 19JUN2015 edition should be used:

**DIR** ( --- )

**DIR** catalogs to the output device the disk device name that follows it in the input stream. The disk device name must be terminated with a period. **DIR** gets its information from the DSR's catalog "file" (see Chapter 8 "Catalog File Access" in TI's *Software Specifications for the 99/4 Disk Peripheral*). **DIR** will not load if **CAT** (below) is loaded. Use **MENU** to show what block to load for **DIR**.

Usage: **DIR DSK1.**

Example:

```

DIR DSK1.
Disk Name: FBFORTH2
Total: 1438 Free: 768 Used: 670
Name      Size Type   B/R P
-----
CHAR01    0  PROGRAM
CHAR01FBF 0  PROGRAM
CHAR01FBF 0  PROGRAM
CHAR01FBF 0  PROGRAM
CHAR01TIW 5  PROGRAM
FBLOCKS   321 DIS/FIX 128
STUFF     321 DIS/FIX 128
-----
6 files   670 sectors
ok:0

```

### 8.9.2 CAT

**CAT** (*n* ---)

**CAT** catalogs to the output device the disk number *n* on the stack for the current DSR.

**CAT** reads the Volume Information Block (VIB) to get the disk name, total sectors and free sectors. The free sectors are calculated by adding all the zero bits in the allocation bitmap found in the VIB.

Next, **CAT** reads the File Descriptor Index Record (FDIR) and finds each file's File Descriptor Record (FDR) from the sector pointers in the FDIR.

Each FDR has the file's name, file type, sectors occupied by the file, protection and EOF byte offset in the last sector. The EOF offset and the sector count are used to calculate the actual size in bytes of a PROGRAM file. The sector size that **CAT** displays for a file is one more than the size of the file body to account for the file's FDR. See Appendix K, "Diskette Format Details" for more specific information about the VIB, FDIR and FDR.

**CAT** will not load if **DIR** is loaded.

Usage: **2 CAT** to catalog DSK2.

Example:

```

2 CAT
Disk Name: FAIRE2014A
Total: 1438 Free: 764 Used: 674
Name      Size Typ B/R Bytes P
-----
BCAT      2   PGM      132
BCAT2     2   PGM      132 Y
CHAR01    5   PGM     2048
CHAR01FBF 5   PGM     2048
CHAR01FBF 5   PGM     1024
CHAR01IW  5   PGM     1024
FBLOCKS   321 D/F 128
STUFF     321 D/F 128
-----
8 files   674 sectors
ok:0

```

## 9 The fbForth 2.0 TMS9900 Assembler

The assembler supplied with your **fbForth 2.0** system is typical of assemblers supplied with fig-Forth systems and is almost identical with the TI Forth assembler—there are some enhancements. 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 **fbForth 2.0** language is available to the user to assist in macro type assembly, if desired. The assembler uses the standard Forth convention of Reverse Polish or Postfix Notation for each instruction. For example, the instruction to add register 1 to register 2 is:

```
R1 R2 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 **fbForth 2.0** language. A complete list of Forth-style instruction mnemonics is given in the next section.

### 9.1 TMS9900 Assembly Mnemonics

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

These words are available when the assembler is loaded. Only the words **C,** and **R0** (see later) conflict with the existing **fbForth 2.0** vocabulary.

Most assembly code in **fbForth 2.0** will probably use **fbForth 2.0**'s workspace registers. The following table describes the register allocation. The user may use registers R0 through R7 for any purpose. They are used as temporary registers only within **fbForth** words which are themselves written in TMS9900 assembly code.

## 9.2 fbForth 2.0's Workspace Registers

Register Name		Usage
Original	Alternate	
0	R0	These registers are available. They are used only within <b>fbForth</b> words written in <b>CODE</b> .
1	R1	
2	R2	
3	R3	
4	R4	
5	R5	
6	R6	
7	R7	
UP	R8	Points to base of User Variable area
SP	R9	Parameter Stack Pointer
W	R10	Inner Interpreter current Word pointer
11	R11	Linkage for subroutines in <b>CODE</b> routines
12	R12	Used for CRU instructions
IP	R13	Interpretive Pointer
RP	R14	Return Stack Pointer
NEXT	R15	Points to the next instruction fetch routine

## 9.3 Loading and Using the Assembler

The **fbForth 2.0** TMS9900 Assembler is located in blocks 21 – 26 of FBLOCKS and is loaded by typing **21 LOAD** . The words **CODE** and **;CODE** and their synonyms, **ASM:** and **DOES>ASM:** are in the resident dictionary and part of the Forth vocabulary. When the assembler is loaded, it is loaded into the Assembler vocabulary. To use the assembler, it must be the context vocabulary, which may be effected by typing **ASSEMBLER** or by using the words **CODE** , **ASM:** , **;CODE** or **DOES>ASM:** , each of which makes Assembler the context vocabulary.

There are only two words in the Assembler vocabulary that are part of the resident dictionary, namely, **NEXT** , and its synonym, **;ASM** . After defining words that use **CODE** , **ASM:** , **;CODE** or **DOES>ASM:** , it is advisable to execute **FORTH** to restore the context vocabulary to Forth, unless such use is immediately followed by **:** (beginning a colon definition), which restores the context vocabulary to the current vocabulary (usually Forth). The important point is that Forth must be the context vocabulary before the Forth words **C** , and **R0** can be executed because **C** , and **R0** are

the only Assembler vocabulary words that conflict with Forth vocabulary words of the same name.

From this point on in this chapter except for the first two examples that show both versions, we will use the synonyms for **CODE** , **;CODE** and **NEXT** , because they are easier to understand, at least that is the author's opinion. The respective synonyms are **ASM:** , **DOES>ASM:** and **;ASM** . Please keep these in mind when attempting to compare **fbForth** code using them with TI Forth code.

Assembly definitions either begin with **ASM:** or end with **DOES>ASM:** . Each are followed by assembly mnemonics or the machine-code equivalent. **ASM:** is used in the following way:

```
ASM: EXAMPLE <assembly mnemonics> ;ASM
```

which is the same as

```
CODE EXAMPLE <assembly mnemonics> NEXT,
```

This defines a Forth word named **EXAMPLE** with an execution procedure defined by the assembly mnemonics that follow **EXAMPLE** , which usually terminate with **;ASM** . The assembly code should end with **;ASM** so the **fbForth 2.0** inner interpreter can get to the next word to be executed. There are several examples using **ASM:** in the sections that follow.

**DOES>ASM:** is used with **<BUILDS** to create the execution procedure of a new defining word very much like the word **DOES>** except that **DOES>ASM:** does not cause the PFA of newly defined words to be left on the stack for the consumption of the code following **DOES>ASM:** as is the case with **DOES>** . **DOES>ASM:** is used as follows:

```
: DEF-WRD <BUILDS ... DOES>ASM: <assembly mnemonics> ;ASM
```

which is the same as

```
: DEF-WRD <BUILDS ... ;CODE <assembly mnemonics> NEXT,
```

Just as with **ASM:** , assembly code following **DOES>ASM:** should end with **;ASM** . 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 **DOES>ASM:** . An example using **DOES>ASM:** is shown in § 9.9 .

## 9.4 fbForth 2.0 Assembler Addressing Modes

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 the **fbForth 2.0** assembler code (column 1) for various instructions, the TI Forth code (column 2) and the conventional Assembler (column 3) method of coding the same instructions. The Wycove Forth equivalents of the **fbForth 2.0** addressing mode words may also be used. The TI Forth code can be used in **fbForth 2.0** with no changes.

The word **;ASM** is defined as a synonym for **NEXT** , (see § 9.4.6 for definition of **\*NEXT** ). The high-level **fbForth 2.0** code for both words is

```
: NEXT, *NEXT B, ;
```

```
: ;ASM NEXT, ;
```

and is equivalent to the following assembly code:

```
B *R15
```

### 9.4.1 Workspace Register Addressing

The registers in the **fbForth 2.0** code below can be referenced directly by number; however, we are using the alternate, easier to read, R designation:

<b>fbForth 2.0</b>	<b>TI Forth</b>	<b>Conventional Assembler</b>		
HEX	HEX			
ASM: EX1	CODE EX1		DEF	EX1
R1 R2 A,	1 2 A,	EX1	A	R1,R2
R3 INC,	3 INC,		INC	R3
R3 FFFC ANDI,	3 FFFC ANDI,		ANDI	R3,>FFFC
;ASM	NEXT,		B	*R15

### 9.4.2 Symbolic Memory Addressing

Symbolic addressing is done with the @() word (Wycove Forth equivalent: @@). It is used after the address.

<b>fbForth 2.0</b>	<b>TI Forth</b>	<b>Conventional Assembler</b>		
0 VARIABLE VAR1	0 VARIABLE VAR1	VAR1	BSS	2
5 VARIABLE VAR2	5 VARIABLE VAR2	VAR2	DATA	5
ASM: EX2	CODE EX2		DEF	EX2
VAR2 @() R1 MOV,	VAR2 @() 1 MOV,	EX2	MOV	@VAR2,R1
R1 2 SRC,	1 2 SRC,		SRC	R1,2
R1 VAR1 @() S,	1 VAR1 @() S,		S	R1,@VAR1
VAR2 @() VAR1 @() SOC,	VAR2 @() VAR1 @() SOC,		SOC	@VAR2,@VAR1
;ASM	NEXT,		B	*R15

### 9.4.3 Workspace Register Indirect Addressing

Workspace Register Indirect addressing is done with the \*? word (Wycove Forth equivalent: \*\*). It is used after the register number to which it pertains. In line 4 below we use the clearer definition of § 9.4.6 for **fbForth 2.0**. TI Forth must use \*? .

<b>fbForth 2.0</b>	<b>TI Forth</b>	<b>Conventional Assembler</b>		
HEX 2000 CONSTANT XRAM	HEX 2000 CONSTANT XRAM	XRAM	EQU	>2000
ASM: EX3	CODE EX3		DEF	EX3
R1 XRAM LI,	1 XRAM LI,	EX3	LI	R1,XRAM
*R1 R2 MOV,	1 *? 2 MOV,		MOV	*R1,R2
;ASM	NEXT,		B	*R15

### 9.4.4 Workspace Register Indirect Auto-increment Addressing

Workspace Register Indirect Auto-increment addressing is done with the **\*?+** word (Wycove Forth equivalent: **\*+**). It is also used after the register to which it pertains. In line 4 below we use the clearer definition of § 9.4.6 for **fbForth 2.0**. TI Forth must use **\*?+**.

<b>fbForth 2.0</b>	<b>TI Forth</b>	<b>Conventional Assembler</b>
HEX 2000 CONSTANT XRAM	HEX 2000 CONSTANT XRAM	XRAM EQU >2000
ASM: EX4	CODE EX4	DEF EX4
R1 XRAM LI,	1 XRAM LI,	EX4 LI R1,XRAM
*R1+ R2 MOV,	1 *?+ 2 MOV,	MOV *R1+,R2
;ASM	NEXT,	B *R15

### 9.4.5 Indexed Memory Addressing

The final addressing type is Indexed Memory addressing. This is performed with the **@(?)** word (Wycove Forth equivalent: **()**) used after the Index and register as shown below. Here we use the clearer definition of § 9.4.6 for **fbForth 2.0**. TI Forth must use **@(?)**.

<b>fbForth 2.0</b>	<b>TI Forth</b>	<b>Conventional Assembler</b>
HEX 2000 CONSTANT XRAM	HEX 2000 CONSTANT XRAM	XRAM EQU >2000
ASM: EX5	CODE EX5	DEF EX5
XRAM @(R1) R2 MOV,	XRAM 1 @(?) 2 MOV,	EX5 MOV @XRAM(R1),R2
DECIMAL	DECIMAL	
XRAM 22 + @(R2)	XRAM 22 + 2 @(?)	MOV @XRAM+22(R2),@XRAM+26(R2)
XRAM 26 + @(R2) MOV,	XRAM 26 + 2 @(?) MOV,	
;ASM	NEXT,	B *R15

### 9.4.6 Addressing Mode Words for Special Registers

In order to make addressing modes easier for the **W**, **RP**, **IP**, **SP**, **UP** and **NEXT** as well as all the numbered registers (**R0** – **R15**), the following words are available and eliminate the need to enter the register name separately. The register number (0 – 15) in the last entry is represented by **n**:

<b>Register Address</b>	<b>Indirect</b>	<b>Indexed</b>	<b>Indirect Auto-increment</b>
<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>
<b>Rn</b>	<b>*Rn</b>	<b>@(Rn)</b>	<b>*Rn+</b>

## 9.5 Handling the fbForth 2.0 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 **fbForth 2.0** 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+ R0 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,  
R1 *SP MOV,
```

The same procedures obtain for the return stack using **\*RP+**, **RP** and **\*RP**; but, if you must manipulate it, be very careful that you restore the return stack when you are finished and before the system needs it.

## 9.6 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 **fbForth 2.0** 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, ... THEN, [ also IF, ... ENDIF, ]  
IF, ... ELSE, ... THEN, [ also IF, ... ELSE, ... ENDIF, ]  
BEGIN, ... UNTIL,  
BEGIN, ... AGAIN,  
BEGIN, ... WHILE, ... REPEAT,
```

Note that **THEN,** is a synonym for TI Forth's **ENDIF,**. **THEN,** is used in the **fbForth 2.0** Assembler example below; but, the **ENDIF,** of the TI Forth example works, as well. Be sure you have FBLOCKS dated 01SEP2014 or later before you attempt to use **THEN,**.

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.

## 9.7 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>

## 9.8 Assembly Example for Structured Constructs

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

<b>fbForth 2.0</b>	<b>TI Forth</b>	<b>Conventional Assembler</b>	
<b>( GENERALIZED SHIFTER )</b>	<b>( GENERALIZED SHIFTER )</b>	<b>* GENERALIZED SHIFTER</b>	
<b>ASM: SHIFT</b>	<b>CODE SHIFT</b>	<b>DEF</b>	<b>SHIFT</b>
<b>*SP+ R0 MOV,</b>	<b>*SP+ 0 MOV,</b>	<b>SHIFT</b>	<b>MOV *SP+,R0</b>
<b>NE IF,</b>	<b>NE IF,</b>	<b>JEQ</b>	<b>L3</b>
<b>*SP R1 MOV,</b>	<b>*SP 1 MOV,</b>	<b>MOV</b>	<b>*SP,R1</b>
<b>R0 ABS,</b>	<b>0 ABS,</b>	<b>ABS</b>	<b>R0</b>
<b>GTE IF,</b>	<b>GTE IF,</b>	<b>JLT</b>	<b>L1</b>
<b>R1 R0 SLA,</b>	<b>1 0 SLA,</b>	<b>SLA</b>	<b>R1,0</b>
<b>ELSE,</b>	<b>ELSE,</b>	<b>JMP</b>	<b>L2</b>
<b>R1 R0 SRL,</b>	<b>1 0 SRL,</b>	<b>L1</b>	<b>SRL R1,0</b>
<b>THEN,</b>	<b>ENDIF,</b>		
<b>R1 *SP MOV,</b>	<b>1 *SP MOV,</b>	<b>L2</b>	<b>MOV R1,*SP</b>
<b>THEN,</b>	<b>ENDIF,</b>		
<b>;ASM</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.

### 9.9 Assembly Example with **DOES>ASM:**

Before giving an example of defining an **fbForth 2.0** defining word with **DOES>ASM:**, an explanation of why you might want to use it in the first place is in order.

The defining words that are part of the **fbForth 2.0** kernel are **:** (paired with **;**), **VARIABLE**, **CONSTANT**, **USER**, **VOCABULARY**, **<BUILDS** (paired with **DOES>** or **DOES>ASM:**) and **CREATE**. The defining words **ASM:** and **DOES>ASM:**, as well as **;ASM**, are all part of the resident dictionary. Of course, most words you would ever need to define can be created with the first three (**:**, **VARIABLE** and **CONSTANT**). However, you too can use **<BUILDS** and **CREATE**, the same words used for defining most of the above, for the eventuality that these do not suffice.

In **fbForth 2.0**, it is not useful to use **CREATE** on the command line unless you really know what you are doing because it creates a dictionary header in which the smudge bit is set and the code field points at the parameter field with no storage allotted for it. This means that the parameter field must be allotted with executable code (or the code field changed to point to some) and the smudge bit must be reset so a dictionary search can find the word. The same discussion obtains for **<BUILDS** except for the smudge bit because **<BUILDS** is defined in **fbForth 2.0** as

```
: <BUILDS CREATE SMUDGE ; ( SMUDGE toggles the smudge bit.)
```

This situation is made easier by using **<BUILDS**, **DOES>** and **DOES>ASM:** within colon definitions as

```
: NEW_DEFINING_WORD <BUILDS ... DOES> ... ;
```

or

```
: NEW_DEFINING_WORD <BUILDS ... DOES>ASM: ... ;ASM
```

You simply replace the first “...” with words you want to execute when **NEW\_DEFINING\_WORD** is compiling a new word, *e.g.*, to reserve space for and store a value in the first cell of the parameter field using **,**. You then replace the second “...” with code to be executed when the new word actually executes. It will be this code to which the code field of the new word will point.

Here, now, is an example of the use of **DOES>ASM:** in the definition of a defining word, *i.e.*, a word that creates new words:

**CONSTANT** is an **fbForth 2.0** word that defines a word, the value of which is pushed to the stack when the word is executed.

```
9 CONSTANT XXX
```

defines the word **XXX** with 9 in its parameter field and the address of the execution code of **CONSTANT** in its code field. **fbForth 2.0** defines **CONSTANT** in high-level Forth essentially as

```
: CONSTANT <BUILDS , DOES> @ ;
```

Using **DOES>ASM:**, it could also be defined with Assembler code as

<b>: CONSTANT</b>	Start colon definition of <b>CONSTANT</b> .
<b>&lt;BUILDS</b>	<b>CONSTANT</b> will create a dictionary header for the word appearing after it in the input stream when <b>CONSTANT</b> is executed. The new word's CFA will point to the address immediately following the CFA. This will be the new word's PFA, but no space will be allocated for the PFA.
<b>,</b>	Comma expects a number on the stack, which it will store at the PFA of the new word, allocating space for it.
<b>DOES&gt;ASM:</b>	The new word's CFA will be changed to point to machine code that follows <b>DOES&gt;ASM:</b> here in <b>CONSTANT</b> . The following machine code is what will run when the new word is executed:
<b>SP DECT,</b>	Make space on the stack.
<b>*W *SP MOV,</b>	Copy current (newly defined) word's parameter field contents to the stack. [ <b>W</b> (R10) contains the current word's PFA.]
<b>;ASM</b>	Return to the interpreter.

which, once you know the machine code, can be coded without the Assembler loaded as

**HEX**

```
: CONSTANT <BUILDS , DOES>ASM: 0649 , C65A , ;ASM
```

or, for machine code, perhaps it would be clearer with the following equivalent:

**HEX**

```
: CONSTANT <BUILDS , ;CODE 0649 , C65A , NEXT,
```

For **CONSTANT** , the first, high-level definition is easier to understand. They are both the same length. In this case, they both create words of the same length. However, there may come a time when only Assembler will do your bidding and **DOES>ASM:** offers that facility.

### 9.10 *ASM: and DOES>ASM: without the Assembler*

**fbForth 2.0** words using **ASM:** or **DOES>ASM:** can be written without the 3208-byte overhead of the **fbForth 2.0** Assembler by using the machine code equivalent to assembly code. The author may well write an **fbForth 2.0** program soon to do the dirty work; but, for now you must endure the painful procedure below to get the job done. Until you have tested and debugged your work, it is probably best to work with one Forth word at a time in an **fbForth 2.0** block.

1. Write, test and debug your Forth word using the **fbForth 2.0** Assembler. Here, we'll use **EX5** from § 9.4.5 for the **ASM:** example and **CONSTANT** (renamed **CONST2** to avoid confusion) from § 9.9 for the **DOES>ASM:** example.
2. Ensure that the **fbForth 2.0** Assembler is loaded by executing **21 LOAD** .
3. Ensure that the dump routines are loaded by executing **16 LOAD** .
4. Load the screen that contains the definition of your Forth word and continue with (5) in the appropriate section below.

### 9.10.1 ASM: without the Assembler

Refer to the example in § 9.4.5 for the following:

- Use ' to find the PFA of **EX5** and dump from the PFA to the end of the word:

```
HERE ' EX5 SWAP OVER - DUMP
```

will dump this to the screen:

```
AE52: C0A1 2000 C8A2 2016 .. ... .
AE5A: 201A 045F          .._
ok:0
```

The column at the left indicates the addresses in RAM where the hexadecimal cells to the right are located. The 8-character, right-hand column is their ASCII representation.

- The last cell should be **045Fh**, corresponding to the **;ASM** instruction.
- Write the high-level part of the word (**ASM: EX5**) followed by the machine code after **EX5** using the dump above to compile the hexadecimal value for each cell with **,** starting with the first cell (parameter field) and ending with **;ASM** (instead of **045Fh**) as follows:

```
HEX
ASM: EX5 C0A1 , 2000 , C8A2 , 2016 , 201A , ;ASM
```

or

```
CODE EX5 C0A1 , 2000 , C8A2 , 2016 , 201A , NEXT,
```

- If all the code was assembly code, you're done. Otherwise, you need to replace values that can vary from one load to the next, such as variables, named constants and dictionary entries not part of the resident dictionary, with the high-level code used in the word's assembly language definition. In the above example, the constant **XRAM** was used, so we need to replace the value **2000h** with the reference that put it there. In this case **XRAM** is used three times to get the cells with **2000h**, **2016h** and **201Ah**. We need to replace the **2000h** with **XRAM**, the **2016h** with **XRAM 16 +** and the **201Ah** with **XRAM 1A +** to get

```
HEX
ASM: EX5 C0A1 , XRAM , C8A2 , XRAM 16 + , XRAM 1A + , ;ASM
```

or

```
CODE EX5 C0A1 , XRAM , C8A2 , XRAM 16 + , XRAM 1A + , NEXT,
```

which can now be entered in an **fbForth 2.0** block to be loaded without the Assembler overhead.

You should test your new version of the word to verify that it is identical to the original assembly version.

### 9.10.2 DOES>ASM: without the Assembler

We need to do more work with **DOES>ASM:** than we did with **ASM:** above. We must find the CFA of (**;CODE**) that **DOES>ASM:** compiled into our word and retrieve the machine code that follows it. Refer to the example in § 9.9 (which we've renamed here as **CONST2** to avoid confusion) for the following:

- Use ' and **CFA** to find the CFA of (**;CODE**) so you can find the cell within the definition of **CONST2** that contains it:

```
HEX ' (;CODE) CFA U.
```

will display this on the screen:

**7254 ok:0**

6. Use ' to find the PFA of **CONST2** and dump from the PFA to the end of the word:

**HERE ' CONST2 SWAP OVER - DUMP**

will dump this to the screen:

```
AE4A: 71CC 6616 7254 0649 q.f.rT.I
AE52: C65A 045F .Z._
ok:0
```

The column at the left indicates the addresses in RAM where the hexadecimal cells to the right are located. The 8-character, right-hand column is their ASCII representation.

7. The last cell should be **045Fh**, corresponding to the **;ASM** instruction.
8. Write the high-level part of the word through **DOES>ASM:** followed by the machine code after **7254h** [the CFA of (**;CODE**) we found above in (5)]. Use the dump above for guidance to compile with , the hexadecimal value for each cell as follows, replacing **045Fh** with **;ASM** for clarity:

```
HEX
: CONSTANT <BUILDS , DOES>ASM: 0649 , C65A , ;ASM
```

or

```
: CONSTANT <BUILDS , ;CODE 0649 , C65A , NEXT,
```

which can now be entered on an **fbForth 2.0** screen to be loaded with only **DOES>ASM:** [or **;CODE** ] and **;ASM** [or **NEXT** , ] and without the Assembler overhead.

9. If all the code was assembly code, as it is here, you're done. Otherwise, you need to replace values that can vary from one load to the next, such as variables, named constants and dictionary entries not part of the resident dictionary, with the high-level code used in the word's assembly language definition. See (8) in § 9.10.1 for an example with a named constant.
10. You should test your new version of the word to verify it is identical to the original assembly version.

## 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 **fbForth 2.0** 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 **fbForth 2.0** ISR handler. To correctly use user variable **ISR**, the following steps should be followed:

### 10.1 Installing an fbForth 2.0 Interrupt Service Routine

- 1) Create and test an **fbForth 2.0** routine to perform the function: **MYISR**
- 2) Determine the Code Field Address (CFA) of the routine in (1): ' **MYISR CFA**
- 3) Write the CFA from (2) into user variable **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 **fbForth 2.0** 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 **fbForth 2.0** Assembler words **;ASM** and **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 **fbForth 2.0** 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 blocks from file to printer while other activities are taking place. This final application will require that file 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 **fbForth 2.0** can be written as either colon definitions or as **ASM:** 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 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 ;</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 + 5</b> down to <b>PAD + 1</b> )
<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)
	( <i>Note:</i> the CFA must be in user variable <b>ISR</b> before writing to <b>83C4h</b> )

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 **fbForth 2.0** 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, a group of CRU (Communications Register Unit) instructions and some additional words that make the stack easier to manipulate.

## 11.1 *BSAVE and BLOAD*

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

The word **BSAVE** is used to save binary images of the dictionary. It has been made part of the resident dictionary in **fbForth 2.0**; so, you no longer need to load it from FBLOCKS. **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 block number *blk<sub>1</sub>* to which the saved image will be written.

**BSAVE** will use as many **fbForth 2.0** blocks as necessary to save the dictionary contents from the address given on the stack to **HERE**. These are saved with 1000 bytes per **fbForth 2.0** block until the entire image is saved. **BSAVE** returns on the stack the number *blk<sub>2</sub>* of the first available Forth block after the image.

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

Byte #	Contents
0–1	Address at which the first image byte of this Forth block 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.

Information for the Forth and Assembler vocabularies is stored in low RAM. This means that, at least, one of **CONTEXT** and **CURRENT** must point to the Forth vocabulary before executing **BSAVE** to insure that **BLOAD** will properly restore vocabulary information. This can be effected by executing **FORTH** immediately before executing **BSAVE** . Furthermore,

if the TMS9900 Assembler is loaded before issuing **BSAVE**, a pointer to the Assembler vocabulary must also be preserved. The following example shows one way to do this:

```
FORTH DEFINITIONS  ok:0
ASSEMBLER  ok:0
' TASK 28 BSAVE . 32  ok:0
```

This way, **CURRENT** will be pointing to the Forth vocabulary and **CONTEXT** will be pointing to the Assembler vocabulary when **BSAVE** saves the binary image of the user dictionary.<sup>17</sup>

**BLOAD** ( *blk --- flag* )

**BLOAD** is part of your **fbForth 2.0** 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 an **fbForth 2.0** block number *blk* 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 / 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 Using BSAVE to Customize How fbForth 2.0 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 **fbForth 2.0**. 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 **fbForth 2.0** and the last word that *cannot* be forgotten using **FORGET**. Its definition is simply

```
: TASK ;
```

Its address can be used to **BSAVE** a personalized **fbForth 2.0** 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 37 empty blocks of FBLOCKS, starting with block 27, to save your system image:

```
' TASK 27 BSAVE .
```

(*Be sure to back up the original FBLOCKS file before trying this!*). It is important that you ensure that this procedure does not compromise **fbForth 2.0** system blocks you may need for your new personalized system. The **.** after **BSAVE** will report the next available block from the value left on the stack. Subtracting the starting block number (27, in this case) from that number will tell you how many blocks it took to save the binary image in the above **BSAVE** line.

<sup>17</sup> The problem with this scenario is that the user must remember to restore **CONTEXT** and **CURRENT** to what the user expects them to be after **BLOAD**ing the image. A future revision of **fbForth 2.0** will fix **BSAVE** to properly save the Forth and Assembler vocabulary pointers and **BLOAD** to properly restore them.

You now need to add the code to block 1 to load what you have just saved the next time you boot your system. You currently have lines 5 – 15 to add your code as long as it eventually ends with **27 BLOAD** for the above case. You also must remove (or put at the end of your added code) the **;S** at the end of line 4 because **;S** exits loading and interpreting the block. This will load your **BSAVE**d system and it will happen a lot faster than loading the text blocks because they now don't need to be interpreted.

### 11.1.2 An Overlay System with BSAVE and 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 blocks 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. 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 block 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 **16 LOAD** 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.
3. Load application.
4. Load first overlay.
5. **BSAVE** application using the address of **TASK** to a free Forth block:
 

```

' TASK 30 BSAVE .

```
6. **BSAVE** first overlay using the address of **OVERLAYS** to a free Forth block:
 

```

' OVERLAYS 40 BSAVE .

```
7. For each overlay following the first do the following:
  - a) **FORGET OVLY\_STRT**
  - b) **100 LOAD** (100 should be where the Forth block for next overlay resides.)
  - c) **' OVERLAYS 50 BSAVE .** (Obviously, 50 should be a different block 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 (1<sup>st</sup> Ed.)*, p. 80ff. It will be necessarily slower to load overlays because it involves interpreting source blocks. 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 **fbForth 2.0** block 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 blocks will be text blocks, 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** ( *blk ---* )

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 **fbForth 2.0** Instruction Pointer over it.

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

is used to compile **SLIT** and the desired character string into the current dictionary definition. See the **fbForth 2.0** Glossary ( Appendix D ) for more detail.

To use **CLOAD**, there must always be a Forth block 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 block number on the stack is loaded.

**27 CLOAD F00**

This instruction, for example, will load **fbForth 2.0** block 27 only if a dictionary search via (**FIND**) fails to find **F00**. **F00** should be the last word loaded by the command **27 LOAD** to insure all the code dependencies were loaded.

It is also possible to use **CLOAD** to abort the loading of the currently loading **fbForth 2.0** block. This is done by using the command:

**0 CLOAD TESTWORD**

If this line of code were located on **fbForth 2.0** block 50, and the word **TESTWORD** were 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 an **R->BASE** at execution time.

## 11.3 CRU Words

The five words below have been included to assist in performing CRU (Communications Register Unit) related functions. They allow the **fbForth 2.0** programmer to perform the **LDCR**, **STCR**, **TB**, **SB0** and **SBZ** operations of the TMS9900 without using the Assembler. See CRU documentation in the *Editor/Assembler Manual* for more information. These words are not part of the resident dictionary. They must be loaded from block 5 of FBLOCKS (01SEP2014 or later). You can always type **MENU** to view the loadable options for **fbForth 2.0**.

Please note that the CRU base address used here is the CRU bit number, not the “CRU Address” in § 24.3.3 “CRU Allocation” of the *Editor/Assembler Manual*. Each of the instructions below doubles the CRU bit number *addr* before putting it in R12. If you are accustomed to using the already-shifted CRU base address for a device, you will need to shift it right 1 bit (divide by 2) for *addr* in all of these words such that it is the actual CRU base bit number. For **TB**, **SB0** and **SBZ**, you will need to compose the CRU bit number from the base + the bit to be tested, set or reset because each of these words operates on bit 0 of the address (*addr*) passed to it, *i.e.*, they do not operate like their namesakes in TMS9900 Assembler. In the author’s opinion, these words should emulate the behavior of their TMS9900 Assembler counterparts; but, rather than break old TI Forth code by changing their definitions, the author offers the following comparison of **fbForth 2.0** code and TMS9900 Assembler code for the same operation:

Scenario: You wish to set CRU bit 9 of the disk controller’s CRU address space, **1100h – 11FEh** (CRU bits **880h – 8FFh**).

TMS9900 Assembler: Load R12 with **1100h** followed by **SB0 9**:

```
LI   R12,>1100
SB0  9
```

**fbForth 2.0**: Convert the CRU base address (**1100h**) to the CRU bit number (**880h**) it represents, add the bit number (**9**) to be set and push the result to the stack before executing **SB0** :

```
HEX 889 SB0 ok:0
```

which actually executes the following TMS9900 Assembler code:

```
MOV  *SP+,R12    <- -pop 889h from the stack to R12
A    R12,R12     <- -double the CRU bit number
SB0  0           <- -set the bit pointed to by R12
```

**LDCR** ( $n_1 n_2 addr \text{ ---}$ )

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 low-order  $n_2$  bits of value  $n_1$  are transferred to the CRU, where the following condition,  $n_2 \leq 15$ , is enforced by  $n_2 \text{ AND } 0Fh$ . If  $n_2 = 0$ , 16 bits are transferred. For program clarity, you may certainly use  $n_2 = 16$  to transfer 16 bits because  $n_2 = 0$  will be the value actually used by the final machine code.

**STCR** ( $n_1 addr \text{ --- } n_2$ )

Performs the TMS9900 **STCR** instruction. The CRU base address *addr* will be shifted left one bit and stored in workspace register R12 prior to executing the TMS9900 **STCR** instruction. There will be  $n_1$  bits transferred from the CRU to the stack as  $n_2$ , where the following condition,  $n_1 \leq 15$ , is enforced by  $n_1 \text{ AND } 0Fh$ . If  $n_1 = 0$ , 16 bits

will be transferred. For program clarity, you may certainly use  $n_1 = 16$  to transfer 16 bits because  $n_1 = 0$  will be the value actually used by the final machine code.

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

**TB** performs the TMS9900 **TB** instruction. The bit at CRU address *addr* is tested by this instruction. Its value (*flag* = 1|0 ) is returned to the stack. The CRU base address *addr* will be shifted left one bit and stored in workspace register R12 prior to executing the TMS9900 instruction, **TB** 0, to effect testing the bit.

**SBO** ( *addr* --- )

This word expects to find on the stack the CRU address *addr* of the bit to be set to 1. **SBO** will put this address into workspace register R12, shift it left (double it) and execute TMS9900 instruction, **SBO** 0, to effect setting the bit.

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

This word expects to find on the stack the CRU address *addr* of the bit to be reset to 0. **SBZ** will put this address into workspace register R12, shift it left (double it) and execute TMS9900 instruction, **SBZ** 0, to effect resetting the bit.

## 11.4 Useful Additional Stack Words

The words in this section were, for the most part, required by the new Stack-based String Library (see Chapter 14). The author added a few complementary words to round out the set. They are loaded from block 41 of FBLOCKS (21NOV2014 or later).

**2DUP** (  $n_1 n_2$  ---  $n_1 n_2 n_1 n_2$  )

Duplicate the top two numbers on the stack.

**2DROP** (  $n_1 n_2$  --- )

Drop the top two numbers from the stack.

**NIP** (  $n_1 n_2$  ---  $n_2$  )

Remove from the stack the number that is under the top number.

**TUCK** (  $n_1 n_2$  ---  $n_2 n_1 n_2$  )

Put a copy of the top number under the top two numbers on the stack.

**CELLS** (  $n$  ---  $2n$  )

Replace  $n$  (a number of cells) with  $2n$  (the number of bytes in  $n$  cells).

**-ROT** (  $n_1 n_2 n_3$  ---  $n_3 n_1 n_2$  )

Rotate right the top three numbers on the stack, resulting in the top number on the bottom.

**PICK** (  $+n$  ---  $[n]$  )

Copy to the top of the stack the  $n^{\text{th}}$  number down. The  $0^{\text{th}}$  number is the top number.  $[n]$  means “the contents of cell  $n$  from the top of the stack”. The number  $n$  must be positive.

0 **PICK** is equivalent to **DUP** .

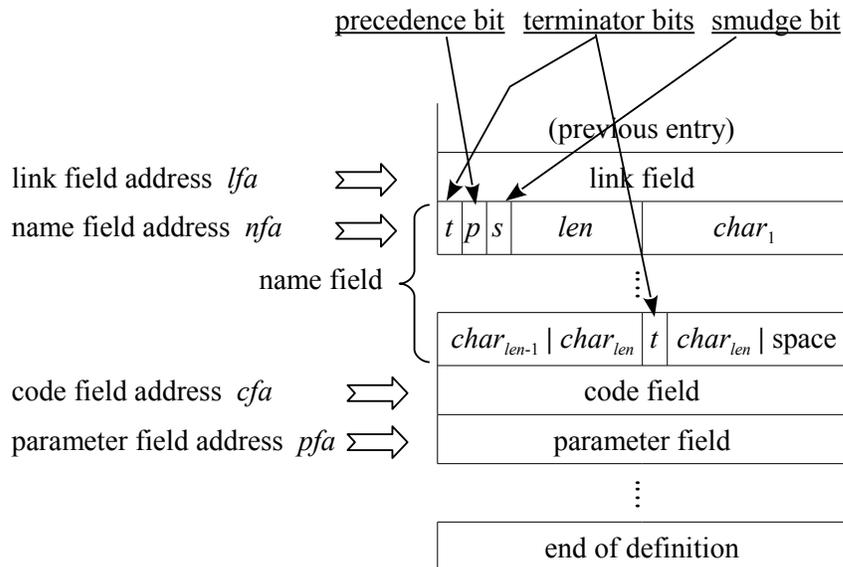
1 **PICK** is equivalent to **OVER** .

- ROLL**           ( [ *n* ] ... [0] +*n* --- [ *n*-1 ] ... [0] [ *n* ] )  
 Rotate left the top *n*+1 numbers on the stack, resulting in the *n*<sup>th</sup> number down moving to the top of the stack. The number *n* must be positive. The source for **ROLL** was Marshall Linker via George Smyth's "Forth Forum" column in the *MANNERS Newsletter* (1985) Vol. 4(5), pp. 12 – 16.  
       **0 ROLL** is a null operation.  
       **1 ROLL** is equivalent to **SWAP** .  
       **2 ROLL** is equivalent to **ROT** .
- WITHIN**       ( *n*<sub>1</sub> *n*<sub>2</sub> *n*<sub>3</sub> --- *flag* )  
 Result *flag* is true (1) if  $n_2 \leq n_1 < n_3$  and false (0) otherwise.
- <>**           ( *n*<sub>1</sub> *n*<sub>2</sub> --- *flag* )  
 Result *flag* is true (1) if  $n_1 \neq n_2$  and false (0) otherwise.
- \$.**           ( *n* --- )  
 Display the top number on the stack as an unsigned hexadecimal number.
- EXIT**       ( --- )                   [ *immediate word* ]  
**EXIT** is a synonym for **;S** , which stops interpretation of a Forth block or ends the current word's execution and returns to the calling procedure.

## 12 fbForth 2.0 Dictionary Entry Structure

The structure of an entry (a Forth *word*) in the **fbForth 2.0** dictionary is briefly described in this chapter to give the reader a better understanding of **fbForth 2.0** 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 vocabulary list to which the 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 name field is a packed character string (see footnote 4 on page 19) in that the first byte is the length byte followed by the character string that represents the name. 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 **fbForth 2.0** words have a maximum length of 31 characters. The name field in **fbForth 2.0** 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 **fbForth 2.0** 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 $pfa$ onto stack
<b>CONSTANT</b>	Runtime code of <b>CONSTANT</b>	Pushes contents of word's $pfa$ 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 $pfa$
<b>CODE</b>	$pfa$ of word	Executes machine code stored beginning at this word's $pfa$
<b>ASM:</b>	$pfa$ of word	Executes machine code stored beginning at this word's $pfa$

## 12.4 Parameter Field

The parameter field follows the code field. The parameter field address *pfa* points to this address, which can be retrieved by using ' :

```
' cccc
```

where **cccc** is the name of the Forth word for which you desire the *pfa*.. If the word is not found, however, you will get an error message as well as two values on the stack that indicate the character offset and screen number (0 for terminal) of the error. **-FIND** (*q.v.*) will also return the *pfa* along with the length byte of the name field and *true* if the word is found in the dictionary or just *false* if it is not found. It is used the same way as ' ; but, more work is required if all you want is the *pfa*, so it is more suited to colon definitions:

```
-FIND cccc DROP DROP
```

If you know only the *nfa*, you can retrieve the *pfa* by executing **PFA** .

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
<b>ASM:</b>	Machine code comprising this word's runtime code

## 12.5 Notes on Resident Dictionary Words

The structure of a Forth word described above is strictly true only for words defined in CPU RAM space. For words in the resident dictionary, the various fields are split into two ROM banks in the 32KB cartridge, with the link and name fields in bank 2 and the code and parameter fields in bank 0. There are three additional fields following the name field of each word in bank 2. The first contains the *cfa*; the second, the *pfa*; the third, the address of the previous word's *pfa* pointer.

The resident dictionary structure is only searchable in bank 2, where two linked lists reside. The first is the normal link-field-to-name-field chain and the second is a *pfa*-pointer-pointer-to-*pfa*-pointer chain, mentioned in the previous paragraph. The only way to find the *nfa*, given a *pfa* in ROM, is for **NFA** to search the latter chain from the last ROM definition to the first until the *pfa*'s pointer is found, back up three bytes to the last byte of the name field and traverse the name field to its first byte. The address of that byte is the *nfa*.

To find the *lfa*, **LFA** first finds the *nfa* via **NFA** and then backs up two bytes, *i.e.*,  $lfa = nfa - 2$ .

Finally, **CFA** finds the *cfa* from the *pfa* in the same way for both ROM and RAM:  $cfa = pfa - 2$ .

# 13 Screen Fonts and the Font Editor

Words introduced in this chapter:

**FONTED      FNT                  SCRFNT      USEFFL**

## 13.1 Loading the Default Screen Font

The **fbForth 2.0** word **FNT** loads either the default font or a user font into the Pattern Descriptor Table (PDT) depending on the value of the user variable **SCRFNT**. The default font is loaded by **FNT** from the cartridge at **fbForth 2.0** startup because the value at startup in **SCRFNT** is 0. The default font contains the patterns for ASCII character codes 0 – 127. The font pattern for each character is 8 bytes, which means that 1KB of pattern code is loaded into the PDT. This font contains true lowercase characters with true descenders.

It should be noted that each time the VDP mode is changed (except for GRAPHICS2 [bitmap]), the current screen font is reloaded. The user can always change the value of **SCRFNT** to 0 to force reloading the default screen font. However, if **SCRFNT** is changed to a nonzero value without ever having given **fbForth 2.0** a valid font filename (see next section), the next execution of **FNT** will result in a file I/O error.

## 13.2 User Fonts

**fbForth 2.0** allows users to load their own fonts instead of the default font from cartridge ROM as long as a few rules are followed:

- The font file should not be larger than 2KB, *i.e.*, it should not code for more than 256 characters. Attempting to load font files larger than 2KB will result in a file I/O error.
- The first character is assumed to code for ASCII 0 and must start at the first byte of the file. *TI Writer* CHARA1-style font files will not work because the font code begins at byte 6, not byte 0, of the file. The Font Editor (see next section) can be used to correct the file's pattern registration so that it will load properly.
- Font files larger than 1KB (ASCII 0 – 127) will cause a problem in SPLIT2 mode in that characters 128 – 255 will appear in the top of the bitmap graphics part of the screen. This can be corrected with the following code after **SPLIT2** has been invoked:

```
PDT 1024 + 1024 0 VFILL
```

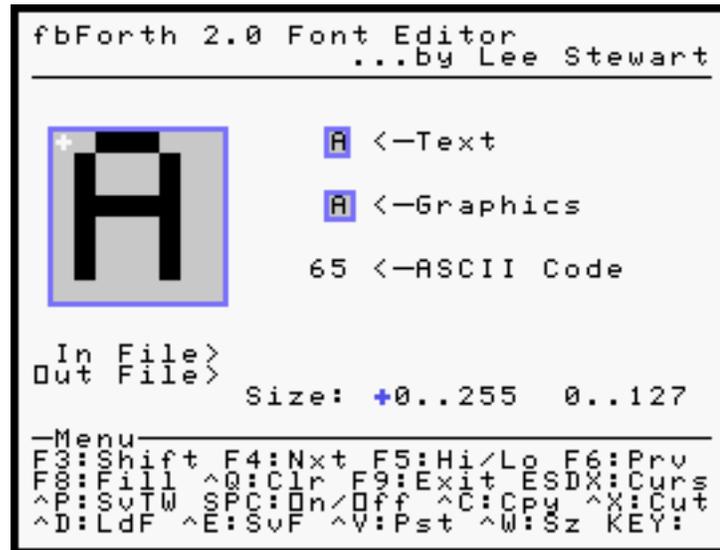
To install a user font file that will be loaded the next time **FNT** is executed, the word **USEFFL** must be followed by the full pathname of the font file as in the following example:

```
USEFFL DSK1.MYFONT
```

Once **USEFFL** executes without an error, the user can change **SCRFNT** at will to alternate between the default font (**SCRFNT** = 0) and the user-supplied font (**SCRFNT** ≠ 0).

### 13.3 Using the Font Editor

Typing **FONTEd** opens the font editor with 2KB of the current font loaded from the PDT and with the uppercase 'A' (ASCII 65) in the edit box. A full 2KB is always loaded into the edit buffer from the PDT regardless of the actual size of the screen font loaded by the last execution of **FNT** :



Editing the font will not affect the current font because the working buffer is not the PDT. Though all the menu keys are shown at the bottom of the screen, they are rather cryptic, so here is a brief description:

Function	Keystroke
<FCTN+4>	Next character pattern
<FCTN+6>	Previous pattern
<FCTN+S>	Move edit cursor left
<FCTN+D>	Move edit cursor right
<FCTN+X>	Move edit cursor down
<FCTN+E>	Move edit cursor up
<FCTN+5>	Select character pattern 128 characters up/down
<CTRL+Q>	Turn off all pixels of character pattern block
<FCTN+8>	Turn on all pixels of character pattern block
<CTRL+C>	Copy character pattern to clipboard
<CTRL+X>	Cut character pattern to clipboard
<CTRL+V>	Paste character pattern from clipboard
<SPACE>	Toggle current character pattern pixel on/off
Key	Select any character key to edit its pattern
<CTRL+D>	Load font file
<FCTN+3>	Toggle 6-byte font pattern offset

Function	Keystroke
<CTRL+W>	Toggle output file size between 1KB (ASCII 1 – 127) and 2KB (ASCII 0 – 255)
<CTRL+E>	Save font in fbForth 2.0 format
<CTRL+P>	Save font in TI Writer format
<FCTN-9>	Exit font editor

To aid in editing, the current pattern is shown actual size in a 6x8-pixel, text-mode character box and an 8x8-pixel, graphics-mode character box. The ASCII value of the current character is also shown. The edit cursor is a white '+' and can be moved around with the arrow keys. The pixel under the cursor can be toggled on/off with the space bar. You will see any changes appearing in all three display boxes.

You can load *TI-Writer*-format font files and toggle the font offset with <FCTN+3> so the patterns are properly registered for editing. Saving the font as an **fbForth-2.0**-style font will write the font starting at the first byte of the file. Saving it as a *TI-Writer*-style font will write six bytes of zeroes before writing the font to the file. The only other output variation is choosing whether to save only patterns 0 – 127 (1KB file) or all patterns 0 – 255 (2KB file). The blue '+' character marks which of these occurs with the next file save and is toggled with <CTRL+W>.

### 13.4 Modifying the 64-Column Editor's Font

The 64-column editor does not use the normal screen fonts described above, so modifying it will be a bit more of a challenge. The following graphic shows the complete character set, with the true lowercase letters and the '@' designed by the author, for bitmap mode:

```
!"#$%&'()*+,-./0123456789:;<=>?@
ABCDEFGHIJKLMN O PQRSTU VWXYZ[\]^_`
abcdefghijklmnopqrstuvwxyz{|}~
```

This character set is used principally by the 64-column editor via the word **SMASH** defined in block 13 of FBLOCKS. Designing the characters for a 3x7 matrix was quite a challenge. The '@' should probably be re-designed.

To design your own 3x7 font, use a 4x8-pixel grid for each character. Each row of the character pattern is one nybble of the pattern code; so, each character is four bytes. You then need only overwrite the character codes for the tiny character set in block 15, lines 3 – 9 of FBLOCKS. Loading the following three blocks from a blocks file of your design, with contiguous block numbers, would accomplish this [Note: The comment, ( ^0 ) (Shift+0), on line 5 is a substitute for ( ) , a syntax error]:

**BLOCK #10**

```

0 ( DEFINITIONS FOR true lowercase TINY CHARACTERS) BASE->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 , 4000 , ( * ) 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 , AE86 , ( @ )
13 04AA , EAAA , ( A ) 0CAA , CAAC , ( B ) 0688 , 8886 , ( C )
14 0CAA , AAAC , ( D ) 0E88 , C88E , ( E ) 0E88 , C888 , ( F )
15 -->

```

**BLOCK #11**

```

0 ( DEFINITIONS FOR true lowercase 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 , AAAE , ( O )
4 0CAA , C888 , ( P ) 0EAA , AAEC , ( Q ) 0CAA , CAAA , ( R )
5 0688 , 422C , ( S ) 0E44 , 4444 , ( T ) 0AAA , AAAE , ( 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 , ( ` ) 000E , 2EAE , ( a )
10 088C , AAAC , ( b ) 0006 , 8886 , ( c ) 0226 , AAA6 , ( d )
11 0004 , AE86 , ( e ) 0688 , E888 , ( f ) 0006 , A62C , ( g )
12 088C , AAAA , ( h ) 0404 , 4442 , ( i ) 0202 , 22A4 , ( j )
13 088A , ACAA , ( k ) 0444 , 4444 , ( l ) 000A , EEAA , ( m )
14 0008 , EAAA , ( n ) 0004 , AAA4 , ( o ) 000C , AC88 , ( p )
15 -->

```

**BLOCK #12**

```

0 ( DEFINITIONS FOR true lowercase TINY CHARACTERS concluded)
1 0006 , A622 , ( q ) 0008 , E888 , ( r ) 0006 , 842C , ( s )
2 044E , 4442 , ( t ) 000A , AAA6 , ( u ) 000A , AAA4 , ( v )
3 000A , AEEA , ( w ) 000A , A4AA , ( x ) 000A , A62C , ( y )
4 000E , 248E , ( z ) 0644 , 8446 , ( { ) 0444 , 0444 , ( | )
5 0C44 , 244C , ( } ) 02E8 , 0000 , ( ~ ) 0EEE , EEEE , ( DEL )
6 DECIMAL TCHAR 15 BLOCK 192 + 194 MOVE FLUSH FORGET TCHAR R->BASE
7
8
9
10
11
12
13
14
15

```

## 14 The Stack-based String Library

This chapter describes the stack-based string library package ported by Mark Wills to **fbForth 2.0** from code he wrote for his own TurboForth V1.2 (see his website: *turboforth.net*). The string library simplifies string handling through the use of a string stack. To accommodate users of the string library in **fbForth 2.0** and with permission, the author has freely edited Mark's documentation written 2/27/2014 for TurboForth V1.2 (*cf.* his website) for this chapter. In addition, the author rendered some minor assistance in this **fbForth-2.0** port.

[ Mark Wills' Note for his PDF documentation: *This paper is adapted from a paper that I wrote describing a string library that I developed for ANS Forth systems. The code<sup>18</sup> presented at the end of this paper has been modified where appropriate for compatibility with the Forth-83<sup>19</sup> standard, and, specifically, TurboForth V1.2. The original ANS paper can be downloaded as a PDF: [turboforth.net/downloads/docs/ANS\\_String\\_Lib.pdf](http://turboforth.net/downloads/docs/ANS_String_Lib.pdf). ]*

Mark Wills' abstract for TurboForth V1.2:

String handling is not one of Forth's strong points. Out-of-the-box support for strings is all but non-existent in standard Forth. Whilst the concept of strings does exist in the language, relatively few words are provided to allow effective string manipulation. The normal approach for Forth programmers is to roll one's own string functions as required. Issues such as heap allocation and de-allocation, and memory fragmentation are thorny issues which are often passed over in preference for a 'quick-and-dirty' solution that solves the problem at hand. This paper presents a Forth-83<sup>19</sup> Forth compliant library which affords the Forth programmer such facilities as string constants, transient strings, and a wide range of string manipulation words. Issues such as memory allocation, memory de-allocation and memory fragmentation are rendered irrelevant through the provision of a string stack, which is used to host and manipulate transient strings.

### 14.1 Introduction—The Concepts behind the Library

The String Library offers two types of strings:

- Transient strings—these exist on a string stack, which is separate from the data and return stacks. Their size is variable, and may be increased and decreased in size as necessary.
- String constants—declared with a maximum size, string constants are generally initialized to a constant string value throughout the life of the application. It is possible to change the string assigned to a string constant, but its maximum size cannot be changed.

#### 14.1.1 Coding Conventions

The following coding-style conventions are employed in the library:

---

18 The code for the **fbForth 2.0** version of the String Stack Library is located in Appendix J starting at block 42.

19 **fbForth 2.0** is not Forth-83 compliant. It is based on fig-Forth, with some features of Forth-79 thrown in, both of which predate the Forth-83 standard. Most of the modifications necessary to port the string library to **fbForth 2.0** were due to this disparity.

- Words intended to be called by a user of the library all end with a dollar sign. The dollar sign should be read as the word “string”. For example, **DROP\$** would be pronounced “drop string”.
- Low-level words internal to the library for housekeeping, or general factors of code are surrounded with parentheses. For example: **(lenOf\$)** .

### 14.1.2 Stack Notation

Normal Forth stack notation conventions are used. Where words have an effect on the string stack, the string stack effects are shown alongside the normal data stack effects. For example,

```
VAL$ ( --- ud ) ( SS: str --- )
```

The above example indicates that the word **VAL\$** takes a string from the string stack and results in an unsigned double being pushed to the data stack.

The suggested pronunciation of the word is also given in quotes following the stack signature(s).

If a word expects additional characters from the input stream (the terminal input buffer or a block buffer), “IS:” is shown on the “before execution” side of the stack effects followed by a descriptor in italics and, possibly, a terminator (usually a double quote) in the font used for Forth words in this document.

For example,

```
$CONST ( max_len IS:name" --- )
```

### 14.1.3 Loading the String Stack Library

The String Stack Library is supplied in the FBLOCKS file, ready to **LOAD** . Typing **MENU** will provide you with instructions for **LOADing** the library. Currently, it is **LOADed** by typing **42 LOAD** .

The string stack must be initialized to some convenient size by executing **INIT\$** once the library is **LOADed**:

```
512 INIT$ ok:020
```

will initialize the string stack to 512 bytes. **INIT\$** should only be executed once because initializing the string stack a second time will orphan the previous instance and waste memory.

## 14.2 String Constant Words

Since only a handful of words are associated with string constants, they will be documented first:

```
$CONST ( max_len IS:name --- ) RUNTIME: ( --- $Caddr ) “string constant”
```

The word **\$CONST** declares a string constant. Declared at compile time, string constants require a maximum length and a name. For example,

```
50 $CONST WELCOME ok:0
```

---

<sup>20</sup> Note that computer responses are underlined as is the case here for ok:0 .

The above example declares a string with a maximum size of 50 characters. It shall be referenced in code using the name **WELCOME** .

Note the runtime stack effect. It can be seen that at runtime, when the name of the string is referenced, it shall push its address to the data stack. The label *\$Caddr* indicates that it is the address of a string constant. A string constant pushes the address of its maximum length field which can be read with the word **MAXLEN\$** .

**MAXLEN\$** (*\$Caddr* --- *max\_len*) “maximum length of string”

Given the address of a string constant on the data stack the word **MAXLEN\$** returns the maximum allowed string length for that string constant. For example,

```
50 $CONST WELCOME ok:0
WELCOME MAXLEN$ . 50 ok:0
```

The above code fragment shall display the value 50.

**:=** (*\$Caddr* IS:*string*" --- ) “assign string constant”

Given the address of a string constant on the data stack, the word **:=** initializes the string constant with the string from the input stream. For example,

```
50 $CONST WELCOME ok:0
WELCOME := " Hello and welcome!" ok:0
```

**.\$CONST** (*\$Caddr* --- ) “display string constant”

Given the address of a string constant on the data stack the word **.\$CONST** shall display the string. For example,

```
50 $CONST WELCOME ok:0
WELCOME := " Hello and welcome!" ok:0
WELCOME .$CONST CR
Hello and welcome! ok:0
```

**CLEN\$** (*\$Caddr* --- *len*) “string constant length”

Given the address of a string constant on the data stack the word **CLEN\$** returns its actual length on the data stack. For example,

```
50 $CONST WELCOME ok:0
WELCOME := " Hello and welcome!" ok:0
WELCOME CLEN$ . 18 ok:0
```

The above code displays 18—the length of the string **WELCOME** .

**>\$** (*\$Caddr* --- ) (SS: --- *str*) “to string stack”

Given the address of a string constant on the data stack the word **>\$** copies the contents of the string to the string stack where it can be manipulated. For example,

```
50 $CONST WELCOME ok:0
WELCOME := " Hello and welcome!" ok:0
WELCOME >$ ok:0
```

Note that the string stack has received a copy of the string contained within **WELCOME** . The string **WELCOME** still exists as a string constant.

### 14.3 String Stack Words

The convention within this document is to refer to words that exist on the string stack as transient strings. They are referred to as transient strings because they generally only exist for a short time on the string stack. Strings are placed on the string stack (which is separate from the data and returns stacks) and then manipulated in some way before being consumed. Memory allocation and de-allocation is managed by virtue of the strings being on the stack in the same way that the size of the data stack is managed by simply adding or removing values on the data stack.

**\$"** (IS:*string*" ---) (SS: --- *str*) “string to string stack”

The word **\$"** takes a string from the input stream and pushes it to the string stack. The end of the string is indicated by a quotation mark. For example,

```
$" Hello, World!" ok:0
```

In this example the string “Hello, world!” is pushed directly to the string stack, thus becoming the top item on the string stack.

Note that **\$"** is a state-smart word. It can be used in both colon definitions and also directly at the command line. The correct action will be taken in either case.

In order that the runtime actions of **\$"** may be compiled into a definition if so desired, the runtime action of this word is encapsulated within the word (**\$"**). Therefore, if the runtime behavior of this word is to be compiled into another word, one must compile the word (**\$"**).

**DUP\$** (---) (SS: *str*<sub>1</sub> --- *str*<sub>1</sub> *str*<sub>1</sub>) “duplicate string”

The word **DUP\$** duplicates the top item on the string stack. For example,

```
$" Hello, World!" DUP$ ok:0
```

The string stack now contains two copies of the string.

**DROP\$** (---) (SS: *str* ---) “drop string”

The word **DROP\$** removes the topmost string item from the string stack. For example,

```
$" Hello, World!" ok:0
$" How are you?" ok:0
DROP$ ok:0
```

At this point the string “Hello, World!” is the topmost string the string stack. “How are you?” was pushed onto the string stack, but it was immediately dropped.

**SWAP\$** (---) (SS: *str*<sub>1</sub> *str*<sub>2</sub> --- *str*<sub>2</sub> *str*<sub>1</sub>) “swap string”

The word **SWAP\$** swaps the topmost two strings on the string stack. For example,

```
$" Hello, World!" ok:0
$" How are you?" ok:0
SWAP$ ok:0
```

At this point, the string “Hello, World!” is the topmost string on the string stack.

**NIP\$** ( --- ) ( SS:  $str_1 str_2$  ---  $str_2$  ) “nip string”

The word **NIP\$** removes the string underneath the topmost string from the string stack. For example,

```
$" red"  ok:0
$" blue" ok:0
```

At this point, “blue” is on the top of the string stack, with “red” underneath it.

```
NIP$
```

At this point, “red” has been removed from the string stack, leaving “blue” as the topmost string.

**OVER\$** ( --- ) ( SS:  $str_1 str_2$  ---  $str_1 str_2 str_1$  ) “over string”

The word **OVER\$** pushes a copy of the string  $str_1$  to the top of the string stack, above  $str_2$ . For example,

```
$" red"  ok:0
$" green" ok:0
OVER$ ok:0
```

At this point, the string stack contains the following strings:

```
“red” (the topmost string)
“green”
“red”
```

**ROT\$** ( --- ) ( SS:  $str_1 str_2 str_3$  ---  $str_2 str_3 str_1$  ) “rotate strings”

The word **ROT\$** rotates the top three strings to the left. The third string down (prior to the execution of **ROT\$** ) moves to the top of the string stack.

Note that, for ease of implementation, this routine copies (using **PICK\$** ) the strings to the top of the string stack in their correct final order, then removes the 3 original strings underneath. Consequently, it is possible to run out of string stack space. If this happens, the condition will be correctly trapped in (**set\$SP**) .

**-ROT\$** ( --- ) ( SS:  $str_1 str_2 str_3$  ---  $str_3 str_1 str_2$  ) “rotate strings”

The word **-ROT\$** rotates the top three strings to the right. The top string (prior to the execution of **-ROT\$** ) moves to the third position. Note that, for ease of implementation, this routine copies (using **PICK\$** ) the strings to the top of the string stack in their correct final order, then removes the 3 original strings underneath. Consequently, it is possible to run out of string stack space. If this happens, the condition will be correctly caught in (**set\$SP**) .

**>\$CONST** (  $\$Caddr$  --- ) ( SS:  $str$  --- ) “to string constant”

The word **>\$CONST** takes the topmost string from the string stack and moves it into the string constant whose address is on the data stack. For example,

```
4 $CONST COLOR ok:0
$" red" COLOR >$CONST ok:0
```

At this point, the string constant **COLOR** has the value “red”. To verify, display the string using **.\$CONST** as follows:

```
COLOR .$CONST red ok:0
```

**+\$** ( --- ) ( SS:  $str_1 str_2 - str_1 \& str_2$  ) “concatenate strings”

The word **+\$** replaces the top two strings on the string stack with their concatenated equivalent. For example,

```
 $" red" $" blue" +$ ok:0
```

At this point, “red” and “blue” have been removed from the string stack. The topmost string on the string stack has the value “redblue”. Note that the topmost string goes to the right of the newly concatenated string.

**LEN\$** ( --- *len* ) ( SS: --- ) “length of string”

The word **LEN\$** returns the length of the topmost string on the string stack. For example,

```
 $" Hello world!" len$ . 12 ok:0
```

displays the value 12.

**MID\$** ( *start end* --- ) ( SS:  $str_1 --- str_1 str_2$  ) “mid-string”

The word **MID\$** produces a sub-string on the string stack, consisting of the characters from the topmost string starting at character *start* and ending at character *end*. For example,

```
 $" redgreenblue" 3 7 mid$ ok:0
```

At this point, the topmost two strings on the string stack are as follows:

```
“green” (the topmost item)
“redgreenblue”
```

Note, as indicated in the string stack signature, the original string ( $str_1$ ) is retained. Note also that the first character in the string (the leftmost character) is character number 0.

**LEFT\$** ( *len* --- ) ( SS:  $str_1 --- str_1 str_2$  ) “left of string”

The word **LEFT\$** pushes the leftmost *len* characters to the string stack as a new string. The original string is retained. For example,

```
 $" redgreenblue" 3 LEFT$ ok:0
```

The above causes the string “red” to be pushed to the string stack.

**RIGHT\$** ( *len* --- ) ( SS:  $str_1 --- str_1 str_2$  ) “right of string”

The word **RIGHT\$** causes the rightmost *len* characters to be pushed to the string stack as a new string. The original string is retained. For example,

```
 $" redgreenblue" 4 RIGHT$ ok:0
```

The above causes the string “blue” to be pushed to the string stack.

**FINDC\$** (*char* --- *pos*|-1) (SS: ---) “find character in string”

The word **FINDC\$** returns the position of the first occurrence of the character *char*, beginning at the left side of the topmost string, with the search proceeding towards the right. If the character is not found, -1 is returned. For example,

```
$" redgreenblue" 98 FINDC$ . 8 ok:0
```

Displays the value 8, as the character ‘b’ (ASCII 98) is found in the 8th character position (where the first character is character 0).

**FIND\$** (*start* --- *pos* | -1) (SS: ---) “find string in string”

The word **FIND\$** searches the second string on the string stack, starting from position *start*, for the first occurrence of the topmost string and pushes its starting position to the data stack. As a convenience, to make subsequent searches for the same substring easier, both strings are retained on the string stack. For example,

```
$" redgreenbluegreen" $" green" 0 FIND$ . 3 ok:0
```

displays the value 3, as the substring is found at character position 3 (the leftmost character being character 0). The strings “redgreenbluegreen” and “green” remain on the stack; thus, the second instance of “green” could be found if desired.

**REPLACE\$** (--- *pos* | -1) (SS: *str*<sub>1</sub> *str*<sub>2</sub> *str*<sub>3</sub> --- *str*<sub>4</sub> | [*str*<sub>1</sub> *str*<sub>2</sub>]) “replace string”

The word **REPLACE\$** searches string *str*<sub>2</sub> for the first occurrence of string *str*<sub>3</sub>. If it is found, it is replaced with the string *str*<sub>1</sub>, the position of *str*<sub>3</sub> within *str*<sub>2</sub> is pushed to the data stack, *str*<sub>1</sub> and *str*<sub>3</sub> are removed from the string stack and the new string *str*<sub>4</sub> is left on the string stack. For example,

```
512 INITS$ ok:0
" PURPLE" ok:0
" redgreenblue" ok:0
" green" ok:0
REPLACE$ . 3 ok:0
$.S
-----+-----+-----
Index|Length|String
-----+-----+-----
  0|    13|redPURPLEblue

Allocated stack space: 16 bytes
Total stack space: 512 bytes
Stack space remaining: 496 bytes
ok:0
```

If the search string *str*<sub>3</sub> is not found, -1 is pushed to the data stack, *str*<sub>1</sub> and *str*<sub>2</sub> are left on the string stack, ready for another search if desired.

**.\$** (---) (SS: *str* ---) “display string”

The word **.\$** pops the topmost string from the string stack and displays it. For example,

```
$" Hello, World!" .$ Hello, World! ok:0
```

The above code displays the string “Hello, World!” on the output device.

**REV\$** (---) (SS:  $str_1$  ---  $str_2$ ) “reverse string”

The word **REV\$** replaces the topmost string on the string stack with its reversed equivalent. For example,

```
$" green" REV$ .$ neerg ok:0
```

The above displays “neerg”.

**LTRIM\$** (---) (SS:  $str_1$  ---  $str_2$ ) “trim left of string”

The word **LTRIM\$** removes leading spaces from the topmost string. For example,

```
$"      hello!" LTRIM$ .$ hello! ok:0
```

Displays “hello!” with the leading spaces removed.

**RTRIM\$** (---) (SS:  $str_1$  ---  $str_2$ ) “trim right of string”

The word **RTRIM\$** removes trailing spaces from the topmost string. For example,

```
$" hello!      " RTRIM$ .$ hello! ok:0
```

Displays “hello!” with the trailing spaces removed.

**TRIM\$** (---) (SS:  $str_1$  ---  $str_2$ ) “trim string”

The word **TRIM\$** removes both leading and trailing spaces from the topmost string. For example,

```
$"      hello!      " TRIM$ .$ hello! ok:0
```

The above code removes leading *and* trailing spaces and displays the string.

**UCASE\$** (---) (SS:  $str_1$  ---  $str_2$ ) “convert to upper case”

The word **UCASE\$** converts all lower case characters in the topmost string to upper case. For example,

```
$" hello world! 1234" UCASE$ .$ HELLO WORLD! 1234 ok:0
```

The above displays “HELLO WORLD! 1234”

**LCASE\$** (---) (SS:  $str_1$  ---  $str_2$ ) “convert to lower case”

The word **LCASE\$** converts all upper case characters in the topmost string to lower case. For example,

```
$" HELLO WORLD! 1234" LCASE$ .$ hello world! 1234 ok:0
```

The above displays “hello world! 1234”.

**CMP\$** (--- -1|0|+1) (SS:  $str_1$   $str_2$  ---  $str_1$   $str_2$ ) “compare strings”

The word **CMP\$** performs a case-sensitive comparison of the topmost two strings on the string stack and returns -1 if  $str_1 < str_2$ , 0 if  $str_1 = str_2$  and +1 if  $str_1 > str_2$ . The strings are retained. For example,

```
$" hello" $" HELLO" CMP$ . 1 ok:0
```

Displays “1” since the first string is greater than the second (the comparison is case sensitive).

```
$" hello" $" hello" CMP$ . 0 ok:0
```

Displays “0” since the strings are identical.

```
$" hell" $" hello" CMP$ . -1 ok:0
```

Displays “-1” since the first string is less than the second.

A case in-sensitive comparison can easily be built as follows:

```
: CMPCIS$ (---flag) (SS: str1 str2 --- str1 str2)
   OVER$ OVER$ UCASE$ SWAP$ UCASE$ CMP$ DROP$ DROP$ ; ok:0
```

The above code creates copies of  $str_1$  and  $str_2$  (using **OVER\$**) then converts them both to upper case. **CMPS\$** then compares the strings placing the appropriate flag on the data stack. Finally, the uppercase versions of  $str_1$  and  $str_2$  are removed from the string stack; thus,  $str_1$  and  $str_2$  are retained, unchanged.

**PICK\$** (*index* ---) (SS: --- *str*) “pick string”

Given the index of a string on the string stack, copy the indexed string to the top of the string stack. **0 PICK\$** is equivalent to **DUP\$**, **1 PICK\$** is equivalent to **OVER\$** etc. For example,

```
$" blue" ok:0
$" green" ok:0
$" red" ok:0
2 PICK$ ok:0
```

The above causes the string “blue” to be copied to the top of the string stack.

**VAL\$** (--- *d*) (SS: *str* ---)

The word **VAL\$** uses **NUMBER** to convert the topmost string on the string stack to a double number  $d$  (2-cell, 32-bit integer) on the data stack. An error occurs if the string cannot be represented as a double number. An erroneous value (but, without an error report) will result if a convertible number is outside the signed, 32-bit range: -2147483648 – 2147483647.

The same interpretation rules apply to the putative number string that apply to a number typed at the terminal or loaded from a blocks file:

- ‘-’ and ‘.’ are the only non-numeric characters allowed.
- ‘-’ must be the first character in negative-number strings.
- ‘.’ can occur anywhere in the number string any number of times. It is ignored except that the position of the last ‘.’ relative to the right end of the number is stored in **DPL**.
- The number string is converted to a number in the current radix.

A number that is known to be a 16-bit number can be managed by dropping the leading 0 cell from the stack. A better procedure would be to **DUP** the top cell, test it and deal with the possibility that it may not be 0, which it must be for the double number to be successfully converted to a 16-bit number.

Note, in the following examples, that the decimal point only affects output—the double number on the stack is a 32-bit integer. **DPL** is updated every time **NUMBER** successfully converts a string to a double number.

Examples:

```
$" 9900" VAL$ D. 9900 ok:0
$" 9900" VAL$ DROP . 9900 ok:0
$" 1234567890" VAL$ D. 1234567890 ok:0
$" 9.900" VAL$ D. 9.900 ok:0
$" 9.945" $" 1234.0" D. D. 1234.0 994.5 ok:0
```

**\$.S** (---) (SS: ---)

The word **\$.S** displays a non-destructive string stack dump to the output device. The length of each string is given, along with the total number of strings on the string stack. The amount of space allocated to the string stack, the amount of space in use, and the amount of free space is also reported. An example appears above under the description of **REPLACE\$**.

**DEPTH\$** (--- *n*) (SS: ---)

Returns the current depth of the string stack, with 0 meaning the string stack is empty.

**RESET\$** (---) (SS: ---)

Resets, *i.e.*, empties, the string stack.

## 14.4 The String Stack

The string stack is **ALLOT**ed from dictionary space by **INIT\$**, which *must* be executed before the String Stack Library can be used. The constant (**\$\$Size**) determines the amount of space reserved and is set by **INIT\$** by the user after the library is loaded.

## 14.5 Error Checking

Error checking is included in all words that could cause a string stack underflow or overflow condition. In the event that an underflow or overflow is detected, the code aborts with an error message.

Other words such as **DUP\$** also perform checks. For example, **DUP\$** checks that there is at least one item on the string stack. **SWAP\$** checks that there are at least two items on the string stack, *etc.*

## 14.6 String Stack Format

The string stack grows from higher memory addresses to lower memory addresses.

The format of the strings on the string stack is very simple, as follows:

Actual length (1 cell)	String payload (1 char=1 byte)
------------------------	--------------------------------

## 14.7 String Constant Format

String Constants have the same format, but are preceded by a maximum length cell in order to check that a requested string can be accommodated within the string constant:

Maximum length (1 cell)	Actual length (1 cell)	String payload (1 char=1 byte)
----------------------------	---------------------------	-----------------------------------

## 14.8 Throw Codes

The words in the library perform sanity checks on input parameters where necessary. In particular, the string stack, being statically **ALLO**Ted from dictionary space, is carefully guarded, since the string stack is very likely to have code and/or data on either side of it, resulting in catastrophic software failure in the event of a string stack underflow or overflow. Where errors are detected, the library throws the following **THROW** codes:

Throw Code	Nature of Error	Thrown By
9900	String stack underflow	<b>(set\$SP)</b>
9901	String too large to assign	<b>:=</b> "
9902	String stack is empty	<b>PICK\$    DUP\$    LEN\$ &gt;\$CONST   MID\$    LEFT\$ RIGHT\$    FINDC\$    .\$ REV\$      LTRIM\$    RTRIM\$ UCASE\$    LCASE\$    DROP\$</b>
9903	Need at least 2 strings on string stack	<b>SWAP\$    NIP\$    OVER\$ FIND\$    CMP\$    +\$</b>
9904	String too large for string constant	<b>&gt;\$CONST</b>
9905	Illegal LEN value	<b>MID\$    LEFT\$    RIGHT\$</b>
9906	Need at least 3 strings on string stack	<b>ROT\$    -ROT\$    REPLACE\$</b>
9907	String is not a legal number	<b>VAL\$</b>
9999	String stack not initialized	any <b>THROW</b> if <b>(\$sSize) = 0</b>

It should be noted that the author of this library has not checked that the **THROW** codes listed here are used in other systems or libraries elsewhere.

## 14.9 Author Information

The library was developed by Mark Wills in February, 2014. The code was released to the public domain. He can be contacted by email via: [markwills1970@gmail.com](mailto:markwills1970@gmail.com).

## 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+O>	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	:	3Ah	58
ESC <CTRL+,>	1Bh	27	;<CTRL+/>	3Bh	59
FS <CTRL+;>	1Ch	28	< <FCTN+O>	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> <FCTN+I>	3Fh	63

...continued from previous page—

Character	ASCII Code		Character	ASCII Code	
	hex	decimal		hex	decimal
@	<FCTN+J>	40h 64	`	<FCTN+C>	60h 96
A	<FCTN+K>	41h 65	a		61h 97
B	<FCTN+L>	42h 66	b		62h 98
C	<FCTN+M>	43h 67	c		63h 99
D	<FCTN+N>	44h 68	d		64h 100
E		45h 69	e		65h 101
F	<FCTN+Y>	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
[	<FCTN+R>	5Bh 91	{	<FCTN+F>	7Bh 123
\	<FCTN+Z>	5Ch 92		<FCTN+A>	7Ch 124
]	<FCTN+T>	5Dh 93	}	<FCTN+G>	7Dh 125
^		5Eh 94	~	<FCTN+W>	7Eh 126
_	<FCTN+U>	5Fh 95	DEL	<FCTN+V>	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 How fbForth 2.0 differs from Starting FORTH (1<sup>st</sup> Ed.)

Page	Word	Changes Required
10	<b>BACKSPACE</b>	< <i>FCTN+S</i> > produces a backspace on the TI 99/4A.
10	<b>ok</b>	<b>fbForth 2.0</b> automatically prints a space before “ <b>ok:n</b> ”.
16		The <b>fbForth 2.0</b> dictionary can store names up to 31 characters in length.
18	<b>^</b>	Not a special character in <b>fbForth 2.0</b> .
18	<b>."</b>	Will execute inside or outside a colon definition in <b>fbForth 2.0</b> .
42	<b>/MOD</b>	Uses signed numbers in <b>fbForth 2.0</b> . Remainder has sign of dividend.
42	<b>MOD</b>	Uses signed numbers in <b>fbForth 2.0</b> . Remainder has sign of dividend.
50	<b>.S</b>	The resident <b>fbForth 2.0</b> version prints a vertical bar ‘ ’ instead of ‘0’ followed by the stack contents. The stack contents will be printed as unsigned numbers. The definition shown does not work in <b>fbForth 2.0</b> , even changing ‘S’ to <b>SP@ 2-</b> to account for vocabulary differences, because of the expectation that the bottom stack location contains ‘0’ for an empty stack. It also does not print the extra number at the left to mark the bottom of the stack when the stack is not empty.
52	<b>2SWAP</b>	This word is not in <b>fbForth 2.0</b> but can be created with the following definition: <pre>      : 2SWAP ROT &gt;R ROT R&gt; ;</pre>
52	<b>2DUP</b>	This word is not in <b>fbForth 2.0</b> but can be created with the following definition: <pre>      : 2DUP OVER OVER ;</pre>
52	<b>2OVER</b>	This word is not in <b>fbForth 2.0</b> but can be created with the following definition: <pre>      : 2OVER SP@ 6 + @ SP@ 6 + @ ;</pre>
52	<b>2DROP</b>	This word is not in <b>fbForth 2.0</b> but can be created with the following definition: <pre>      : 2DROP DROP DROP ;</pre>
57		When you redefine a word that is already in the dictionary, <b>fbForth 2.0</b> will issue a message saying “ <b>WORD isn’t unique.</b> ”. In the example, a message saying “ <b>GREET isn’t unique.</b> ” would appear.
60		In <b>fbForth 2.0</b> , there is no unique limit to the number of blocks (screens) in a blocks file except the number of blocks included when the file was created.

Page	Word	Changes Required
63-82		The <b>fbForth 2.0</b> Editor is different (much better) than the editor described in this section. Read the section of this <b>fbForth 2.0 Manual</b> describing the Editor.
83	<b>DEPTH</b>	<b>DEPTH</b> is defined in the resident <b>fbForth 2.0</b> dictionary.
84	<b>COPY</b>	<b>fbForth 2.0</b> has <b>CPYBLK</b> for this purpose, <i>q.v.</i>
84-5		Ignore Editor words.
89ff	<b>THEN</b>	<b>THEN</b> is in the <b>fbForth 2.0</b> 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>NOT</b>	This word is not in <b>fbForth 2.0</b> , but can be created with the following definition: <pre>      : NOT 0= ;</pre>
101	<b>?DUP</b>	This word is identical to <b>-DUP</b> in <b>fbForth 2.0</b> . Use the following definition if necessary: <pre>      : ?DUP -DUP ;</pre>
101ff	<b>ABORT"</b>	As with the Forth-79 Standard, <b>fbForth 2.0</b> provides <b>ABORT</b> instead of <b>ABORT"</b> .
102	<b>?STACK</b>	In <b>fbForth 2.0</b> this word automatically calls <b>ABORT</b> and prints the appropriate error message.
107	<b>2*</b>	This word is not in <b>fbForth 2.0</b> , but can be created with the following definition: <pre>      : 2* DUP + ;</pre>
107	<b>2/</b>	This word is not in <b>fbForth 2.0</b> , but can be created with the following definition: <pre>      : 2/ 1 SRA ;</pre>
108	<b>NEGATE</b>	This word is not in <b>fbForth 2.0</b> , but can be created with the following definition: <pre>      : NEGATE MINUS ;</pre>
110	<b>I</b>	This word exists in <b>fbForth 2.0</b> but also has a duplicate definition, <b>R</b> . <b>I</b> and <b>R</b> are identical in function. They both get a copy of the return stack top.
110	<b>I'</b>	This word is not in <b>fbForth 2.0</b> , but can be created with the following definition: ( <i>Note: R is a synonym for I .</i> ) <pre>      : I' R&gt; R SWAP &gt;R ;</pre>
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 correctly.
112		Ignore the last two paragraphs. They do not apply.

Page	Word	Changes Required
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 <b>fbForth 2.0</b> .
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 <b>fbForth 2.0</b> , 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 <b>fbForth 2.0</b> , 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 <b>fbForth 2.0</b> , but can be created with the following definition: <pre> : U/MOD U/ ; </pre>
161	<b>/LOOP</b>	This word is not in <b>fbForth 2.0</b> .
162	<b>OCTAL</b>	<b>OCTAL</b> does not exist in <b>fbForth 2.0</b> . See p. 163 for definition.
164-5		Numbers in <b>fbForth 2.0</b> 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 <b>fbForth 2.0</b> .
173	<b>D-</b>	This word is not in <b>fbForth 2.0</b> , but can be created with the following definition: <pre> : D- DMINUS D+ ; </pre>
173	<b>DNEGATE</b>	This word is not in <b>fbForth 2.0</b> , but can be created with the following definition: <pre> : DENEGATE DMINUS ; </pre>
173	<b>DMAX</b>	This word is not in <b>fbForth 2.0</b> , but can be created with the following definition:

Page	Word	Changes Required
		<pre> : DMAX 2OVER 2OVER D- SWAP DROP 0&lt;   IF 2SWAP ENDIF   2DROP ; </pre>
173	<b>DMIN</b>	<p>This word is not in <b>fbForth 2.0</b>, but can be created with the following definition:</p> <pre> : DMIN 2OVER 2OVER 2SWAP D- SWAP DROP 0&lt;   IF 2SWAP ENDIF   2DROP ; </pre>
173	<b>D=</b>	<p>This word is not in <b>fbForth 2.0</b>, but can be created with the following definition:</p> <pre> : D= D- 0= SWAP 0= AND ; </pre>
173	<b>D0=</b>	<p>This word is not in <b>fbForth 2.0</b>, but can be created with the following definition:</p> <pre> : D0= 0. D= ; </pre>
173	<b>D&lt;</b>	<p>This word is not in <b>fbForth 2.0</b>, but can be created with the following definition:</p> <pre> : D&lt; D- SWAP DROP 0&lt;; </pre>
173	<b>DU&lt;</b>	<p>This word is not in <b>fbForth 2.0</b>, but can be created with the following definition:</p> <pre> : DU&lt; ROT SWAP OVER OVER   U&lt;   IF <i>(determined less using high order halves)</i>     DROP DROP DROP DROP 1   ELSE <i>(test if high halves equal)</i>     =     IF <i>(equal so just test low halves)</i>       U&lt;     ELSE <i>(test fails)</i>       DROP DROP 0     ENDIF   ENDIF   ENDIF ; </pre>
174	<b>M+</b>	<p>This word is not in <b>fbForth 2.0</b>, but can be created with the following definition:</p> <pre> : M+ 0 D+ ; </pre>
174	<b>M/</b>	<p>This word is different in <b>fbForth 2.0</b> and can be changed with the following definition:</p> <pre> : M/ M/ SWAP DROP ; </pre>

Page	Word	Changes Required
174	<b>M*/</b>	Not available in <b>fbForth 2.0</b> 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 <b>fbForth 2.0</b> .
183ff		Variables in <b>fbForth 2.0</b> 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: <pre>      : VARIABLE &lt;BUILDS 0 , DOES&gt; ;</pre>
193	<b>2VARIABLE</b>	This word is not in <b>fbForth 2.0</b> , but can be created with the following definition: <pre>      : 2VARIABLE &lt;BUILDS 0. , , DOES&gt; ;</pre> 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 <b>fbForth 2.0</b> , but can be created with the following definition: <pre>      : 2! &gt;R R ! R&gt; 2+ ! ;</pre>
193	<b>2@</b>	This word is not in <b>fbForth 2.0</b> , but can be created with the following definition: <pre>      : 2@ &gt;R R 2+ @ R&gt; @ ;</pre>
193	<b>2CONSTANT</b>	This word is not in <b>fbForth 2.0</b> , but can be created with the following definition: <pre>      : 2CONSTANT &lt;BUILDS , , DOES&gt; 2@ ;</pre> 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>	<b>fbForth 2.0</b> 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> .
207	<b>CREATE</b>	The <b>CREATE</b> word of <b>fbForth 2.0</b> behaves somewhat differently. Hackers should consult fig-Forth documentation.
216	<b>EXECUTE</b>	Because this word operates a little differently in <b>fbForth 2.0</b> , it must be preceded by the word <b>CFA</b> . The example should read: <pre>      ' GREET CFA EXECUTE</pre>

Page	Word	Changes Required
217		The example illustrating indirect execution must be modified to work in <b>fbForth 2.0</b> : <pre>' GREET CFA POINTER ! POINTER @ EXECUTE</pre>
218	[ ' ]	In <b>fbForth 2.0</b> , this word is unnecessary as the word ' will take the following word of a definition when used in a definition.
219	NUMBER	In <b>fbForth 2.0</b> , NUMBER is always able to convert double precision numbers.
219	'NUMBER	<b>fbForth 2.0</b> does not use 'NUMBER to locate the NUMBER routine.
220		In <b>fbForth 2.0</b> , the name field is variable length and contains up to 31 characters. Also, the link field precedes the name field in <b>fbForth 2.0</b> .
225	EXIT	This word is ;S in <b>fbForth 2.0</b> . ;S is the word compiled by ; so to create EXIT we might use: <pre>: EXIT [COMPILE] ;S ; IMMEDIATE</pre>
225	I	In <b>fbForth 2.0</b> , the interpreter pointer is called IP , not I .
232		See Chapter 1 in this <b>fbForth 2.0 Instruction Manual</b> for instructions for loading elective blocks.
232	RELOAD	This instruction is not available in <b>fbForth 2.0</b> .
233	H	This word is DP ( dictionary pointer ) in <b>fbForth 2.0</b> .
235	'S	In <b>fbForth 2.0</b> , SP@ is used instead of 'S .
240		See Appendix F in this <b>fbForth 2.0 Instruction Manual</b> for a complete list of user variables.
240	>IN	This word is IN in <b>fbForth 2.0</b> .
245	LOCATE	<b>fbForth 2.0</b> does not support LOCATE .
256	COPY	In <b>fbForth 2.0</b> , use the word CPYBLK . CPYBLK is disk resident. See Appendix D for location and usage.
259	[ ' ]	Change the [ ' ] to ' in the bottom example. In <b>fbForth 2.0</b> , ' will compile the address of the next word in the colon definition.
261	>TYPE	Unnecessary in non-multiprogramming systems. Not present in <b>fbForth 2.0</b> .
265	RND	<b>fbForth 2.0</b> has two random number generators: RND and RNDW . See Appendix D for descriptions. See also definitions for SEED and RANDOMIZE .
266	MOVE	In <b>fbForth 2.0</b> , MOVE moves <i>u</i> words in memory, not <i>u</i> bytes. Use CMOVE instead. If you must conform to <i>Starting FORTH (1<sup>st</sup> Ed.)</i> , MOVE can be redefined: <pre>: MOVE 2/ MOVE ;</pre>

Page	Word	Changes Required
266	<CMOVE	Not present in <b>fbForth 2.0</b> . 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	WORD	In <b>fbForth 2.0</b> , the word <b>WORD</b> does not leave an address on the stack.
270	TEXT	This word is not available in <b>fbForth 2.0</b> , but can be defined as follows: <pre> : TEXT PAD 72 BLANKS PAD HERE - 1-       DUP ALLOT MINUS SWAP WORD ALLOT ; </pre> <p>If you want the count to also be stored at PAD, remove the <b>1-</b> from the definition.</p>
277	>BINARY	This is named <b>(NUMBER)</b> in <b>fbForth 2.0</b> .
277		Because <b>WORD</b> does not leave an address on the stack, it is necessary to redefine <b>PLUS</b> as follows: <pre> : PLUS 32 WORD DROP NUMBER + ." = " . ; </pre>
279	NUMBER	This definition of <b>NUMBER</b> is not compatible with <b>fbForth 2.0</b> .
281	-TEXT	Not in <b>fbForth 2.0</b> . Use the definition on page 282.
292		<b>fbForth 2.0</b> uses the word pair < <b>BUILDS ... DOES</b> > to define a new defining word. < <b>BUILDS</b> calls <b>CREATE</b> as part of its function.
297		To create a byte <b>ARRAY</b> in <b>fbForth 2.0</b> : <pre> : ARRAY &lt;BUILDS OVER , * ALLOT       DOES&gt; DUP @ ROT * + + 2+ ; </pre>
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>BUILDS</b> .
301	(DO)	This is the runtime behavior of <b>DO</b> just as listed. <b>2&gt;R</b> is not used, however.
301	DO	The given definition of <b>DO</b> is not compatible with <b>fbForth 2.0</b> . <b>fbForth 2.0</b> 's definition of <b>DO</b> is much more complex because of compile-time error checking.
303	(LITERAL)	The <b>fbForth 2.0</b> name for this word is <b>LIT</b> .
306		<b>fbForth 2.0</b> remains in compilation mode until a <b>;</b> is typed.

## Appendix D The fbForth 2.0 Glossary

**fbForth 2.0** words appear in this glossary on the left of the word's entry line and ordered in the ASCII collating sequence, displayed as a handy reference at the bottom of each page of this appendix. If the word is an immediate word, that fact is shown in the middle of the entry line as “[*immediate word*]”. The block in FBLOCKS that needs to be loaded to load the word's definition is enclosed in “[ ]” and right-justified on the entry line preceded by some or all of the description given by executing **MENU**. The word's definition can be found in or following that block. If the word is part of the core system, it is listed as “Resident”. *Note:* With the exception of **;ASM** and **NEXT, ,** words in the Assembler vocabulary are only referenced in Chapter 9 “The fbForth 2.0 TMS9900 Assembler”.

The state of the top of the parameter stack (usually referred to simply as “the stack”) before and after execution of an **fbForth 2.0** word is shown schematically as “( *before* --- *after* )”, where “*before*” and “*after*” represent 0 or more cells relevant to the **fbForth 2.0** word being described and “---” represents the execution of the word. The topmost, *i.e.*, most accessible, item on the stack is on the right. These stack effects are usually listed on the second line. However, when an **fbForth 2.0** word is a compiler word, *i.e.*, it can only appear within the definition of another word, the compilation and runtime stack effects will be shown on the lines beginning the relevant descriptions.

The stack effects of the return stack will also be shown when the return stack is affected by the execution of the **fbForth 2.0** word. 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.

For the Stack-based String Library (see Chapter14), the String Stack effects will be shown similarly as the return-stack effects with “( SS: <before> --- <after> )”. A few of these words expect information from the input stream, which will be shown as part of <before> in the parameter-stack effects with “IS:” (see Chapter 14 for more details).

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

When the following terms and abbreviations are part of the stack effects schematic, each *before* and *after* token in the schematic represents 1 cell (16-bits or 2 bytes) on the stack unless otherwise noted under “Meaning”.

Term/Abbreviation	Meaning
<i>\$Caddr</i>	string constant address
<i>addr, addr<sub>1</sub>, ...</i>	memory address
<i>b</i>	byte
<i>col</i>	column position
<b>cccc , nnnn , xxxx</b>	string representations
<i>cf</i>	code field address
<i>char</i>	ASCII character code

---

ASCII Collating Sequence: ! " # \$ % & ' ( ) \* + , - . / digits ; < = > ? @ ALPHA [ \ ] ^ \_ ` alpha { | } ~

Term/Abbreviation	Meaning
<i>count</i>	count ( length )
<i>d, d<sub>1</sub>, d<sub>2</sub>, ...</i>	signed double-precision numbers (2 cells each)
<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>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 numbers (4 cells each)
<i>lfa</i>	link field address
<i>n, n<sub>1</sub>, n<sub>2</sub>, ...</i>	signed single-precision numbers
<i>nfa</i>	name field address
<i>pfa</i>	parameter field address
<i>row</i>	row position
<i>rem</i>	remainder
<i>blk</i>	block number
<i>spr</i>	sprite number
<i>str</i>	string address
<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 (2 cells)
<i>vaddr</i>	VDP address

## D.2 fbForth 2.0 Word Descriptions

<b>!</b>		Resident
	( <i>n addr ---</i> )	
	Stores 16 bit-number <i>n</i> at address. Pronounced “store”.	
<b>!CSP</b>		Resident
	( --- )	
	Saves the stack position in user variable <b>CSP</b> . Used as part of compiler security.	
<b>#</b>		Resident
	( <i>d<sub>1</sub> --- d<sub>2</sub></i> )	
	Converts the rightmost digit of a double number <i>d<sub>1</sub></i> to an ASCII character, which is placed in a pictured numeric output string built downward from <b>PAD</b> to <b>HERE</b> . The digit to convert is the remainder from division of <i>d<sub>1</sub></i> by the current radix contained in <b>BASE</b> . The quotient <i>d<sub>2</sub></i> is maintained for further processing. Used between <b>&lt;#</b> and <b>#&gt;</b> . See <b>#S</b> , <b>&lt;#</b> and <b>#&gt;</b> . The details of pictured numeric output are shown at <b>&lt;#</b> .	

- #>** Resident  
 ( *d* --- *addr count* )  
 Terminates pictured numeric output conversion by dropping *d* and leaving the text address and character count suitable for **TYPE** , *q.v.* The details of pictured numeric output are shown at **<#** .
- #MOTION** Resident  
 ( *n* --- )  
 Sets sprite numbers 0 to *n* – 1 in automotion.
- #S** Resident  
 ( *d*<sub>1</sub> --- *d*<sub>2</sub> )  
 Generates pictured numeric output as ASCII text at **PAD** from *d*<sub>1</sub> by executing **#** until a zero double number *d*<sub>2</sub> results. Used between **<#** and **#>** , *q.v.* The details of pictured numeric output are shown at **<#** .
- \$"** Stack-based String Library [42]  
 ( IS:*string*" --- ) ( SS: --- *str* ) “string to string stack”  
 The word  **\$"** takes a string from the input stream and pushes it to the string stack. The end of the string is indicated by a quotation mark. For example,  
 **\$" Hello, World!" ok:0**  
 In this example the string “Hello, world!” is pushed directly to the string stack, thus becoming the top item on the string stack.  
 **\$"** is a state-smart word. It can be used in both colon definitions and also directly at the command line. The correct action will be taken in either case.  
 In order that the runtime actions of  **\$"** may be compiled into a definition if so desired, the runtime action of this word is encapsulated within the word ( **\$"**) . Therefore, if the runtime behavior of this word is to be compiled into another word, one must compile the word ( **\$"**) .
- \$.** More Useful Stack Words etc. [41]  
 ( *n* --- )  
 Display the top number on the stack as an unsigned hexadecimal number.
- \$.S** Stack-based String Library [42]  
 ( --- ) ( SS: --- )  
 The word  **\$.S** displays a non-destructive string stack dump to the output device. The length of each string is given, along with the total number of strings on the string stack. The amount of space allocated to the string stack, the amount of space in use and the amount of free space is also reported. An example appears above under the description of  **REPLACE\$** .

**\$CONST**

Stack-based String Library [42]

( *max\_len* IS:*name* --- ) “string constant”RUNTIME: ( --- *\$Caddr* )

The word **\$CONST** declares a string constant. Declared at compile time, string constants require a maximum length and a name. For example,

```
50 $CONST WELCOME ok:0
```

The above example declares a string with a maximum size of 50 characters. It shall be referenced in code using the name **WELCOME**.

A string constant pushes the address of its maximum length field which can be read with the word **MAXLEN\$**.

' *[immediate word]* Resident

( --- *pfa* )

Used in the form:

' **nnnn**

Searches the dictionary for **nnnn** and, if found, leaves the parameter field address *pfa* of the word. As a compiler directive, ' , because it is an immediate word, executes in a colon definition to compile the address of a literal, *viz.*, the *pfa* of the found word. If the word is not found after a search of **CONTEXT** and **CURRENT**, the word is displayed followed by '?' to indicate the error. The stack is then cleared, the contents of **IN** and **BLK** are left on the stack and **QUIT** is called. Pronounced “tick”.

( *[immediate word]* Resident

( --- )

( is used in the form:

( **cccc** )

It starts a comment that will not be compiled if it occurs in a definition. It causes the interpreter to consume characters from the input stream until a ')' is found or the end of the input stream (block or TIB) is reached. May occur during execution or in a colon definition. A blank after the leading parenthesis is required. This is most useful for commenting Forth source code in blocks.

(+LOOP) Resident

( *n* --- )

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

(. ") Resident

( --- )

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

- (;CODE)** Resident
- ( --- )  
 The runtime procedure, compiled by **DOES>ASM:** and **;CODE** , that rewrites the code field of the most recently defined word to point to the machine code sequence following **DOES>ASM:** or **;CODE** . See **DOES>ASM:** and **;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. It is defined as
- : (ABORT) ABORT ;**
- If you wished to redefine it to execute your error procedure, say **MY\_ERROR\_PROC** , you would replace **ABORT** with **MY\_ERROR\_PROC** as shown in the redefinition of **(ABORT)** below:
- : (ABORT) MY\_ERROR\_PROC ;**
- (DO)** Resident
- ( --- )  
 The runtime procedure compiled by **DO** , which moves the loop control parameters to the return stack. See **DO** .
- (DOES>)** Resident
- ( --- )  
 The runtime procedure compiled by **DOES>** .
- (FIND)** Resident
- ( *addr nfa* --- *false* | *pfa b true* )  
 Searches the dictionary starting at the name field address *nfa*, looking for a match to the text at *addr*. The addresses, *addr* and *nfa*, both point to the length byte of packed character strings (see footnote 4 on page 19). Returns the parameter field address *pfa*, length byte *b* of name field, and *true* for a match. If no match is found, only *false* is left. [Note: See Chapter 12 about the length byte of a name field.]
- (LINE)** Resident
- ( *n blk* --- *addr count* )  
 Converts the line number *n* and the Forth block number *blk* to the disk buffer address *addr* containing the data and the number *count* of characters. If the block is not in a block buffer, it is loaded from the current blocks file. If *count* is 64, the full-line text length of the block is indicated.
- (LOOP)** Resident
- ( --- )  
 The runtime procedure compiled by **LOOP** , which increments the loop index and tests for loop completion. See **LOOP** .

<b>(NUMBER)</b>	Resident
$( d_1 \text{ addr}_1 \text{ --- } d_2 \text{ addr}_2 )$	
The double number $d_1$ should be 0, <i>i.e.</i> , the stack should contain two 16-bit zeroes. The address $\text{addr}_1$ must point to the packed character string of the ASCII text to be converted to a double number, which will be left as $d_2$ . The conversion begins at $\text{addr}_1 + 1$ with respect to the current radix in <b>BASE</b> . The new value is accumulated with double number $d_1 = 0$ as the initial value. If a decimal point is encountered in the string, <b>DPL</b> is updated with the number of digits to the right of the decimal point. The address of the first unconvertible digit is $\text{addr}_2$ . <b>(NUMBER)</b> is used by <b>NUMBER</b> .	
<b>(OF)</b>	Resident
$( \text{---} )$	
The run time procedure compiled by <b>OF</b> .	
<b>(UB)</b>	Resident
$( \text{addr ---} )$	
Runtime routine compiled or executed by <b>USEBFL</b> that changes the current blocks file to the filename as a packed character string (see footnote 4 on page 19) pointed to by $\text{addr}$ .	
<b>*</b>	Resident
$( n_1 \ n_2 \text{ --- } n_3 )$	
Leaves the signed product of two signed numbers.	
<b>*/</b>	Resident
$( n_1 \ n_2 \ n_3 \text{ --- } \text{quot} )$	
Leaves the quotient <i>quot</i> of $(n_1 * n_2) / n_3$ , where all are signed numbers. Retention of an intermediate signed 32-bit product permits greater accuracy than would be available with the sequence :	
$n_1 \ n_2 \ * \ n_3 \ /$	
<b>*/MOD</b>	Resident
$( n_1 \ n_2 \ n_3 \text{ --- } \text{rem} \ \text{quot} )$	
Leaves the quotient <i>quot</i> and remainder <i>rem</i> of the operation $(n_1 * n_2) / n_3$ . An intermediate signed 32-bit product is used as for <b>*/</b> . In fact, <b>*/MOD</b> is used by <b>*/</b> .	
<b>+</b>	Resident
$( n_1 \ n_2 \text{ --- } n_3 )$	
Leaves the sum of $n_1 + n_2$ as $n_3$ .	
<b>+!</b>	Resident
$( n \ \text{addr} \text{ ---} )$	
Adds $n$ to the value at the address. Pronounced “plus store”.	

**+\$** Stack-based String Library [42]

( --- ) ( SS:  $str_1 str_2 - str_1 \& str_2$  ) “concatenate strings”

The word **+\$** replaces the top two strings on the string stack with their concatenated equivalent. For example,

```
$" red" "$" blue" +$ ok:0
```

At this point, “red” and “blue” have been removed from the string stack. The topmost string on the string stack now has the value “redblue”. Note that the topmost string goes to the right of the newly concatenated string.

**+ -** Resident

(  $n_1 n_2 --- n_3$  )

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

**+BUF** Resident

(  $addr_1 --- addr_2 flag$  )

Advance the disk buffer address  $addr_1$  to the address of the next buffer  $addr_2$ . Boolean flag is false when  $addr_2$  is the buffer presently pointed to by user variable **PREV**.

**+LOOP** Resident

*[immediate word]*

Used in a colon definition in the form:

```
DO ... n +LOOP
```

Compile time: (  $addr\ 3 ---$  )

**+LOOP** compiles the runtime word (**+LOOP**) and the branch offset computed from **HERE** to the address  $addr$  left on the stack by **DO**. The value 3 is used for compile-time error checking.

Runtime: (  $n ---$  )

**+LOOP** selectively controls branching back to the corresponding **DO** based on  $n$ , the loop index and the loop limit. The signed increment  $n$  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 ( $n > 0$ ), or until the new index is equal to or less than the limit ( $n < 0$ ). Upon exiting the loop, the parameters are discarded and execution continues ahead.

**,** Resident

(  $n ---$  )

Store  $n$  into the next available dictionary memory cell, advancing the dictionary pointer. Pronounced “comma”.

**-** Resident

(  $n_1 n_2 --- n_3$  )

Leave the difference  $n_3$  of  $n_1 - n_2$ .

<b>--&gt;</b>	<i>[immediate word]</i>	Resident
	( --- )	
	Continues interpretation with the next Forth block in the current blocks file. --> can only be used while loading blocks. Pronounced “next block”.	
<b>-DUP</b>		Resident
	( $n_1$ --- $n_1$   $n_1$ $n_1$ )	
	Duplicate $n_1$ 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 a <b>DUP</b> ed 0.	
<b>-FIND</b>		Resident
	( --- <i>false</i>   <i>pfa len true</i> )	
	Accepts the next text word (delimited by blanks) in the input stream to <b>HERE</b> as a packed character string (see footnote 4 on page 19), 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>len</i> and <i>true</i> are left. Otherwise, only <i>false</i> is left. [Note: See Chapter 12 about the length byte.]	
<b>-ROT</b>		More Useful Stack Words etc. [41]
	( $n_1$ $n_2$ $n_3$ --- $n_3$ $n_1$ $n_2$ )	
	Rotate right the top three numbers on the stack, resulting in the top number on the bottom.	
<b>-ROT\$</b>		Stack-based String Library [42]
	( --- ) ( SS: $str_1$ $str_2$ $str_3$ --- $str_3$ $str_1$ $str_2$ ) “rotate strings”	
	The word <b>-ROT\$</b> rotates the top three strings to the right. The top string prior to the execution of <b>-ROT\$</b> moves to the third position. See Chapter 14 for implementation details regarding stack space limitations.	
<b>-TRAILING</b>		Resident
	( $addr$ $n_1$ --- $addr$ $n_2$ )	
	Adjusts the character count $n_1$ of a character string at <i>addr</i> to suppress the output of trailing blanks by <b>TYPE</b> , <i>i.e.</i> , the characters at $addr + n_2$ to $addr + n_1$ are blanks. If the character string is a packed character string (see footnote on page ), <i>addr</i> points to the first character after the length byte. <b>-TRAILING</b> starts at the last character and steps to the beginning of the string as it looks for trailing blanks, decrementing $n_1$ until a non-blank character is encountered. At that point, $n_1$ is replaced with $n_2$ . The output parameters of <b>COUNT</b> are suitable input parameters for <b>-TRAILING</b> .	
<b>.</b>		Resident
	( $n$ --- )	
	Prints a number from a signed 16-bit two’s complement value $n$ , converted according to the numeric base stored in <b>BASE</b> . A trailing blank follows. Pronounced “dot”.	

<b>."</b>	<i>[immediate word]</i>	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>.\$</b>		Stack-based String Library [42]
	( --- ) ( SS: <i>str</i> --- ) "display string" The word <b>.\$</b> pops the topmost string from the string stack and displays it. For example, <b>\$" Hello, World!" . \$ <u>Hello, World!</u> <u>ok:0</u></b>	
<b>.\$CONST</b>		Stack-based String Library [42]
	( <i>\$Caddr</i> --- ) "display string constant" Given the address of a string constant on the data stack the word <b>.\$CONST</b> shall display the string. For example, <b>50 \$CONST WELCOME <u>ok:0</u></b> <b>WELCOME := " Hello and welcome!" <u>ok:0</u></b> <b>WELCOME . \$CONST CR</b> <b><u>Hello and welcome!</u> <u>ok:0</u></b>	
<b>.BASE</b>		Resident
	( --- <i>n</i> ) Print the decimal value <i>n</i> of the current radix (number base).	
<b>.LINE</b>		Resident
	( <i>n blk</i> --- ) Print on the terminal device a line of text from the current blocks file corresponding to the line number <i>n</i> of block number <i>blk</i> . Trailing blanks are suppressed.	
<b>.R</b>		Resident
	( <i>n<sub>1</sub> n<sub>2</sub></i> --- ) Prints 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>		Resident
	( --- ) Prints the entire contents of the parameter stack as unsigned numbers in the current <b>BASE</b> . The bottom of the stack is shown by an initial '['.	
<b>/</b>		Resident
	( <i>n<sub>1</sub> n<sub>2</sub></i> --- <i>n<sub>3</sub></i> ) Leaves the quotient <i>n<sub>3</sub></i> of <i>n<sub>1</sub> / n<sub>2</sub></i> .	

<b>/MOD</b>	<p>( <math>n_1</math> <math>n_2</math> --- <i>rem quot</i> )</p> <p>Leaves the remainder <i>rem</i> and signed quotient <i>quot</i> of <math>n_1 / n_2</math>. The remainder has the sign of the dividend.</p>	Resident
<b>0 1 2 3</b>	<p>( --- <math>n</math> )</p> <p>These small numbers are used so often that it is useful to define them by name in the dictionary as constants. Doing so saves compile time because the interpreter searches the dictionary for a match before it decides whether it is a number. Also, numbers, otherwise, require two extra bytes of dictionary storage when used in definitions.</p>	Resident
<b>0&lt;</b>	<p>( <math>n</math> --- <i>flag</i> )</p> <p>Leaves a true flag if the number <math>n</math> is less than zero (negative). Otherwise, <b>0&lt;</b> leaves a false flag.</p>	Resident
<b>0=</b>	<p>( <math>n</math> --- <i>flag</i> )</p> <p>Leaves a true flag if the number is equal to zero. Otherwise, <b>0=</b> leaves a false flag.</p>	Resident
<b>0&gt;</b>	<p>( <math>n</math> --- <i>flag</i> )</p> <p>Leaves a true flag if the number is greater than zero (positive). Otherwise, <b>0&gt;</b> leaves a false flag.</p>	Resident
<b>0BRANCH</b>	<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> , <b>END</b> and <b>WHILE</b> .</p>	Resident
<b>1+</b>	<p>( <math>n_1</math> --- <math>n_2</math> )</p> <p>Increments <math>n_1</math> by 1.</p>	Resident
<b>1-</b>	<p>( <math>n_1</math> --- <math>n_2</math> )</p> <p>Decrements <math>n_1</math> by 1.</p>	Resident
<b>2+</b>	<p>( <math>n_1</math> --- <math>n_2</math> )</p> <p>Leaves <math>n_1</math> incremented by 2 as <math>n_2</math>.</p>	Resident

**2-** Resident

(  $n_1$  ---  $n_2$  )

Leaves  $n_1$  decremented by 2 as  $n_2$ .

**2DROP** More Useful Stack Words etc. [41]

(  $n_1$   $n_2$  --- )

Drop the top two numbers from the stack.

**2DUP** More Useful Stack Words etc. [41]

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

Duplicate the top two numbers on the stack.

**:** *[immediate word]* Resident

( --- )

Used in the form, called a colon definition:

```
 : cccc ... ;
```

Creates a dictionary entry defining **cccc** as equivalent to the sequence of Forth word definitions in ‘...’ until the next **;**, **DOES>ASM:** or **;CODE**. The compiling process is done by the text interpreter as long as **STATE** is non-zero. Other details are that the **CONTEXT** vocabulary is set to the **CURRENT** vocabulary and that words with the precedence bit (see § 12.2 “Name Field”) set are executed rather than being compiled.

If you wish to **FORGET** an unfinished definition, the word likely will not be found. If it is the last definition attempted, you can make it findable by executing **SMUDGE** and then **FORGET**ting it.

**:** (*traceable*) *[immediate word]* TRACE — Colon Definition Tracing [23]

( --- )

This is an alternate definition of **:** that adds the capability to colon definitions of being traced when they are executed. When a colon definition is compiled under the **TRACE** option, tracing output may be turned on with **TRON** and off with **TROFF** prior to executing the word so defined. After **TRON** is executed, each time the word is executed its name will be output along with the contents of the stack. See **TRACE**, **UNTRACE**, **TRON** and **TROFF**.

**:=** Stack-based String Library [42]

( *\$Caddr* IS:string" --- ) “assign string constant”

Given the address of a string constant on the data stack, the word **:=** initializes the string constant with the string from the input stream. For example,

```
50 $CONST WELCOME ok:0  
WELCOME := " Hello and welcome!" ok:0
```

<b>;</b>	<i>[immediate word]</i>	Resident
	( --- )	
	Terminates a colon definition and stops further compilation. Compiles the runtime <b>;S</b> .	
<b>;ASM</b>		Resident
	( --- )	
	Synonym for Assembler word <b>NEXT</b> , . <b>;ASM</b> should be paired with <b>ASM:</b> to clearly surround assembly code or machine code:	
	<b>ASM: cccc &lt;assembly mnemonics&gt; ;ASM</b>	
	<b>;ASM</b> puts <b>045Fh</b> (machine code for ALC: <b>B *R15</b> ) at <b>HERE</b> and advances <b>HERE</b> . See Chapter 9 “The fbForth 2.0 TMS9900 Assembler” for details. See also <b>ASM:</b> , <b>NEXT</b> , and <b>CODE</b> for more information.	
<b>;CODE</b>	<i>[immediate word]</i>	Resident
	( --- )	
	Used with <b>&lt;BUILDS</b> in the form:	
	<b>: cccc &lt;BUILDS ... ;CODE &lt;assembly mnemonics&gt; NEXT,</b>	
	Stops compilation and terminates a new defining word, <b>cccc</b> , by compiling <b>(;CODE)</b> . Sets the <b>CONTEXT</b> vocabulary to <b>ASSEMBLER</b> , assembling to machine code the assembly mnemonics following <b>;CODE</b> .	
	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>(;CODE)</b> in the definition of <b>cccc</b> , <i>i.e.</i> , when <b>nnnn</b> is executed, it does so by jumping to that code in <b>cccc</b> . An existing defining word ( <b>&lt;BUILDS</b> in this case) must exist in <b>cccc</b> prior to <b>;CODE</b> . See Chapter 9 “The fbForth 2.0 TMS9900 Assembler” for more details.	
<b>;S</b>		Resident
	( --- )	
	Stops interpretation of a Forth block. <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> )	
	Leaves a true flag if $n_1$ is less than $n_2$ . Otherwise, <b>&lt;</b> leaves a false flag.	
<b>&lt;#</b>		Resident
	( --- )	
	Sets up for pictured numeric output formatting using the words, <b>&lt;#</b> , <b>#</b> , <b>HOLD</b> , <b>#S</b> , <b>SIGN</b> and <b>#&gt;</b> . <b>&lt;#</b> initializes <b>HLD</b> with <b>PAD</b> . <b>HLD</b> is decremented by <b>#</b> via <b>HOLD</b> for each successive digit converted. A few format examples follow:	

**<# #S #>** converts all digits.  
**<# #S SIGN #>** converts all digits with a preceding sign.  
**<# # # #S #>** converts at least 3 digits with leading zeroes.  
**<# # # 46 HOLD #S #>** converts all digits with a dot before last 2 digits.

Though **<#** requires no input parameters, you should provide the parameters on the stack that are required by all of the formatting words between **<#** and **#>**. At the very least, this is the double number you wish to convert. **DABS** should usually be executed prior to **<#** because **<# ... #>** will not properly convert negative numbers. If you wish to include a sign in the output, a signed number should be pushed to the stack before the double number to be converted.

The conversion is done on a 31-bit (positive) double number producing text at **PAD** (working downward toward **HERE**), eventually suitable for output by **TYPE**. The picture template between **<#** and **#>** represents the output picture from right to left, *i.e.*, the rightmost digit is processed first. The following is an example of generalized output from a double number on the stack that may be positive or negative:

**SWAP OVER DABS <# #S SIGN #> TYPE**

In the example above, **SWAP** puts the high-order cell, which contains the sign bit, on the bottom; **OVER** copies it back to its proper place on top, leaving 3 cells (*n d*) on the stack; and **DABS** forces *d* positive. This arrangement is what is expected by **SIGN**.

*Important note:* You should not execute words that change **HERE** or **PAD** until after you have finished formatting the number and retrieving the converted output.

See **#**, **#S**, **SIGN**, **#>**, **HLD** and **HOLD** for more information.

**<>** More Useful Stack Words etc. [41]

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

Result *flag* is true (1) if *n*<sub>1</sub> ≠ *n*<sub>2</sub> and false (0) otherwise.

**<BUILDS** Resident

( --- )

It is used within a colon-definition to build a new defining word:

**: cccc <BUILDS ... DOES> ... ;** or  
**: cccc <BUILDS ... ;CODE ... NEXT,** or clearer equivalent  
**: cccc <BUILDS ... DOES>ASM: ... ;ASM**

Each time **cccc** is executed, **<BUILDS** defines a new word with a high-level (**DOES>**) or machine-code (**;CODE** or **DOES>ASM:**) execution procedure. Executing **cccc** in the form:

**cccc nnnn**

uses **<BUILDS** to create a dictionary entry for **nnnn**. For the definition with **DOES>**, when **nnnn** is later executed, it has the address of its parameter area on the stack and executes the words after **DOES>** in **cccc**. For the definition with **DOES>ASM:**, when **nnnn** is later executed, it executes the words after **DOES>ASM:** in **cccc**. **<BUILDS**

allows runtime procedures to be written in high-level code with **DOES>** or in assembler code with **;CODE** or **DOES>ASM:**. See **DOES>ASM:** for equivalence with **;CODE**.

**<BUILDS** is simply defined as

```
      : <BUILDS CREATE SMUDGE ;
```

<b>&lt;CLOAD&gt;</b>	Resident
( --- )	
The runtime procedure compiled by <b>CLOAD</b> .	
<b>=</b>	Resident
( $n_1$ $n_2$ --- <i>flag</i> )	
Leaves a true flag if $n_1 = n_2$ . Otherwise, it leaves a false flag.	
<b>=CELLS</b>	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.	
<b>&gt;</b>	Resident
( $n_1$ $n_2$ --- <i>flag</i> )	
Leaves a true flag if $n_1 > n_2$ . Otherwise, it leaves a false flag.	
<b>&gt;\$</b>	Stack-based String Library [42]
( $\$Caddr$ --- ) ( SS: --- <i>str</i> ) “to string stack”	
Given the address of a string constant on the data stack, the word <b>&gt;\$</b> copies the contents of the string to the string stack where it can be manipulated. For example,	
<pre> 50 \$CONST WELCOME ok:0 WELCOME := " Hello and welcome!" ok:0 WELCOME &gt;\$ ok:0 </pre>	
<b>&gt;\$CONST</b>	Stack-based String Library [42]
( $\$Caddr$ --- ) ( SS: <i>str</i> --- ) “to string constant”	
The word <b>&gt;\$CONST</b> takes the topmost string from the string stack and moves it into the string constant whose address is on the data stack. For example,	
<pre> 4 \$CONST COLOR ok:0 \$" red" COLOR &gt;\$CONST ok:0 </pre>	
At this point, the string constant <b>COLOR</b> has the value “red”. To verify, display the string using <b>.\$CONST</b> as follows:	
<pre> COLOR .\$CONST red ok:0 </pre>	

<b>&gt;DEG</b>	<p><math>(f_1 \text{ --- } f_2)</math></p> <p>Converts an 8-byte floating point number <math>f_1</math> from radians to <math>f_2</math> degrees.</p>	Resident
<b>&gt;F</b>	<p><i>[immediate word]</i></p> <p><math>( \text{---} f )</math></p> <p>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.</p>	Resident
<b>&gt;R</b>	<p><math>( n \text{ ---} ) ( R: \text{---} n )</math></p> <p>Removes 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.</p>	Resident
<b>&gt;RAD</b>	<p><math>(f_1 \text{ ---} f_2)</math></p> <p>Converts an 8-byte floating point number <math>f_1</math> from degrees to <math>f_2</math> radians.</p>	Resident
<b>?</b>	<p><math>( addr \text{ ---} )</math></p> <p>Prints the value contained at address <i>addr</i> in free format according to the current radix stored in <b>BASE</b> . This word is short for the two words, <b>@ . .</b></p>	Resident
<b>?COMP</b>	<p><math>( \text{---} )</math></p> <p>This word is typically used in the definitions of compile-only words to insure the word containing it is being used in a definition. When <b>?COMP</b> is executed in other than compile mode, it displays the word just interpreted with a '?', issues the error message, "compilation only", clears the stack, leaves the contents of <b>IN</b> and <b>BLK</b> and executes <b>QUIT</b> , e.g.,</p> <p style="padding-left: 40px;"><b>9 0 DO I . LOOP DO ? <u>compilation only</u></b></p> <p>Though <b>LOOP</b> is also a compile-only word, <b>DO</b> is the first one encountered and the one that triggers the above error.</p>	Resident
<b>?CSP</b>	<p><math>( \text{---} )</math></p> <p>This word is used in the definitions of <b>;</b> , <b>;</b> <b>CODE</b> and <b>DOES&gt;ASM:</b> to insure that the stack position at the end of the definition is at the same height as when it was started with <b>:</b> , which stores the stack pointer in <b>CSP</b> . The error condition typically occurs with unbalanced conditionals. Whichever terminating word tested the stack height will be displayed followed by a '?' and "definition not finished", e.g.,</p> <p style="padding-left: 40px;"><b>: XXXX IF ; ; ? <u>definition not finished</u></b></p>	Resident

<b>?ERROR</b>	Resident
( <i>flag n ---</i> )	
Issues an error message corresponding to error number <i>n</i> if the Boolean flag is true. <b>?ERROR</b> is the word that all the error-checking words in <b>fbForth 2.0</b> execute to actually check for an error and to display the error message. It is defined as	
<b>: ?ERROR SWAP IF ERROR ELSE DROP THEN ;</b>	
<b>?EXEC</b>	Resident
( --- )	
This word is used in the definitions of <b>:</b> , <b>CODE</b> , <b>ASM:</b> and most of the words in the <b>ASSEMBLER</b> vocabulary to insure those words are executing and not being used in a definition. <b>?EXEC</b> issues the error message, “execution only”, as in	
<b>: XXXX : ... ; : ? <u>execution only</u></b>	
<b>?FLERR</b>	Resident
( --- )	
Determines if the most recently executed floating-point (FP) operation resulted in an error. This word will give valid information any time before executing another FP operation clears the FP error location at <b>8354h</b> . <b>?FLERR</b> issues the error message, “floating point error”, upon finding an error. The nature of the floating-point error may be ascertained by executing <b>FLERR</b> , <i>q.v.</i> , to get the FP error number and cross-referencing the code in the error table in § 7.14 “Floating Point Error Codes”.	
<b>?KEY</b>	Resident
( --- <i>char</i> )	
Scans the keyboard for input. If no key is pressed, a 0 is left on the stack. Otherwise, the 7-bit ASCII code of the key pressed is left on the stack.	
<b>?KEY8</b>	Resident
( --- <i>n</i> )	
Scans the keyboard for input. If no key is pressed, a 0 is left on the stack. Otherwise, the 8-bit code of the key pressed is left on the stack.	
<b>?LOADING</b>	Resident
( --- )	
This word is used in the definition of <b>--&gt;</b> to insure that <b>fbForth 2.0</b> is loading from the current blocks file rather than executing on the command line. <b>?LOADING</b> issues error message, “use only when loading”, if not loading as in	
<b>--&gt; --&gt; ? <u>use only when loading</u></b>	
<b>?PAIRS</b>	Resident
( <i>n<sub>1</sub> n<sub>2</sub> ---</i> )	
Issue the error message, “conditionals not paired”, if <i>n<sub>1</sub></i> does not equal <i>n<sub>2</sub></i> . The message indicates that compiled conditionals do not match, such as when a <b>DO</b> has been left without a <b>LOOP</b> , an <b>IF</b> has no corresponding <b>ENDIF</b> or <b>THEN</b> , <i>etc.</i>	

**?STACK** Resident

( --- )

**INTERPRET** uses **?STACK** to check whether the parameter stack is out of bounds after processing a word or number. If the top of the stack is lower than its base, “empty stack” will be displayed. If the stack has run into the output buffer at **PAD** in the other direction, “full stack” will be displayed. **?STACK** is defined as

```

: ?STACK
  SP@ S0 @ SWAP U< 1 ?ERROR
  SP@ HERE 128 + U< 7 ?ERROR ;

```

**?TERMINAL** Resident

( --- *flag* )

Scans the terminal keyboard for actuation of the break key (<**BREAK**>). A true flag indicates actuation. On the TI-99/4A, <**FCTN+4**>, <**BREAK**> and <**CLEAR**> are all the same key.

**@** Resident

( *addr* --- *n* )

Leave the 16-bit contents *n* of *addr*.

**A\$M** TMS9900 Assembler [21]

( --- )

This word is compiled into the **FORTH** vocabulary and marks the end of the **ASSEMBLER** vocabulary. It is used by **CLOAD** to determine whether the TMS9900 Assembler has been loaded.

**ABORT** Resident

( --- )

**ABORT** is **fbForth 2.0**'s warm start. It clears the stacks, sets both **CONTEXT** and **CURRENT** to the **FORTH** vocabulary, enters the execution state and, after printing “**fbForth 2.0**”, executes **INTERPRET** to get user input from the terminal.

**ABS** Resident

(  $n_1$  ---  $n_2$  )

Leaves the absolute value of  $n_1$  as  $n_2$ .

**AGAIN** *[immediate word]* Resident

Used in a colon definition in the form:

```

BEGIN ... AGAIN

```

Compile time: ( *addr* 1 --- )

**AGAIN** compiles **BRANCH** with an offset from **HERE** to *addr*, which it copies to the space reserved for it at *addr*. The value 1 is used for compile-time error checking.

Runtime: ( --- )

**AGAIN** forces execution to return to the corresponding **BEGIN** . There is no effect on the stack. Execution cannot leave the loop unless **R> DROP** is executed one level below by some word in the loop.

**ALLOT** Resident

( *n* --- )

Adds the signed number *n* to the dictionary pointer **DP** , which moves **HERE** by *n* bytes. It has the effect of reserving *n* bytes of dictionary space if it is positive and moving **HERE** backwards to reclaim memory if it is negative (*be careful!*).

**ALTIN** Resident

( --- *addr* )

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

**ALTOUT** Resident

( --- *addr* )

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

**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** Resident

( --- )

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

**ASM:** Resident

( --- )

Synonym for **CODE** intended to be paired with **;ASM** , a synonym for **NEXT** , . It is used as follows:

**ASM: NEW-WORD <assembly mnemonics> ;ASM**

See Chapter 9 The fbForth 2.0 TMS9900 Assembler for details. See also **;ASM** , **CODE** and **NEXT** , .

**ASSEMBLER** *[immediate word]* Resident

( --- )

The name of the **fbForth 2.0** Assembler vocabulary. Execution makes **ASSEMBLER** the **CONTEXT** vocabulary. Because **ASSEMBLER** is immediate, it will execute during the creation of a colon definition to select this vocabulary at compile time. See **VOCABULARY** .

<b>ATN</b>	Resident
$( f_1 \text{ --- } f_2 )$	
Calculates the arctangent in radians of $f_1$ leaving the floating point result $f_2$ on the stack.	
<b>B/BUF</b>	Resident
$( \text{--- } 1024 )$	
This constant leaves the number of bytes $n$ per disk buffer (always 1024 in <b>fbForth 2.0</b> ), the byte count read from the current blocks file by <b>BLOCK</b> . It is included for backward compatibility with TI Forth	
<b>B/SCR</b>	Resident
$( \text{--- } 1 )$	
This constant always leaves 1 on the stack. It is included for backward compatibility with TI Forth, where it is the number of blocks per editing screen. By convention, an editing screen is 1024 bytes organized as 16 lines of 64 characters each.	
<b>BACK</b>	Resident
$( addr \text{ --- } )$	
Calculates the backward branch offset from <b>HERE</b> to $addr$ and compile into the next available dictionary memory address. Used by <b>LOOP</b> , <b>+LOOP</b> , <b>UNTIL</b> and <b>AGAIN</b> to calculate the distance back to the beginning of the loop.	
<b>BANK@</b>	Resident
$( addr \ n_1 \text{ --- } n_2 )$	
Returns on the stack the contents $n_2$ of the cell at address $addr$ in bank $n_1$ . If the bank number does not exist, the returned value will be from bank 0. If the address is in CPU RAM space, the returned value will be from there.	
<b>BANKC@</b>	Resident
$( addr \ n \text{ --- } b )$	
Returns on the stack the contents $b$ of the byte at address $addr$ in bank $n$ . See <b>BANK@</b> .	
<b>BASE</b>	Resident
$( \text{--- } addr )$	
A user variable containing the current radix or number base used for input and output conversion.	
<b>BASE-&gt;R</b>	Resident
$( \text{---} )$	
Places the current radix on the return stack. Caution must be exercised when using <b>BASE-&gt;R</b> and <b>R-&gt;BASE</b> with <b>CLOAD</b> as these will cause the return stack to be polluted if a <b>LOAD</b> is aborted and the <b>BASE-&gt;R</b> is not balanced by a <b>R-&gt;BASE</b> at execution time. See <b>R-&gt;BASE</b> .	

<b>BEEP</b>	<p>( --- )</p> <p>Produces the sound associated with correct input or prompting.</p>	Resident
<b>BEGIN</b>	<p><i>[immediate word]</i></p> <p>Occurs in a colon-definition in the form:</p> <p style="padding-left: 40px;"><b>BEGIN ... UNTIL</b> or <b>BEGIN ... END</b>  <b>BEGIN ... AGAIN</b>  <b>BEGIN ... WHILE ... REPEAT</b></p> <p>Compile time: ( --- <i>addr</i> 1 )</p> <p><b>BEGIN</b> leaves its return address <i>addr</i> for branching calculation and storage by <b>UNTIL</b> , <b>END</b> , <b>AGAIN</b> and <b>REPEAT</b> and a 1 for compiler error checking.</p> <p>Runtime: ( --- )</p> <p><b>BEGIN</b> marks the start of a sequence that may be repetitively executed. It serves as a return point from the corresponding <b>UNTIL</b> , <b>AGAIN</b> or <b>REPEAT</b> . When executing <b>UNTIL</b> , a return to <b>BEGIN</b> will occur if the top of the stack is false; for <b>AGAIN</b> and <b>REPEAT</b> a return to <b>BEGIN</b> always occurs.</p>	Resident
<b>BFLNAM</b>	<p>( <i>flag</i> --- [ ]   <i>addr</i> )</p> <p>Helper routine that gets a blocks filename from the input stream into PAD or HERE and passes a name pointer (<i>addr</i>) if <i>flag</i> is true (used on command line), but passes nothing if <i>flag</i> is false (<i>addr</i> is compiled by <b>SLIT</b> in a colon definition).</p>	Resident
<b>BL</b>	<p>( --- <i>char</i> )</p> <p>A constant that leaves the ASCII value 32 (<b>20h</b>) for “blank”.</p>	Resident
<b>BLANKS</b>	<p>( <i>addr count</i> --- )</p> <p>Fills an area of memory beginning at <i>addr</i> with <i>count</i> blanks.</p>	Resident
<b>BLK</b>	<p>( --- <i>addr</i> )</p> <p>A user variable containing the block number being interpreted. If zero, input is being taken from the terminal input buffer.</p>	Resident
<b>BLKRW</b>	<p>( [ <i>bfncaddr</i>   #<i>blks</i> <i>bfncaddr</i>   <i>bufaddr</i> <i>blk#</i> ] <i>opcode</i> --- <i>flag</i> )</p> <p>Blocks I/O utility routine called by <b>DO_BRW</b> . Addresses passed point to blocks file name (<i>bfncaddr</i>) and block RAM buffer (<i>bufaddr</i>). The number of items required on the stack depends on the opcode (passed by the corresponding command) as follows:</p> <p style="padding-left: 40px;">( <i>bfncaddr</i> -14 --- <i>flag</i> )    passed by <b>USEBFL</b>  ( #<i>blks</i> <i>bfncaddr</i> -16 --- <i>flag</i> )    passed by <b>MKBFL</b></p>	Resident

( *bufaddr blk# -18 --- flag* ) passed by **RBLK**  
 ( *bufaddr blk# -20 --- flag* ) passed by **WBLK**

**BLOAD** Resident

( *blk --- flag* )

Loads the binary image at *blk* 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.

**BLOCK** Resident

( *n --- addr* )

Leaves the memory address of the block buffer containing block *n*. If the block is not already in memory, it is transferred from the current blocks file to whichever buffer was least recently written. If the block occupying that buffer has been marked as updated, it is written to the current blocks file before block *n* is read into the buffer. See also **BUFFER** , **R/W** , **UPDATE** and **FLUSH** .

**BOOT** Resident

( --- )

Clears the stack, changes the radix to decimal, clears the error count, sets both **CURRENT** and **CONTEXT** to the Forth vocabulary, sets input stream to the terminal, makes the default blocks file (DSK1.FBLOCKS) current and loads block 1.

**BPB** Resident

( --- *vaddr* )

Gets the offset in VRAM from the **fbForth 2.0** record buffer (in **DISK\_BUF** ) for blocks file PABs from user variable **3Eh**, adds the offset to the contents of **DISK\_BUF** and pushes it to the stack.

**BPOFF** Resident

( --- *vaddr* )

Pushes to stack the VRAM address *vaddr* containing the offset of the current blocks file's PAB. This offset is used to manage blocks file PABs space, which has room for two PABs. This offset is toggled between 0 and 70 each time a new blocks file is made current.

**BRANCH** Resident

( --- )

The runtime procedure to unconditionally branch. An in-line offset is added to the interpretive pointer (IP) to branch ahead or back. **BRANCH** is compiled by **ELSE** , **AGAIN** , **REPEAT** , and **ENDOF** .

**BSAVE** Resident

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

Places a binary image (starting at *blk<sub>1</sub>* and going as far as necessary) of all dictionary contents between *addr* and **HERE** . The next available Forth block number *blk<sub>2</sub>* is returned on the stack. **BSAVE** empties all block buffers before saving the image

because the current blocks file may have changed. It is the user's responsibility to flush any dirty buffers before executing this command. Note that this is different behavior from TI Forth's **BSAVE**, which first flushes any dirty buffers. See **BLOAD**.

**BUFFER** Resident

( *n* --- *addr* )

Obtains the next memory buffer, assigning it to block *n*. 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.

**C!** Resident

( *b* *addr* --- )

Stores the low-order byte (8 bits) of *b* (16-bit number on the stack) at *addr*.

**C,** Resident

( *b* --- )

Stores the low-order byte (8 bits) of *b* (16-bit number on the stack) into the next available dictionary byte ( **HERE** ), advancing the dictionary pointer one byte. This instruction should be used with caution on computers with byte-addressing, word-oriented CPUs such as the TMS9900. If **HERE** is left at an odd address and the next operation stores a cell at **HERE**, the last byte will be overwritten. See **=CELLS**.

**C/L** Resident

( --- *n* )

Returns on the stack the number of characters per line (stored in **C/L\$**). The default value is 64 and usually represents the number of characters per line of a Forth block as it is edited (16 lines per 1024-byte block).

**C/L\$** Resident

( --- *addr* )

A user variable whose value is the number of characters per line. See **C/L**.

**C@** Resident

( *addr* --- *b* )

Leaves the 8-bit contents *b* of memory address *addr* on the stack.

**CASE** *[immediate word]* Resident

Used in a colon definition to initiate the construct:

```

CASE
  n1 OF ... ENDOF
  n2 OF ... ENDOF
  ...
ENDCASE

```

Compile time: ( --- *csp* 4 )

**CASE** gets the value *csp* of **CSP** to the stack for later restoration at the end of **ENDCASE**'s compile-time activity. It stores the current stack position in **CSP** to help **ENDCASE** track how many **OF ... ENDOF** branch distances to process. It finally pushes 4 to the stack for compile-time error checking by **OF** and **ENDCASE**.

RUNTIME: ( *n* --- *n* )

**CASE** itself does nothing with the number *n* on the stack; but, it must be there for **OF** or **ENDCASE** to consume. If  $n = n_1$ , the code between the immediately following **OF** and **ENDOF** is executed. Execution then continues after **ENDCASE**. If *n* does not match any of the values preceding any **OF**, the code between the last **ENDOF** and **ENDCASE** is executed and may use *n*; but, one cell *must* be left for **ENDCASE** to consume or a stack underflow will result. Execution then continues after **ENDCASE**.

**CAT** CAT -- Disk Catalog Utility [58]

( *n* --- )

**CAT** catalogs to the output device the disk number *n* on the stack for the current DSR. **CAT** reads the VIB, FDIR and each file's FDR to get its information (see Appendix K, "Diskette Format Details"). **CAT** will not load if **DIR** is loaded.

Usage: **2 CAT** to catalog DSK2.

**CEIL** Resident

( *f*<sub>1</sub> --- *f*<sub>2</sub> )

Finds the least integer *f*<sub>2</sub> (in floating point format) not less than the floating point number *f*<sub>1</sub>.

**CELLS** More Useful Stack Words etc. [41]

( *n* --- *2n* )

Replace *n* (a number of cells) with *2n* (the number of bytes in *n* cells).

**CFA** Resident

( *pfa* --- *cfa* )

Converts the parameter field address *pfa* of a definition to its code field address *cfa*.

**CHAR** Resident

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

Defines character # *char* to have the pattern specified by the 4 numbers (*n*<sub>1</sub>, *n*<sub>2</sub>, *n*<sub>3</sub>, *n*<sub>4</sub>) on the stack. The definition for character #0 by default resides at **800h**. Each character definition is 8 bytes long with each number on the stack representing two bytes.

**CHARPAT** Resident

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

Places the 4-cell (8-byte) pattern of a specified character *char* on the stack. By default, the definition for character #0 resides at **800h**.

- CLEAR** Resident  
( *blk* --- )  
Gets a block buffer for block# *blk*, fills it with blanks and marks it as updated.
- CLEN\$** Stack-based String Library [42]  
( *\$Caddr* --- *len* ) “string constant length”  
Given the address of a string constant on the data stack the word **CLEN\$** returns its actual length on the data stack. For example,  
**50 \$CONST WELCOME ok:0**  
**WELCOME := " Hello and welcome!" ok:0**  
**WELCOME CLEN\$ . 18 ok:0**
- CLINE** 64-Column Editor [6] Compact List [13]  
( *addr count n* --- )  
Prints one line of tiny characters on the display screen. **CLINE** expects on the stack the address *addr* of the line to be written in memory, the number of characters *count* in that line, and the line number *n* on which it is to be written on the display screen. **CLINE** calls **SMASH** to do the actual work. See **SMASH** and **CLIST** .
- CLIST** 64-Column Editor [6] Compact List [13]  
( *blk* --- )  
Lists the specified Forth block in tiny characters to the monitor. **CLIST** executes 16 calls to **CLINE** for the requisite 16 lines. See **CLINE** and **TCHAR** .
- CLOAD** *[immediate word]* Resident  
( *blk* --- )  
Used in the form:  
**blk CLOAD WWW**  
**CLOAD** will load Forth block *blk* only if the word **WWW** is not in the **CONTEXT** vocabulary. **WWW** should be the last word loaded when the series of blocks beginning with *blk* is loaded. A block number of 0 (*blk* = 0) will suppress loading of the current Forth block if the specified word has already been compiled.
- CLR\_BLKs** Resident  
( *blk<sub>1</sub> blk<sub>2</sub>* --- )  
**CLR\_BLKs** will **CLEAR** a range of blocks to blanks in the current blocks file. The blocks will be marked as updated (see **CLEAR** ).
- CLS** Resident  
( --- )  
Clears the display screen by filling the screen image table with blanks. The screen image table runs from **SCRN\_START** to **SCRN\_END** .

**CLSE** Resident

( --- )

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* --- )

Moves *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. This is *not* overlap safe for *addr<sub>1</sub>* < *addr<sub>2</sub>*.

**CMP\$** Stack-based String Library [42]

( --- -1|0|+1 ) ( SS: *str<sub>1</sub>* *str<sub>2</sub>* --- *str<sub>1</sub>* *str<sub>2</sub>* ) “compare strings”

The word **CMP\$** performs a case-sensitive comparison of the topmost two strings on the string stack and returns -1 if *str<sub>1</sub>* < *str<sub>2</sub>*, 0 if *str<sub>1</sub>* = *str<sub>2</sub>* and +1 if *str<sub>1</sub>* > *str<sub>2</sub>*. The strings are retained. For example,

```
" hello" $" HELLO" CMP$ . 1 ok:0
" hello" $" hello" CMP$ . 0 ok:0
" hell" $" hello" CMP$ . -1 ok:0
```

A case insensitive comparison can easily be built as follows:

```
: CMPCIS$ ( --- flag ) ( SS: str1 str2 --- str1 str2 )
  OVER$ OVER$ UCASE$ SWAP$
  UCASE$ CMP$ DROP$ DROP$ ;
```

The above code creates copies of *str<sub>1</sub>* and *str<sub>2</sub>* (using **OVER\$** ) then converts them both to upper case. **CMP\$** then compares the strings placing the appropriate flag on the data stack. Finally, the uppercase versions of *str<sub>1</sub>* and *str<sub>2</sub>* are removed from the string stack; thus, *str<sub>1</sub>* and *str<sub>2</sub>* are retained, unchanged.

**CODE** Resident

( --- )

A defining word initializing the definition of a code (assembly) word. It sets the context vocabulary to Assembler. See Chapter 9 “The fbForth 2.0 TMS9900 Assembler” for details. See also **ASM:** .

**COINC** Resident

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

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

**COINCALL** Resident

( --- flag )

Detects a coincidence between the visible portions of any two sprites on the display screen. A true flag indicates a coincidence, but not which sprites.

<b>COINCXY</b>	Resident
( <i>dotcol dotrow spr tol --- flag</i> )	
Detects a coincidence between a specified sprite and a given point ( <i>dotcol, dotrow</i> ) within a given tolerance of <i>tol</i> dot positions. A true flag indicates a coincidence.	
<b>COLD</b>	Resident
( --- )	
<b>COLD</b> is the cold-start procedure that resets user variables to their startup values, including the dictionary pointer (to point to just after the resident dictionary) and restarts <b>fbForth 2.0</b> via <b>BOOT</b> (resets the current blocks file to the default DSK1.FBLOCKS, loading block 1) and <b>ABORT</b> , <i>q.v.</i> It may be called from the terminal to remove application programs and to restart <b>fbForth 2.0</b> .	
<b>COLOR</b>	Resident
( $n_1 n_2 n_3$ --- )	
Causes a specified character set $n_3$ to have the given foreground color $n_1$ and background color $n_2$ .	
<b>COLTAB</b>	Resident
( --- <i>vaddr</i> )	
A constant whose value is the beginning VDP address of the color table. The default value is <b>380h</b> . This constant can only be changed via user variable number <b>22h</b> .	
<b>COMPILE</b>	Resident
( --- )	
<b>COMPILE</b> is a compile-only word that will execute when its containing word executes, which means that its containing word must be a compile-only word that executes during compilation, <i>i.e.</i> , an immediate word. This effectively defers compilation of the word following <b>COMPILE</b> until the word containing them is executed within the definition of yet another word.	
When the word containing <b>COMPILE</b> executes during the compilation of a new word, the execution address <i>cf</i> of the word following <b>COMPILE</b> is copied (compiled) into the dictionary entry for the new word's definition. For example,	
<pre> : WORD1 ... COMPILE WORD0 ... ; IMMEDIATE : WORD2 WORD1 ... ; </pre>	
When <b>WORD2</b> is compiled, <b>WORD1</b> executes, which executes <b>COMPILE</b> to place the <i>cf</i> of <b>WORD0</b> into the definition of <b>WORD2</b> .	
<b>CONSTANT</b>	Resident
( $n$ --- )	
A defining word used in the form:	
$n$ <b>CONSTANT</b> <i>cccc</i>	
to create word <i>cccc</i> , with its parameter field containing $n$ . When <i>cccc</i> is later executed, it will invoke <b>CONSTANT</b> 's execution procedure to 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>	Resident
( $f_1$ --- $f_2$ )	
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>b</i> )	
Leave the byte address <i>addr</i> <sub>2</sub> and byte count <i>b</i> of the packed character string (see footnote 4 on page 19) beginning at <i>addr</i> <sub>1</sub> . It is presumed that the first byte at <i>addr</i> <sub>1</sub> contains the character count <i>b</i> and that the actual text starts with the second byte. Typically, <b>COUNT</b> is followed by <b>TYPE</b> .	
<b>CPYBLK</b>	CPYBLK -- Block Copying Utility [4]
( --- )	
Copy a range of blocks from one blocks file to the same or a different blocks file. The destination file must already exist. The copy is overlap safe for same file copies. The source blocks copied are enumerated during the copy.	
Usage:	
<b>CPYBLK <i>src_start src_end src-file dst_start dst-file</i> ,</b>	
where <b><i>src_start</i></b> and <b><i>src_end</i></b> are source start and end block numbers, <b><i>src-file</i></b> is the source blocks file, <b><i>dst_start</i></b> is the destination start block number and <b><i>dst-file</i></b> is the destination blocks file.	
Example:	
<b>CPYBLK 4 10 DSK1.FBLOCKS 25 DSK2.MYBLOCKS</b> <b>4 5 6 7 8 9 10 ok:0</b>	
will copy blocks 4 – 10 from DSK1.FBLOCKS to DSK2.MYBLOCKS, starting at block 25.	
<b>CR</b>	Resident
( --- )	
Transmit a carriage return and a line feed to the current output device.	
<b>CREATE</b>	Resident
( --- )	
A defining word used in the form:	
<b>CREATE cccc</b>	
by such words as <b>:</b> , <b>&lt;BUILDS</b> , <b>ASM:</b> and <b>CODE</b> to create a dictionary header for a Forth definition. The code field contains the address of the word's parameter field.	

Space for the parameter field is *not* reserved by **CREATE** . The new word is created in the **CURRENT** vocabulary. It should be noted that new word names should *never* exceed 31 characters in length in **fbForth 2.0!**

<b>CSP</b>	<p>( --- <i>addr</i> )</p> <p>A user variable temporarily storing the stack pointer position for compilation error checking.</p>	Resident
<b>CURPOS</b>	<p>( --- <i>addr</i> )</p> <p>A user variable that stores the current VDP (Visual Display Processor) screen cursor position.</p>	Resident
<b>CURRENT</b>	<p>( --- <i>addr</i> )</p> <p>A user variable pointing to the vocabulary into which new definitions will be compiled. <b>DEFINITIONS</b> will store the contents of <b>CONTEXT</b> into <b>CURRENT</b> . At system startup, <b>CURRENT</b> points to the <b>FORTH</b> vocabulary.</p>	Resident
<b>D+</b>	<p>( <math>d_1 d_2</math> --- <math>d_3</math> )</p> <p>Leave the double number sum of two double numbers ( <math>d_3 = d_1 + d_2</math> ).</p>	Resident
<b>D+-</b>	<p>( <math>d_1 n</math> --- <math>d_2</math> )</p> <p>Apply the sign of <math>n</math> to the double number <math>d_1</math>, leaving it as <math>d_2</math>.</p>	Resident
<b>D.</b>	<p>( <math>d</math> --- )</p> <p>Print a signed double number from a 32-bit two's complement value <math>d</math>. The high-order 16 bits are most accessible on the stack. Conversion is performed according to the current radix in <b>BASE</b> . A blank follows. Pronounced "d dot".</p>	Resident
<b>D.R</b>	<p>( <math>d n</math> --- )</p> <p>Print a signed double number <math>d</math> right-aligned in a field <math>n</math> characters wide.</p>	Resident
<b>DABS</b>	<p>( <math>d_1</math> --- <math>d_2</math> )</p> <p>Leave the absolute value <math>d_2</math> of a double number <math>d_1</math>.</p>	Resident
<b>DCOLOR</b>	<p>( --- <i>addr</i> )</p> <p>A variable which contains the dot-color information used by <b>DOT</b> . Its value may be a two-digit hexadecimal number that will be used to set the foreground and background color or -1 to signal that no color information is to be changed.</p>	Resident

<b>DECIMAL</b>	Resident
( --- )	
Set the radix in <b>BASE</b> for decimal input/output.	
<b>DEFBF</b>	Resident
( --- <i>addr</i> )	
Gets the default blocks filename (DSK1.FBLOCKS) from VRAM to <b>PAD</b> and leaves the <b>PAD</b> address on the stack.	
<b>DEFINITIONS</b>	Resident
( --- )	
Sets the <b>CURRENT</b> vocabulary to the <b>CONTEXT</b> vocabulary by copying the contents of <b>CONTEXT</b> to <b>CURRENT</b> . Executing a vocabulary name makes it the <b>CONTEXT</b> vocabulary and executing <b>DEFINITIONS</b> makes both specify the same vocabulary. The following example will make both <b>CONTEXT</b> and <b>CURRENT</b> point to the <b>FORTH</b> vocabulary, which is the system default:	
<b>FORTH DEFINITIONS ok:0</b>	
<b>DEG/RAD</b>	Resident
( --- <i>f</i> )	
Constant in floating point format representing degrees/radian = 57.295779513082.	
<b>DELALL</b>	Resident
( --- )	
Delete all sprites. <b>DELALL</b> stops sprite motion, fills the sprite motion table with zeroes and stores <b>D0h</b> in the <b>y</b> position of all 32 sprites to leave them in an undefined state. <b>DELALL</b> does nothing to the sprite descriptor table. See § 6.6.2 for details.	
<b>DELALL</b> must be used to initialize sprites after changing to the desired VDP mode.	
<b>DELSPR</b>	Resident
( <i>spr</i> --- )	
Delete the specified sprite by positioning it off-screen at $x = 1$ , $y = 192$ ; setting it to sprite pattern #0; and clearing its motion table entries.	
<b>DEPTH</b>	Resident
( --- <i>n</i> )	
Return the number of cells on the parameter stack. This word is used by the new command-line ( <b>ok:n</b> ) response, where <b>n</b> indicates stack depth.	
<b>DEPTH\$</b>	Stack-based String Library [42]
( --- <i>n</i> ) ( SS: --- )	
Returns the current depth <i>n</i> of the string stack, with 0 meaning the string stack is empty.	

<b>DIGIT</b>	Resident
<i>( char n<sub>1</sub> --- false   n<sub>2</sub> true )</i>	
Convert the ASCII character <i>char</i> (using number base <i>n<sub>1</sub></i> ) to its binary equivalent <i>n<sub>2</sub></i> , accompanied by a true flag. If the conversion is invalid, leave only a false flag. For example, “ <b>DECIMAL 53 10 DIGIT</b> ” will leave “ <b>5 1</b> ” on the stack because 53 is the ASCII code for ‘5’ and is a legitimate digit in base 10. On the other hand, “ <b>DECIMAL 74 16 DIGIT</b> ” will leave only “ <b>0</b> ” on the stack because 74 is the ASCII code for ‘J’ and is <i>not</i> a legitimate digit in base 16. However, “ <b>DECIMAL 74 20 DIGIT</b> ” will leave “ <b>19 1</b> ” on the stack because ‘J’ is a legitimate digit in base 20.	
<b>DIR</b>	DIR--Disk Catalog Utility [36]
<i>( --- )</i>	
<b>DIR</b> catalogs to the output device the disk device name that follows it in the input stream. The disk device name must be terminated with a period. <b>DIR</b> gets its information from the DSR’s catalog “file”. <b>DIR</b> will not load if <b>CAT</b> is loaded.	
Usage: <b>DIR DSK1.</b>	
<b>DISK_BUF</b>	Resident
<i>( --- addr )</i>	
A user variable that points to the first byte in VDP RAM of the 128-byte <b>fbForth 2.0</b> record buffer.	
<b>DKB+</b>	Resident
<i>( n --- )</i>	
Defining word used to create words that calculate addresses from user variables containing offsets from <b>fbForth 2.0</b> ’s VRAM record buffer. Execution of the defined word pushes to the stack an address calculated by adding the record buffer address to the offset passed in the user variable, the user-variable-table offset of which is the parameter field value <i>n</i> passed to <b>DKB+</b> .	
Usage: <b>userVarOffset DKB+ new_word</b>	
<b>DLITERAL</b>	Resident
<i>[immediate word]</i>	
Compile time: <i>( d --- )</i> Runtime: <i>( --- d )</i> Interpreting: <i>( --- )</i>	
Same behavior as <b>LITERAL</b> , <i>q.v.</i> , except for a double number <i>d</i>	
<b>DLT</b>	Resident
<i>( --- )</i>	
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<sub>1</sub> --- d<sub>2</sub> )</i>	
Convert <i>d<sub>1</sub></i> to its double number two’s complement <i>d<sub>2</sub></i> , <i>i.e.</i> , <i>d<sub>2</sub> = -d<sub>1</sub></i> .	

**DMODE** Resident

( --- *addr* )

A variable that determines which dot mode is currently in effect. A **DMODE** 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 **DRAW** , **UNDRAW** and **DTOG** words.

**DO** Resident

*[immediate word]*

Occurs in a colon-definition in the form:

```
DO ... LOOP
DO ... +LOOP
```

Compile time: ( --- *addr* 3 )

When compiling within the colon-definition, **DO** compiles (**DO**) , leaving the following address *addr* and the value 3 for later error checking by the compile-time action of **LOOP** or **+LOOP** .

Runtime: ( *lim strt* --- )

**DO** begins a sequence with repetitive execution controlled by a loop limit *lim* and an index with initial value *strt*. **DO** removes these from the stack and puts them on the return stack, with the index on top. Upon reaching **LOOP** , the index is incremented by one. Until the new index equals or exceeds the limit, execution loops back to just after **DO** , otherwise the loop parameters are discarded and execution continues ahead. Both *lim* and *strt* are determined at runtime and may be the result of other operations. Within a loop, **I** will copy the current value of the index to the stack. See **I** , **LOOP** , **+LOOP** and **LEAVE** .

**DOES>** Resident

*[immediate word]*

( --- )

A word which defines the runtime action within a high-level defining word. **DOES>** alters the code field and first parameter of the new word to execute the sequence of compiled word addresses following **DOES>** . 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.

**DOES>ASM:** Resident

*[immediate word]*

( --- )

This is a synonym for **;CODE** , *q.v.*, intended to be paired with **;ASM** to form more readable **fbForth 2.0** Assembly language code as, for example,

```
: cccc <BUILDS ... DOES>ASM: ... ;ASM
```

See Chapter 9 The fbForth 2.0 TMS9900 Assembler for details.

<b>DOT</b>	Resident
<i>( dotcol dotrow --- )</i>	
In bitmap graphics, plots a dot at <i>(dotcol,dotrow)</i> in whatever mode is selected by <b>DMODE</b> and in whatever color is selected by <b>DCOLOR</b> .	
<b>DO_BRW</b>	Resident
<i>( [ bfnaddr   #blks bfnaddr   bufaddr blk# ] opcode --- )</i>	
Helper routine that executes <b>BLKRW</b> and processes returned flag. See <b>BLKRW</b> for items required on stack for each opcode and for an explanation of the stack effects abbreviations.	
<b>DP</b>	Resident
<i>( --- addr )</i>	
A user variable, the dictionary pointer, which contains the address of the next free memory above the dictionary. The value may be read by <b>HERE</b> and altered by <b>,</b> and <b>ALLOT</b> , among other words.	
<b>DPL</b>	Resident
<i>( --- addr )</i>	
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 for no decimal point. <b>DPL</b> is updated for every double number input.	
<b>DRAW</b>	Resident
<i>( --- )</i>	
Sets <b>DMODE</b> equal to 0. This means that dots are plotted in the ‘on’ state.	
<b>DROP</b>	Resident
<i>( n --- )</i>	
Drop the top number from the stack.	
<b>DROP\$</b>	Stack-based String Library [42]
<i>( --- ) ( SS: str --- ) “drop string”</i>	
The word <b>DROP\$</b> removes the topmost string item from the string stack. For example,	
<pre> \$" Hello, World!"  ok:0 \$" How are you?"  ok:0 DROP\$  ok:0 </pre>	
At this point the string “Hello, World!” is the topmost string the string stack. “How are you?” was pushed onto the string stack, but it was immediately dropped.	
<b>DSPLY</b>	Resident
<i>( --- )</i>	
Assigns the attribute DISPLAY to the file pointed to by <b>PAB-ADDR</b> .	

<b>DSRLNK</b>	Resident
( --- )	
Links an <b>fbForth 2.0</b> program to any Device Service Routine (DSR) in ROM. Before this instruction may be used, a PAB must be set up in VDP RAM and a pointer to PAB + 9 stored at <b>8356h</b> . See the <i>Editor/Assembler Manual</i> and Chapter 8 of this manual for additional setup information. This word automatically passes 8 to the DSR to execute DSR routines. It cannot execute DSR subprograms that require passing 10.	
<b>DTOG</b>	Resident
( --- )	
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>	Memory Dump Utility [16]
( <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. <b>DUMP</b> is 80-column-text-mode aware if your computer is so equipped. See <b>PAUSE</b> .	
<b>DUP</b>	Resident
( <i>n</i> --- <i>n n</i> )	
Duplicates the value on top of the stack.	
<b>DUP\$</b>	Stack-based String Library [42]
( --- ) ( SS: <i>str</i> <sub>1</sub> --- <i>str</i> <sub>1</sub> <i>str</i> <sub>1</sub> ) “duplicate string”	
The word <b>DUP\$</b> duplicates the top item on the string stack. For example,	
<b>\$" Hello, World!" DUP\$ ok:0</b>	
<b>DXY</b>	Resident
( <i>dotcol</i> <sub>1</sub> <i>dotrow</i> <sub>1</sub> <i>dotcol</i> <sub>2</sub> <i>dotrow</i> <sub>2</sub> --- <i>n</i> <sub>1</sub> <i>n</i> <sub>2</sub> )	
Places on the stack the square of the <i>x</i> distance <i>n</i> <sub>1</sub> and the square of the <i>y</i> distance <i>n</i> <sub>2</sub> between the points ( <i>dotcol</i> <sub>1</sub> , <i>dotrow</i> <sub>1</sub> ) and ( <i>dotcol</i> <sub>2</sub> , <i>dotrow</i> <sub>2</sub> ).	
<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>	Resident
( --- )	
Brings you back into the 40/80-column editor on the last <b>fbForth 2.0</b> block you edited. This block is pointed to by <b>SCR</b> . Must be in Text or Text80 mode.	

- ED@** (*EDITOR2 Vocabulary*) 64-Column Editor [6]  
 ( --- )  
 Brings you back into the 64-column editor on the last **fbForth 2.0** block you edited. This block is pointed to by **SCR** .
- EDIT** Resident  
 ( *blk* --- )  
 Brings you into the 40/80-column editor on the specified **fbForth 2.0** block, loading it from the current blocks file if necessary. Must be in Text or Text80 mode.
- EDIT** (*EDITOR2 Vocabulary*) 64-Column Editor [6]  
 ( *blk* --- )  
 Brings you into the 64-column editor on the specified **fbForth 2.0** block, loading it from the current blocks file if necessary.
- ELSE** [*immediate word*] Resident  
 Occurs within a colon-definition in the form:  
**IF ... ELSE ... ENDIF**  
 Compile time: ( *addr*<sub>1</sub> 2 --- *addr*<sub>2</sub> 2 )  
**ELSE** emplaces **BRANCH** , reserving a branch offset and leaves the address *addr*<sub>2</sub> and 2 for error testing because the incoming '2' is consumed for error checking. **ELSE** also resolves the pending forward branch from **IF** by calculating the offset from *addr*<sub>1</sub> to **HERE** and storing it at *addr*<sub>1</sub>.
- 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 7-bit ASCII character *char* to the current output device. **OUT** , *q.v.*, is incremented for each character output.
- EMIT8** Resident  
 ( *char* --- )  
 Transmit an 8-bit character *char* to the current output device. **OUT** , *q.v.*, 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 current blocks file. This is also an initialization procedure executed by **COLD** , *q.v.*, before first use of the default blocks file.

**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, *i.e.*, the character about to be read. This procedure will not process past an ASCII NUL (0), treating it as an unconditional parsing terminator.

**WORD** uses the output from **ENCLOSE** to advance **IN** by *n*<sub>3</sub> and calculate the parsed word's length as *n*<sub>2</sub> - *n*<sub>1</sub> for use in constructing the packed character string (see footnote 4 on page 19) for the word, which **WORD** copies to **HERE**.

If we let each '{ }' represent one character; each character is either a non-delimiter character, 'chr', a delimiter character, 'delim', or the null character, '0', **ENCLOSE** allows three possible parsing scenarios after leading delimiter characters are skipped:

- 1) *n*<sub>1</sub>*n*<sub>3</sub>{0}*n*<sub>2</sub>
- 2) *n*<sub>1</sub>{chr}...{chr}*n*<sub>2</sub>*n*<sub>3</sub>{0}
- 3) *n*<sub>1</sub>{chr}...{chr}*n*<sub>2</sub>{delim}*n*<sub>3</sub>{chr | 0}...

The offsets, *n*<sub>1</sub>, *n*<sub>2</sub> and *n*<sub>3</sub> are shown above in the positions they indicate when returned on the stack by **ENCLOSE**. Where they are shown next to each other, they, in fact, have the same value. One thing to keep in mind is that *n*<sub>3</sub> will never point to the position after an ASCII 0.

Scenario (1) above is important because it is the only way that **INTERPRET**, otherwise an infinite loop, can be forced to exit. The null character will be parsed as a single-character word that will be found in the dictionary and executed by **INTERPRET**, causing **INTERPRET**'s demise.

**END**

[immediate word]

Resident

COMPILE TIME: ( *addr* 1 --- )    RUNTIME: ( *flag* --- )

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

**ENDCASE**

[immediate word]

Resident

Occurs in a colon definition as the termination of the **CASE ... ENDCASE** construct.

COMPILE TIME: ( *csp* *addr*<sub>1</sub> ... *addr*<sub>*n*</sub> 4 --- )

It uses the 4 for compile-time error checking. It uses the value in **CSP** put there by **CASE** to track the number of **OF ... ENDOF** clauses for which it must calculate branch distances from the addresses (*addr*<sub>1</sub> ... *addr*<sub>*n*</sub>) that each **ENDOF** left on the stack.

RUNTIME: ( *n* --- )

If all **OF ... ENDOF** clauses fail, any code after the last **ENDOF**, including **ENDCASE**, will execute. **ENDCASE** will remove the number *n* left on the stack by the failure of the last **OF**.

If you include code between the last **ENDOF** and **ENDCASE**, it must leave at least one number on the stack for **ENDCASE** to consume to prevent stack underflow. See **CASE**.

**ENDIF** [immediate word] Resident

Occurs in a colon-definition in the form:

**IF** ... **ENDIF** (also **IF** ... **THEN**)  
**IF** ... **ELSE** ... **ENDIF** (also **IF** ... **ELSE** ... **THEN**)

Compile time: ( *addr* 2 --- )

**ENDIF** computes the forward branch offset from *addr* to **HERE** and stores it at the spot reserved for it at *addr*. The value 2 is used for error testing.

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**.

**ENDOF** [immediate word] Resident

Occurs in a colon definition as the termination of the **OF** ... **ENDOF** construct within the **CASE** ... **ENDCASE** construct.

Compile time: ( *addr*<sub>1</sub> 5 --- *addr*<sub>2</sub> 4 )

**ENDOF** checks for a 5 on the stack. It then compiles **BRANCH**, leaves its address *addr*<sub>2</sub> for processing by **ENDCASE**. It next leaves 4 on the stack for compile-time error checking by the next **OF** or **ENDCASE**. It finally calculates the forward branch offset from *addr*<sub>1</sub> to **HERE** for its matching **OF** and stores the value at the spot reserved for it at *addr*<sub>1</sub>.

Runtime: ( --- )

**ENDOF** causes execution to proceed after **ENDCASE**. See **OF**.

**ERASE** Resident

( *addr* *n* --- )

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

**ERROR** Resident

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

**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*<sub>2</sub> and **BLK** *n*<sub>3</sub> on the stack to assist in determining the location of the error. If **WARNING** > 0, **ERROR** prints the error text of system message number *n*<sub>1</sub>. If **WARNING** = 0, **ERROR** prints *n*<sub>1</sub> as an error number (This was used in TI Forth in a non-disk installation; but, the system messages are always present in **fbForth 2.0**). The last

	thing <b>ERROR</b> does is to execute <b>QUIT</b> , which, as above, stops compilation and restarts the interpreter.	
<b>EXECUTE</b>		Resident
	( <i>cfa</i> --- )	
	Execute the definition whose code field address is on the stack. The code field address is also called the compilation address.	
<b>EXIT</b>	[ <i>immediate word</i> ]	More Useful Stack Words etc. [41]
	( --- )	
	<b>EXIT</b> is a synonym for <b>;S</b> , which stops interpretation of a Forth block or ends the current word's execution and returns to the calling procedure.	
<b>EXP</b>		Resident
	( $f_1$ --- $f_2$ )	
	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.	
<b>EXPECT</b>		Resident
	( <i>addr count</i> --- )	
	Transfer characters from the terminal to <i>addr</i> until <b>&lt;ENTER&gt;</b> or <i>count</i> characters have been received. One or more nulls are added at the end of the text.	
<b>F!</b>		Resident
	( <i>f addr</i> --- )	
	Stores a floating point number $f$ into the 4 words (cells) beginning with the specified address.	
<b>F*</b>		Resident
	( $f_1 f_2$ --- $f_3$ )	
	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>		Resident
	( $f_1 f_2$ --- $f_3$ )	
	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>		Resident
	( $f_1 f_2$ --- $f_3$ )	
	Subtracts $f_2$ from $f_1$ and places the result on the stack ( $f_1 - f_2 = f_3$ ).	
<b>F-&gt;S</b>		Resident
	( <i>f</i> --- <i>n</i> )	
	Converts a floating point number $f$ on the parameter stack into a single precision number $n$ .	

<b>F-D"</b>	<i>[immediate word]</i> ( --- ) 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> .	Resident
<b>F.</b>	( <i>f</i> --- ) Prints a floating point number <i>f</i> in TI Basic format to the output device.	Resident
<b>F/</b>	( <i>f</i> <sub>1</sub> <i>f</i> <sub>2</sub> --- <i>f</i> <sub>3</sub> ) Divides <i>f</i> <sub>1</sub> by <i>f</i> <sub>2</sub> and leaves the floating point quotient <i>f</i> <sub>3</sub> on the stack. $f_1 / f_2 = f_3$ .	Resident
<b>F0&lt;</b>	( <i>f</i> --- <i>flag</i> ) Compares the floating point number <i>f</i> 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.	Resident
<b>F0=</b>	( <i>f</i> --- <i>flag</i> ) Compares the floating point number <i>f</i> 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.	Resident
<b>F&lt;</b>	( <i>f</i> <sub>1</sub> <i>f</i> <sub>2</sub> --- <i>flag</i> ) Leaves a true flag if $f_1 < f_2$ , else leaves a false flag.	Resident
<b>F=</b>	( <i>f</i> <sub>1</sub> <i>f</i> <sub>2</sub> --- <i>flag</i> ) Leaves a true flag if $f_1 = f_2$ , else leaves a false flag.	Resident
<b>F&gt;</b>	( <i>f</i> <sub>1</sub> <i>f</i> <sub>2</sub> --- <i>flag</i> ) Leaves a flag if $f_1 > f_2$ , else leaves a false flag.	Resident
<b>F&gt;R</b>	( <i>f</i> --- ) ( R: --- <i>f</i> ) Moves the 8-byte floating point number <i>f</i> from the parameter stack to the return stack. See <b>R&gt;F</b> .	Resident
<b>F@</b>	( <i>addr</i> --- <i>f</i> ) Retrieves the floating point contents <i>f</i> of the given address (4 words) and places it on the stack.	Resident

**FABS** Resident

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

Converts the floating point number  $f_1$  to its absolute value  $f_2$ .

**FCONSTANT** Resident

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

A defining word used in the form:

**f FCONSTANT cccc**

to create word **cccc**, with its parameter field containing  $f$ . When **cccc** is later executed, it will invoke **FCONSTANT**'s execution procedure to push the 8-byte floating point value of  $f$  to the stack.

**FDROP** Resident

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

Drops the top floating point number  $f$  from the stack.

**FDUP** Resident

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

Duplicates the top floating point number  $f$  on the stack.

**FENCE** Resident

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

A user variable containing an address (usually the NFA of a Forth word) below which **FORGET**ing is trapped. To **FORGET** below this point the user must alter the contents of **FENCE**. It is possible to set the value of **FENCE** to a value that is actually less than the address of the end of the last word in the core dictionary ( **TASK** ) such that **UNFORGETTABLE** [*sic*] will report false; however, **FORGET** will still trap that error.

**FFMT.** Resident

$(f [intLen fracLen] optMask \text{---})$

This word can handle free-format, TI Basic-style output of floating point numbers as well as fixed-format output that includes F-, E- and extended E-type formats. For free-format output, only the floating point number and  $optMask = 0$  is required on the stack.

$optMask$  is composed of the following bits:

- bit 0: 0 = free form TI Basic style
  - no other bits should be set
  - $intLen$  and  $fracLen$  should not be on the stack
  - 1 = fixed format
- bit 1: 2 = explicit sign
- bit 2: 4 = show '+' for positive number instead of space
  - bit 1 must also be set
- bit 3: 8 = E-notation (2 exponent digits)

- bit 4: 16 = extended E-notation (3 exponent digits)
  - bit 3 must also be set

If *optMask* is not 0, *intLen* and *fracLen* must be on the stack, as well. If the sum of *intLen* and *fracLen* exceeds 16, an error message will be displayed:

- *intLen*: number of places before decimal point, including sign position
- *fracLen*: number of places after decimal point, including decimal point and excluding E-notation

Various examples, using floating point numbers defined with **>F** and **FPCON**, *q.v.*, follow:

```
>F 1.234567890123E-7 ok:4
0 FFMT. 1.23457E-07 ok:0
>F 1.234567890123E-102 FCONSTANT FPCON ok:0
FPCON ok:4
0 FFMT. 1.23457E-** ok:0
FPCON FDUP FDUP FDUP FDUP ok:20
2 14 1 FFMT. .0000000000000000 ok:16
2 10 9 FFMT. 1.234567890E-** ok:12
2 12 25 FFMT. 1.23456789012E-102 ok:8
2 14 31 FFMT. +1.2345678901230E-102 ok:4
2 15 9 FFMT. field too big! ok:0
```

**FILE**

Resident

( *vaddr*<sub>1</sub> *addr* *vaddr*<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 **PAB-ADDR**, **PAB-BUF** and **PAB-VBUF** addresses you wish to be associated with the file.

Used in the form:

***vaddr*<sub>1</sub> *addr* *vaddr*<sub>2</sub> FILE cccc**

When **cccc** executes, **PAB-ADDR**, **PAB-BUF** and **PAB-VBUF** are set to *vaddr*<sub>1</sub>, *addr* and *vaddr*<sub>2</sub>, respectively.

**FILES**

Resident

( *n* --- )

Change the number of files **fbForth 2.0** can have open simultaneously. The number of files can be 1 – 16. Each additional file requires an additional 518 bytes of upper VRAM, reducing the available VRAM for your program. Location **8370h** holds the highest available address in VRAM.

**FILL**

Resident

( *addr* *count* *b* --- )

Fill memory beginning at *addr* with *count* bytes of byte *b*.

**FIND\$** Stack-based String Library [42]

( *start* --- *pos* | -1) ( SS: --- ) “find string in string”

The word **FIND\$** searches the second string on the string stack, starting from position *start*, for the first occurrence of the topmost string and pushes its starting position to the data stack. As a convenience to making subsequent searches for the same substring easier, both strings are retained on the string stack. For example,

```
$" redgreenbluegreen" $" green" 0 FIND$ . 3 ok:0
```

displays the value 3, as the substring is found at character position 3, the leftmost character being character 0. The strings “redgreenbluegreen” and “green” remain on the stack; thus, the second instance of “green” could be found with a second search.

**FINDC\$** Stack-based String Library [42]

( *char* --- *pos* | -1) ( SS: --- ) “find character in string”

The word **FINDC\$** returns the position of the first occurrence of the character *char*, beginning at the left side of the topmost string, with the search proceeding towards the right. If the character is not found, -1 is returned. For example,

```
$" redgreenblue" 98 FINDC$ . 8 ok:0
```

Displays the value 8, as the character ‘b’ (ASCII 98) is found in the 8th character position, where the first character is character 0.

**FIRST** Resident

( --- *addr* )

A constant that leaves the address of the first (lowest) block buffer.

**FIRST\$** Resident

( --- *addr* )

A user variable which contains the first byte of the disk buffer area.

**FLD** Resident

( --- *addr* )

A user variable for control of number output field width. Presently unused in fig-Forth and **fbForth 2.0**.

**FLERR** Resident

( --- *n* )

Returns on the stack the contents *n*, reported by the floating point library in its floating point error variable.

**FLOOR** Resident

( *f*<sub>1</sub> --- *f*<sub>2</sub> )

Finds the greatest integer *f*<sub>2</sub> (in floating point format) not less than the floating point number *f*<sub>1</sub>.

**FLUSH** Resident

( --- )

Writes to disk all disk buffers that have been marked as updated.

<b>FMINUS</b>		Resident
	$(f_1 \text{ --- } f_2)$	
	Negates $f_1$ by taking the two's complement of the cell on top of the stack, <i>i.e.</i> , $f_2 = \neg f_1$ . The top cell on the stack is the most significant cell (2 bytes) of a floating point number.	
<b>FNT</b>		Resident
	$(\text{---})$	
	Loads the default font from cartridge ROM to the Pattern Descriptor Table if <b>SCRFONT</b> = 0 or a user font from a font file if <b>SCRFONT</b> $\neq$ 0. To load a font file, you must execute <b>USEFFL</b> ( <i>q.v.</i> ) followed by a filename. To restore loading the default font after having loaded a font file, change the user variable <b>SCRFONT</b> ( <i>q.v.</i> ) back to 0. Executing <b>COLD</b> or restarting the system will also restore <b>SCRFONT</b> to 0.	
<b>FONTEd</b>		Resident
	$(\text{---})$	
	Typing <b>FONTEd</b> opens the font editor with the current font loaded from the PDT. Editing the font will not affect the current font because the working buffer is not the PDT. See Chapter 13 "Screen Fonts and the Font Editor" for details.	
<b>FORGET</b>		Resident
	$(\text{---})$	
	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.	
	If you wish to <b>FORGET</b> an unfinished definition, the word likely will not be found. If it is the last definition attempted, you can make it findable by executing <b>SMUDGE</b> and then <b>FORGET</b> ting it.	
<b>FORTH</b>	<i>[immediate word]</i>	Resident
	$(\text{---})$	
	The name of the primary vocabulary. Execution makes <b>FORTH</b> the <b>CONTEXT</b> vocabulary. Until additional user vocabularies are defined, new user definitions become a part of <b>FORTH</b> because it is at that point also the <b>CURRENT</b> vocabulary. Because <b>FORTH</b> is immediate, it will execute during the creation of a colon definition to select this vocabulary at compile time.	

<b>FOVER</b>	<p><math>(f_1 f_2 \text{ --- } f_1 f_2 f_1)</math> Copies the second floating point number on the stack to the top of the stack.</p>	Resident
<b>FP1</b>	<p><math>(\text{ --- } f)</math> Pushes a floating point 1 <math>f</math> to the stack.</p>	Resident
<b>FP10</b>	<p><math>(\text{ --- } f)</math> Pushes a floating point 10 <math>f</math> to the stack.</p>	Resident
<b>FPB</b>	<p><math>(\text{ --- } vaddr)</math> Pushes VRAM address <math>vaddr</math> of user screen font file PAB to stack.</p>	Resident
<b>FRAC</b>	<p><math>(f_1 \text{ --- } f_2)</math> Truncates <math>f_1</math>, leaving the fractional portion <math>f_2</math> on the stack.</p>	Resident
<b>FRND</b>	<p><math>(\text{ --- } f)</math> Generates a pseudo-random floating point number <math>f</math> greater than or equal to 0 and less than 1.</p>	Resident
<b>FROT</b>	<p><math>(f_1 f_2 f_3 \text{ --- } f_2 f_3 f_1)</math> Moves the third floating point number <math>f_1</math> down from the top of the stack to the top of the stack.</p>	Resident
<b>FSWAP</b>	<p><math>(f_1 f_2 \text{ --- } f_2 f_1)</math> Swaps the top two floating point numbers on the stack.</p>	Resident
<b>FVARIABLE</b>	<p><math>(\text{ --- } addr)</math> A defining word used in the form:  <div style="text-align: center;"><b><math>f</math> FVARIABLE cccc</b></div> to create word <b>cccc</b>, with its parameter field containing <b><math>f</math></b>. When <b>cccc</b> is later executed, it will invoke <b>FVARIABLE</b>'s execution procedure to push <b>cccc</b>'s address <math>addr</math> to the stack.</p>	Resident
<b>FXD</b>	<p><math>(\text{ --- } )</math> Assigns the attribute FIXED to the file whose PAB (Peripheral Access Block) is pointed to by <b>PAB-ADDR</b>.</p>	Resident

<b>GCHAR</b>	<p>( <i>col row --- char</i> )</p> <p>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.</p>	Resident
<b>GOTOXY</b>	<p>( <i>col row ---</i> )</p> <p>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.</p>	Resident
<b>GPLLNK</b>	<p>( <i>addr ---</i> )</p> <p>Links a Forth program to the Graphics Programming Language (GPL) routine located at the given address.</p>	Resident
<b>GRAPHICS</b>	<p>( --- )</p> <p>Converts from present display screen mode into standard Graphics mode configurations.</p>	Resident
<b>GRAPHICS2</b>	<p>( --- )</p> <p>Converts from present display screen mode into standard Graphics2 (Bitmap) mode configuration.</p>	Resident
<b>HCHAR</b>	<p>( <i>col row count char ---</i> )</p> <p>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.</p>	Resident
<b>HERE</b>	<p>( --- <i>addr</i> )</p> <p>Leave the address of the next available dictionary location.</p>	Resident
<b>HEX</b>	<p>( --- )</p> <p>Set the numeric conversion base to sixteen (hexadecimal).</p>	Resident
<b>HLD</b>	<p>( --- <i>addr</i> )</p> <p>A user variable that holds the address of the latest character of text during numeric output conversion.</p>	Resident
<b>HOLD</b>	<p>( <i>char ---</i> )</p> <p>Used between &lt;# and #&gt; to insert an ASCII character into a pictured numeric output string, e.g., <b>2E HOLD</b> will place a decimal point.</p>	Resident

<b>HONK</b>	<p>( --- )</p> <p>Produces the sound associated with incorrect input.</p>	Resident
<b>I</b>	<p>( --- <i>n</i> )</p> <p>Used within a <b>DO</b> loop to copy the loop index to the stack. <b>I</b> is a synonym for <b>R</b> .</p>	Resident
<b>ID.</b>	<p>( <i>nfa</i> --- )</p> <p>Print a definition's name from its name field address <i>nfa</i>.</p>	Resident
<b>IF</b>	<p style="text-align: center;"><i>[immediate word]</i></p> <p>Occurs in a colon definition in form:</p> <p style="margin-left: 40px;"><b>IF (true part) ... THEN</b>  <b>IF (true part) ... ENDIF</b>  <b>IF (true part) ... ELSE (false part) ... THEN</b>  <b>IF (true part) ... ELSE (false part) ... ENDIF</b></p> <p>Compile time: ( --- <i>addr</i> 2 )</p> <p><b>IF</b> compiles <b>ØBRANCH</b> and reserves space for an offset at <i>addr</i>; <i>addr</i> and 2 are used later for resolution of the offset and error testing.</p> <p>Runtime: ( <i>flag</i> --- )</p> <p><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 when an <b>ELSE</b> clause is present. After either part, execution resumes after <b>THEN</b> (or <b>ENDIF</b> ). <b>ELSE</b> and its false part are optional. With no <b>ELSE</b> clause, false execution skips to just after <b>THEN</b> (or <b>ENDIF</b> ).</p>	Resident
<b>IMMEDIATE</b>	<p>( --- )</p> <p>Mark the most recently made definition so that when encountered at compile time, it will be executed rather than being compiled. <i>i.e.</i>, 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 <b>[COMPILE]</b> .</p>	Resident
<b>IN</b>	<p>( --- <i>addr</i> )</p> <p>A user variable containing the byte offset within the current input text buffer (terminal or disk) from which the next text will be accepted. <b>WORD</b> uses and moves the value of <b>IN</b> .</p>	Resident
<b>INDEX</b>	<p>( <i>n</i><sub>1</sub> <i>n</i><sub>2</sub> --- )</p> <p>Prints to the terminal a list of the line #0 comments from Forth block <i>n</i><sub>1</sub> through Forth block <i>n</i><sub>2</sub>. See <b>PAUSE</b> .</p>	Printing Routines [19]

<b>INIT\$</b>	Stack-based String Library [42]
<i>( size --- ) ( SS: --- )</i> “initialize string stack”	
The string stack must be initialized to some convenient size by executing <b>INIT\$</b> once the library is <b>LOADED</b> :	
<b>512 INIT\$ ok:0</b>	
will initialize the string stack to 512 bytes. <b>INIT\$</b> should only be executed once because initializing the string stack a second time will orphan the previous instance and waste memory.	
<b>INPT</b>	Resident
<i>( --- )</i>	
Assigns the attribute INPUT to the file whose PAB is pointed to by <b>PAB-ADDR</b> .	
<b>INT</b>	Resident
<i>( f<sub>1</sub> --- f<sub>2</sub> )</i>	
Leaves the integer portion of a floating point number on the stack.	
<b>INTERPRET</b>	Resident
<i>( --- )</i>	
The outer text interpreter, which sequentially executes or compiles text from the input stream (terminal or disk) depending on <b>STATE</b> . If the word name cannot be found after a search of <b>CONTEXT</b> and then <b>CURRENT</b> , <b>INTERPRET</b> attempts to convert it into a number according to the current radix in <b>BASE</b> . That also failing, an error message echoing the name with a “?” will be given. Text input will be taken according to the convention for <b>WORD</b> . 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 <b>NUMBER</b> .	
<b>INTLNK</b>	Resident
<i>( --- addr )</i>	
A user variable which is a pointer to the Interrupt Service linkage.	
<b>INTRNL</b>	Resident
<i>( --- )</i>	
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> )	
Used within an inner <b>DO</b> loop to copy the loop index of the next outer <b>DO</b> loop to the stack.	
<b>JCRU</b>	Resident
( <i>n</i> <sub>1</sub> --- <i>n</i> <sub>2</sub> )	
Executed by <b>JOYST</b> when <b>JMODE</b> ≠ 0, <b>JCRU</b> allows input from joystick #1 ( <i>n</i> <sub>1</sub> = 1) or #2 ( <i>n</i> <sub>1</sub> = 2). The value <i>n</i> <sub>2</sub> returned will have 0 or more of the 5 least significant bits set for direction and fire-button status. Bit values are 1 = Fire, 2 = W, 4 = E, 8 = S and 16 = N. Two-bit directional combinations are 18 = NW (N + W or 16 + 2), 20 = NE, 10 = SW and 12 = SE. See § 6.8 “Using Joysticks” for more information.	
<b>JKBD</b>	Resident
( <i>n</i> <sub>1</sub> --- <i>char</i> <i>n</i> <sub>2</sub> <i>n</i> <sub>3</sub> )	
Executed by <b>JOYST</b> when <b>JMODE</b> = 0, <b>JKBD</b> allows 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> . See § 6.8 “Using Joysticks” for more information.	
<b>JMODE</b>	Resident
( --- <i>addr</i> )	
A user variable that uses offset <b>26h</b> of the user variable table. It is used by <b>JOYST</b> to determine whether to execute <b>JKBD</b> (= 0) or <b>JCRU</b> (≠ 0). The default value is 0. See <b>JOYST</b> , <b>JKBD</b> and <b>JCRU</b> .	
<b>JOYST</b>	Resident
( <i>n</i> <sub>1</sub> --- [ <i>char</i> <i>n</i> <sub>2</sub> <i>n</i> <sub>3</sub> ]   <i>n</i> <sub>2</sub> )	
Allows 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). Return values depend on the value in <b>JMODE</b> . If <b>JMODE</b> = 0 (default), <b>JOYST</b> executes <b>JKBD</b> , which returns 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> . If <b>JMODE</b> ≠ 0, <b>JOYST</b> executes <b>JCRU</b> , which reads only the joysticks and returns a single value with 0 or more of the 5 least significant bits set. See <b>JCRU</b> and § 6.8 “Using Joysticks” for their meaning.	
<b>KEY</b>	Resident
( --- <i>char</i> )	
Wait for the next terminal keystroke. Leave its ASCII (7-bit) value on the stack.	
<b>KEY8</b>	Resident
( --- <i>char</i> )	
Wait for the next terminal keystroke. Leave its full 8-bit value on the stack.	

<b>L/SCR</b>	Resident
( --- <i>n</i> )	
Returns on the stack the number of lines per Forth block.	
<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.	
<b>LCASE\$</b>	Stack-based String Library [42]
( --- ) ( SS: <i>str</i> <sub>1</sub> --- <i>str</i> <sub>2</sub> ) “convert to lower case”	
The word <b>LCASE\$</b> converts all upper case characters in the topmost string to lower case. For example,	
<b>\$" HELLO WORLD! 1234" LCASE\$ . \$ <u>hello world!</u> <u>1234</u> ok:0</b>	
<b>LD</b>	Resident
( <i>n</i> --- )	
The file I/O process to load a program file from a disk into VDP RAM. The parameter <i>n</i> 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 <b>PAB-ADDR</b> points, before executing this word.	
<b>LDCR</b>	CRU Words [5]
( <i>n</i> <sub>1</sub> <i>n</i> <sub>2</sub> <i>addr</i> --- )	
Performs a TMS9900 LDCR instruction. The CRU base address <i>addr</i> will be shifted left one bit and stored in workspace register R12 prior to executing the TMS9900 LDCR instruction. The low-order <i>n</i> <sub>2</sub> bits of value <i>n</i> <sub>1</sub> are transferred to the CRU, where the following condition, <i>n</i> <sub>2</sub> ≤ 15, is enforced by <i>n</i> <sub>2</sub> AND 0Fh. If <i>n</i> <sub>2</sub> = 0, 16 bits are transferred. For program clarity, you may certainly use <i>n</i> <sub>2</sub> = 16 to transfer 16 bits because <i>n</i> <sub>2</sub> = 0 will be the value actually used by the final machine code. See § 11.3 and CRU documentation in the <i>Editor/Assembler Manual</i> for more information.	
<b>LEAVE</b>	Resident
( --- )	
Force termination of a <b>DO</b> 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 <b>LOOP</b> or <b>+LOOP</b> is encountered.	
<b>LEFT\$</b>	Stack-based String Library [42]
( <i>len</i> --- ) ( SS: <i>str</i> <sub>1</sub> --- <i>str</i> <sub>1</sub> <i>str</i> <sub>2</sub> ) “left of string”	
The word <b>LEFT\$</b> pushes the leftmost <i>len</i> characters to the string stack as a new string. The original string is retained. For example,	

```
$" redgreenblue" 3 LEFT$ ok:0
```

The above causes the string “red” to be pushed to the string stack.

**LEN\$** Stack-based String Library [42]

( --- *len* ) ( SS: --- ) “length of string”

The word **LEN\$** returns the length of the topmost string on the string stack. For example,

```
$" Hello world!" len$ . 12 ok:0
```

**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** Resident

( *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.

**LIST** Resident

( *blk* --- )

Lists the specified Forth block to the output device. See **PAUSE** .

**LIT** Resident

( --- *n* )

Within a colon-definition, **LIT** is automatically compiled before each 16-bit literal number encountered in input text. Later execution of **LIT** causes the contents of the next dictionary address to be pushed to the stack.

**LITERAL** [*immediate word*] Resident

INTERPRETATION: ( --- )

Interpretation of **LITERAL** does nothing, unlike almost all other compiling words.

COMPILE TIME: ( *n* --- )

Compiles the stack value *n* as a 16-bit literal. This will execute during a colon definition. The intended use is:

: **xxx** [ *calculation* ] **LITERAL** ;

Compilation is suspended for the compile-time calculation of a value. Compilation is resumed and **LITERAL** compiles this value.

RUNTIME: ( --- *n* )

Pushes *n* to the stack.

**LN10INV**

Resident

( --- *f* )

Leaves the floating point number  $f = 0.43429448190325$ , which is the inverse of  $\ln(10) = 2.302585092994$ .

**LOAD**

Resident

( *n* --- )

Begin interpretation of Forth block *n*. Loading will terminate at the end of the Forth block or at **;S**. See **;S** and **-->**.

**LOG**

Resident

( *f*<sub>1</sub> --- *f*<sub>2</sub> | *f*<sub>1</sub> )

The floating point operation that returns the natural logarithm  $f_2$  of the floating point number  $f_1$ . If  $f_1$  is 0 or negative, the original number  $f_1$  is returned instead.

**LOG10**

Resident

( *f*<sub>1</sub> --- *f*<sub>2</sub> | *f*<sub>1</sub> )

The floating point operation that returns the decimal logarithm  $f_2$  of the floating point number  $f_1$ . If  $f_1$  is 0 or negative, the original number  $f_1$  is returned instead.

**LOOP**

[*immediate word*]

Resident

Occurs in a colon definition in the form:

**DO ... LOOP**

Compile time: ( *addr* 3 --- )

**LOOP** compiles (**LOOP**) and uses *addr* to calculate an offset to **DO**. The value 3 is used for compile-time error testing.

RUNTIME: ( --- )

**LOOP** selectively controls branching back to the corresponding **DO** based on the loop index and limit. The loop index is incremented by one and compared to the limit. The branch back to **DO** occurs until the index equals or exceeds the limit. At that time, the parameters are discarded and execution continues ahead.

**LTRIM\$**

Stack-based String Library [42]

( --- ) ( SS: *str*<sub>1</sub> --- *str*<sub>2</sub> ) “trim left of string”

The word **LTRIM\$** removes leading spaces from the topmost string. For example,

**\$" hello!" LTRIM\$ . \$ hello! ok:0**

<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>	Resident
$( n_1 \text{ --- } )$	
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>MAXLEN\$</b>	Stack-based String Library [42]
$( \$Caddr \text{ --- } max\_len )$ “maximum length of string”	
Given the address of a string constant on the data stack the word <b>MAXLEN\$</b> returns the maximum allowed string length for that string constant. For example,	
<pre> 50 \$CONST WELCOME ok:0 WELCOME MAXLEN\$ . 50 ok:0 </pre>	
<b>MCHAR</b>	Resident
$( n col row \text{ --- } )$	
Places a square of color $n$ at $( col, row )$ . Used in multicolor mode.	
<b>MENU</b>	Welcome Block [1]
$( \text{ --- } )$	
Displays the available Load Options.	
<b>MESSAGE</b>	Resident
$( n \text{ --- } )$	
Print on the selected output device the text of system error number $n$ . If <b>WARNING</b> = 0, the message will simply be printed as a number ( <b>msg #n</b> ). When <b>WARNING</b> = 0 in	

TI Forth, it means the disk is unavailable; but, this is not necessary in **fbForth 2.0** because error messages are always memory resident.

The word **MESSAGE** now only works for predefined error messages and should not be used to display user-defined messages as was possible with TI Forth. The reason for this is that system messages in **fbForth 2.0** now reside in cartridge ROM. The word **.LINE**, *q.v.*, can be used for this purpose.

**MID\$**

Stack-based String Library [42]

( *start end ---* ) ( SS: *str*<sub>1</sub> --- *str*<sub>1</sub> *str*<sub>2</sub> ) “mid-string”

The word **MID\$** produces a sub-string on the string stack, consisting of the characters from the topmost string starting at character *start* and ending at character *end*. For example,

```
$" redgreenblue" 3 7 mid$ ok:0
```

At this point, the topmost two strings on the string stack are as follows:

“green” (the topmost item)

“redgreenblue”

Note, as indicated in the string stack signature, the original string (*str*<sub>1</sub>) is retained. Note also that the first character in the string (the leftmost character) is character number 0.

**MIN**

Resident

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

Leave the smaller *n*<sub>3</sub> of the two numbers ( *n*<sub>1</sub> and *n*<sub>2</sub> ).

**MINIT**

Resident

( --- )

Initializes the monitor screen for use with **MCHAR** .

**MINUS**

Resident

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

Leaves the two's complement *n*<sub>2</sub> of a number *n*<sub>1</sub>, *i.e.*, negates *n*<sub>1</sub>.

**MKBFL**

Resident

( --- )

Create a blocks file from the string and number in the input stream. To create a file named MYBLOCKS on DSK1 with room for 80 blocks, type

```
MKBFL DSK1.MYBLOCKS 80 ok:0
```

**MOD**

Resident

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

Leave the remainder *rem* of *n*<sub>1</sub>/*n*<sub>2</sub>, with the same sign as *n*<sub>1</sub>.

**MON**

Resident

( --- )

Exit to the TI 99/4A color bar display screen and the system monitor program.

<b>MOTION</b>	Resident
<i>( n<sub>1</sub> n<sub>2</sub> 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<sub>1</sub> addr<sub>2</sub> n --- )</i>	
Moves the contents of $n$ cells (16-bit contents) beginning at $addr_1$ into $n$ cells beginning at $addr_2$ . The contents of $addr_1$ is moved first, proceeding toward high memory. This is <i>not</i> overlap safe for $addr_1 < addr_2$ .	
<b>MULTI</b>	Resident
<i>( --- )</i>	
Converts from present display screen mode into standard Multicolor mode configuration.	
<b>MYSELF</b>	Resident
<i>[immediate word]</i>	
<i>( --- )</i>	
Used in a colon definition. Places the code field address (CFA) of a word into its own definition. This permits recursion.	
<b>NEXT,</b>	Resident
<i>( --- )</i>	
<b>NEXT,</b> should be paired with <b>CODE</b> to surround assembly code or machine code:	
<b>CODE cccc &lt;assembly mnemonics&gt; NEXT,</b>	
<b>NEXT,</b> puts <b>045Fh</b> ( machine code for ALC: <b>B *R15</b> ) at HERE and advances HERE. See <b>ASM:</b> , <b>NEXT,</b> , ; <b>ASM</b> and <b>CODE</b> for more information. See also Chapter 9 “The fbForth 2.0 TMS9900 Assembler”.	
<b>NFA</b>	Resident
<i>( pfa --- nfa )</i>	
Convert the parameter field address <i>pfa</i> of a definition to its name field address <i>nfa</i> .	
<b>NIP</b>	More Useful Stack Words etc. [41]
<i>( n<sub>1</sub> n<sub>2</sub> --- n<sub>2</sub> )</i>	
Remove from the stack the number that is under the top number.	
<b>NIP\$</b>	Stack-based String Library [42]
<i>( --- ) ( SS: str<sub>1</sub> str<sub>2</sub> --- str<sub>2</sub> )</i> “nip string”	
The word <b>NIP\$</b> removes the string underneath the topmost string from the string stack. For example,	
<b>\$" red" ok:0</b> <b>\$" blue" ok:0</b>	
At this point, “blue” is on the top of the string stack, with “red” underneath it.	
<b>NIP\$</b>	

At this point, “red” has been removed from the string stack, leaving “blue” as the topmost string.

**NOP** Resident

( --- )

A do-nothing instruction. **NOP** is useful for patching as in assembly code.

**NULL** [*Literally NUL (ASCII 0)*] [*immediate word*] Resident

( --- )

There is actually no word in **fbForth 2.0** with the name, ‘ **NULL** ’. The name field for **NULL** contains an ASCII 0. Every **fbForth 2.0** buffer, including the terminal input buffer, must end with an ASCII 0. When **INTERPRET** reaches it, it will search for it in the dictionary and will find what we are here calling **NULL**. **NULL** is the only way to exit the endless loop in **INTERPRET**. When **NULL** executes, it drops the top value on the return stack and thus returns, not to **INTERPRET**, but to the word that executed **INTERPRET** (usually **QUIT** or **LOAD**). Here is its definition, keeping in mind that ‘ **NULL** ’ represents an actual NUL (ASCII 0):

```
: NULL BLK @ IF ?EXEC THEN R> DROP ; IMMEDIATE
```

**NUMBER** Resident

( *addr* --- *d* )

Convert a packed character string (see footnote 4 on page 19) left at *addr* with the character count in the first byte, to a signed double number *d*, using the current numeric base. If a decimal point is encountered in the text, its position will be given in **DPL**, but no other effect occurs. If numeric conversion is not possible, an error message will be given.

**OF** [*immediate word*] Resident

Occurs inside a colon definition as part of the **OF ... ENDOF** construct inside of the **CASE ... ENDCASE** construct.

Compile time: ( 4 --- *addr* 5 )

Checks for the value 4 on the stack left there by **CASE** or a previous **ENDOF**, compiles (**OF**), leaves its address *addr* for branching resolution by **ENDOF** and leaves a 5 for its matching **ENDOF** to check.

Runtime: ( *n* --- [ ] | *n* )

The value *n* is compared to the value which was on top of the stack when **CASE**’s runtime action occurred. If the numbers are identical, the words between **OF** and **ENDOF** will be executed. Otherwise, *n* is put back on the stack for execution to continue after **ENDOF**. See **CASE** and **ENDOF**.

**OPN** Resident

( --- )

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

<b>OR</b>	<p>( <math>n_1</math> <math>n_2</math> --- <math>n_3</math> )</p> <p>Leave the bit-wise logical OR <math>n_3</math> of two 16-bit values, <math>n_1</math> and <math>n_2</math>.</p>	Resident
<b>OUT</b>	<p>( --- <i>addr</i> )</p> <p>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.</p>	Resident
<b>OUTPT</b>	<p>( --- )</p> <p>Assigns the attribute OUTPUT to the file whose PAB is pointed to by <b>PAB-ADDR</b> .</p>	Resident
<b>OVER</b>	<p>( <math>n_1</math> <math>n_2</math> --- <math>n_1</math> <math>n_2</math> <math>n_1</math> )</p> <p>Copy the second stack value <math>n_1</math> to the top of the stack.</p>	Resident
<b>OVER\$</b>	<p style="text-align: right;">Stack-based String Library [42]</p> <p>( --- ) ( SS: <math>str_1</math> <math>str_2</math> --- <math>str_1</math> <math>str_2</math> <math>str_1</math> ) “over string”</p> <p>The word <b>OVER\$</b> pushes a copy of the string <math>str_1</math> to the top of the string stack, above <math>str_2</math>. For example,</p> <pre style="margin-left: 40px;">\$" red"  ok:0 \$" green" ok:0 OVER\$  ok:0</pre> <p>At this point, the string stack contains the following strings:</p> <pre style="margin-left: 40px;">"red" (the topmost string) "green" "red"</pre>	
<b>PAB-ADDR</b>	<p>( --- <i>addr</i> )</p> <p>A variable containing the VDP address of the first byte of the current PAB (Peripheral Access Block).</p>	Resident
<b>PAB-BUF</b>	<p>( --- <i>addr</i> )</p> <p>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.</p>	Resident
<b>PAB-VBUF</b>	<p>( --- <i>addr</i> )</p> <p>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-VBUF</b> is also stored in the file's PAB.</p>	Resident

<b>PABS</b>	Resident
( --- <i>addr</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 <b>fbForth 2.0</b> ) above <b>HERE</b> . Every time <b>HERE</b> changes, <b>PAD</b> is updated.	
<b>PAUSE</b>	Resident
( --- <i>flag</i> )	
Checks for a keystroke and issues <i>false</i> if none, <i>true</i> if <b>&lt;BREAK&gt;</b> ( <b>&lt;CLEAR&gt;</b> or <b>&lt;FCTN+4&gt;</b> ) or idles until a second keystroke before issuing <i>false</i> (or <i>true</i> if second keystroke is <b>&lt;BREAK&gt;</b> ). The words <b>LIST</b> , <b>INDEX</b> , <b>DUMP</b> and <b>VLIST</b> all call the word <b>PAUSE</b> . These routines exit when <i>flag</i> = <i>true</i> . <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>PDT</b>	Resident
( --- <i>vaddr</i> )	
A constant which contains the VDP address of the Pattern Descriptor Table. Default value is <b>800h</b> . This constant can only be changed via user variable number <b>28h</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>	Resident
( --- <i>f</i> )	
A floating point approximation of $\pi$ to 13 significant figures. (3.141592653590)	
<b>PICK</b>	More Useful Stack Words etc. [41]
( + <i>n</i> --- [ <i>n</i> ] )	
Copy to the top of the stack the <i>n</i> <sup>th</sup> number down. The 0 <sup>th</sup> number is the top number. [ <i>n</i> ] means “the contents of cell <i>n</i> from the top of the stack”. The number <i>n</i> must be positive.	
<b>0 PICK</b> is equivalent to <b>DUP</b> .	
<b>1 PICK</b> is equivalent to <b>OVER</b> .	
<b>PICK\$</b>	Stack-based String Library [42]
( <i>index</i> --- ) ( SS: --- <i>str</i> ) “pick string”	
Given the index of a string on the string stack, copy the indexed string to the top of the string stack. <b>0 PICK\$</b> is equivalent to <b>DUP\$</b> , <b>1 PICK\$</b> is equivalent to <b>OVER\$</b> etc. For example,	

```

$" blue"  ok:0
$" green" ok:0
$" red"   ok:0
2 PICK$  ok:0

```

The above causes the string “blue” to be copied to the top of the string stack.

**PREV** Resident

( --- *addr* )

A user variable containing the address of the disk buffer most recently referenced. The **UPDATE** command marks this buffer to be later written to disk.

**QUERY** Resident

( --- )

Input 80 characters of text (or until **<ENTER>** is pressed) from the operator’s terminal. Text is positioned at the address contained in **TIB** with **IN** set to 0.

**QUIT** Resident

( --- )

Clear the return stack, stop compilation and return control to the operator’s terminal. No message is given, including the usual **ok:n** .

**R** Resident

( --- *n* ) ( R: *n* --- *n* )

Copy the top of the return stack to the parameter stack.

**R#** Resident

( --- *addr* )

A user variable which may contain the location of an editing cursor or other file-related function.

**R->BASE** Resident

( --- ) ( R: *n* --- )

Restore the current base from the return stack. See **BASE->R** .

**R/W** Resident

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

The fig-Forth standard disk read/write linkage. The only modification to **R/W** for **fbForth 2.0** is that it now calls **RBLK** and **WBLK** instead of the replaced **RDISK** and **WDISK** . The source or destination block buffer address is *addr*, *n*<sub>1</sub> is the sequential number of the referenced block and *flag* indicates whether the operation is write (*flag* = 0) or read (*flag* = 1). **R/W** determines the location on mass storage, performs the read/write and error checking.

**R0** Resident

( --- *addr* )

A user variable containing the initial location of the return stack. Pronounced “r zero”. See **RP!** .

<b>R&gt;</b>	( --- <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> .	Resident
<b>R&gt;F</b>	( --- <i>f</i> ) ( R: <i>f</i> --- ) Moves the 8-byte floating point number <i>f</i> from the return stack to the parameter stack. See <b>F&gt;R</b> .	Resident
<b>RAD/DEG</b>	( --- <i>f</i> ) Constant in floating point format representing degrees/radian = 0.01745329251994.	Resident
<b>RANDOMIZE</b>	( --- ) Creates an unpredictable seed for the random number generator.	Resident
<b>RBLK</b>	( <i>addr blk</i> --- ) Read a block from the current blocks file.	Resident
<b>RD</b>	( --- <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> .	Resident
<b>REC-LEN</b>	( <i>b</i> --- ) Stores the length <i>b</i> of the record for the upcoming write into the appropriate byte in the current PAB.	Resident
<b>REC-NO</b>	( <i>n</i> --- ) Writes a zero-based record number <i>n</i> into the appropriate location in the current PAB.	Resident
<b>REPEAT</b>	<i>[immediate word]</i> Used within a colon-definition in the form: <b>BEGIN ... WHILE ... REPEAT</b> Compile time: ( <i>addr</i> <sub>1</sub> 1 <i>addr</i> <sub>2</sub> 4 --- ) At compile-time, <b>REPEAT</b> processes the <b>0BRANCH</b> offset <i>addr</i> <sub>2</sub> and the offset from <b>HERE</b> to the loop-back address <i>addr</i> <sub>1</sub> , which it stores at the space reserved for it at <i>addr</i> <sub>1</sub> by <b>BEGIN</b> , <i>q.v.</i> The values 1 and 4 are used for error testing.	Resident

RUNTIME: ( --- )

At runtime, **REPEAT** forces an unconditional branch back to just after the corresponding **BEGIN**. See **WHILE** and **BEGIN**.

**REPLACE\$**

Stack-based String Library [42]

( --- pos | -1 ) ( SS: *str*<sub>1</sub> *str*<sub>2</sub> *str*<sub>3</sub> --- *str*<sub>4</sub> | [ *str*<sub>1</sub> *str*<sub>2</sub> ] ) “replace string”

The word **REPLACE\$** searches string *str*<sub>2</sub> for the first occurrence of string *str*<sub>3</sub>. If it is found, it is replaced with the string *str*<sub>1</sub>, the position of *str*<sub>3</sub> within *str*<sub>2</sub> is pushed to the data stack, *str*<sub>1</sub> and *str*<sub>3</sub> are removed from the string stack and the new string *str*<sub>4</sub> is left on the string stack. For example,

```

512 INIT$ ok:0
" PURPLE" ok:0
" redgreenblue" ok:0
" green" ok:0
REPLACE$ . 3 ok:0
$.S
-----+-----+-----
Index|Length|String
-----+-----+-----
  0 |   13 |redPURPLEblue

Allocated stack space: 16 bytes
Total stack space: 512 bytes
Stack space remaining: 496 bytes
ok:0

```

If the search string *str*<sub>3</sub> is not found, -1 is pushed to the data stack, *str*<sub>1</sub> and *str*<sub>2</sub> are left on the string stack, ready for another search.

**RESET\$**

Stack-based String Library [42]

( --- ) ( SS: --- )

Resets, *i.e.*, empties, the string stack.

**REV\$**

Stack-based String Library [42]

( --- ) ( SS: *str*<sub>1</sub> --- *str*<sub>2</sub> ) “reverse string”

The word **REV\$** replaces the topmost string on the string stack with its reversed equivalent. For example,

```
$" green" REV$ .$ neerg ok:0
```

**RIGHT\$**

Stack-based String Library [42]

( *len* --- ) ( SS: *str*<sub>1</sub> --- *str*<sub>1</sub> *str*<sub>2</sub> ) “right of string”

The word **RIGHT\$** causes the rightmost *len* characters to be pushed to the string stack as a new string. The original string is retained. For example,

```
$" redgreenblue" 4 RIGHT$ ok:0
```

The above causes the string “blue” to be pushed to the string stack.

<b>RLTV</b>	Resident
( --- )	
Assigns the attribute RELATIVE to the file whose PAB is pointed to by <b>PAB-ADDR</b> .	
<b>RND</b>	Resident
( $n_1$ --- $n_2$ )	
Generates a positive random integer $n_2$ greater than or equal to 0 and less than $n_1$ .	
<b>RNDW</b>	Resident
( --- $n$ )	
Generates a random word. The value of the word may be positive or negative depending on whether the sign bit is set.	
<b>ROLL</b>	More Useful Stack Words etc. [41]
( [ $n$ ] ... [0] + $n$ --- [ $n-1$ ] ... [0] [ $n$ ] )	
Rotate left the top $n+1$ numbers on the stack, resulting in the $n^{\text{th}}$ number down moving to the top of the stack. The number $n$ must be positive. The source for <b>ROLL</b> was Marshall Linker via George Smyth's "Forth Forum" column in the <i>MANNERS Newsletter</i> (1985) Vol. 4(5), pp. 12 – 16.	
<b>0 ROLL</b> is a null operation. <b>1 ROLL</b> is equivalent to <b>SWAP</b> . <b>2 ROLL</b> is equivalent to <b>ROT</b> .	
<b>ROT</b>	Resident
( $n_1$ $n_2$ $n_3$ --- $n_2$ $n_3$ $n_1$ )	
Rotate the top three values on the stack, bringing the third $n_1$ to the top.	
<b>ROT\$</b>	Stack-based String Library [42]
( --- ) ( SS: $str_1$ $str_2$ $str_3$ --- $str_2$ $str_3$ $str_1$ ) "rotate strings"	
The word <b>ROT\$</b> rotates the top three strings to the left. The third string down (prior to the execution of <b>ROT\$</b> ) moves to the top of the string stack. See Chapter 14 for implementation details regarding stack space limitations.	
<b>RP!</b>	Resident
( --- )	
A procedure to initialize the return stack pointer from user variable <b>R0</b> .	
<b>RSTR</b>	Resident
( $n$ --- )	
Restores the file whose PAB is pointed to by the current PAB to the specified record number $n$ .	
<b>RTRIM\$</b>	Stack-based String Library [42]
( --- ) ( SS: $str_1$ --- $str_2$ ) "trim right of string"	
The word <b>RTRIM\$</b> removes leading spaces from the topmost string. For example,	
<b>\$" hello!        " RTRIM\$ . \$ <u>hello!</u> <u>ok:0</u></b>	

<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>	Resident
( <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>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>	Resident
( --- <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> . This constant can only be changed via user variable number <b>24h</b> .	
<b>SB0</b>	CRU Words [5]
( <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 TMS9900 instruction, <b>SB0 0</b> , to effect setting the bit. See § 11.3 and CRU documentation in the <i>Editor/Assembler Manual</i> for more information.	
<b>SBZ</b>	CRU Words [5]
( <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 TMS9900 instruction, <b>SBZ 0</b> , to effect resetting the bit. See § 11.3 and CRU documentation in the <i>Editor/Assembler Manual</i> for more information.	
<b>SCMP</b>	CPYBLK -- Block Copying Utility [4]
( <i>str</i> <sub>1</sub> <i>str</i> <sub>2</sub> --- -1   0   +1 )	
Compares two strings with leading byte counts pointed to by <i>str</i> <sub>1</sub> and <i>str</i> <sub>2</sub> and leaves the result on the stack: -1, if <i>str</i> <sub>1</sub> < <i>str</i> <sub>2</sub> ; 0, if <i>str</i> <sub>1</sub> = <i>str</i> <sub>2</sub> ; +1, if <i>str</i> <sub>1</sub> > <i>str</i> <sub>2</sub> .	
<b>SCR</b>	Resident
( --- <i>addr</i> )	
A user variable containing the Forth block number most recently referenced by <b>LIST</b> or <b>EDIT</b> .	

<b>SCRFONT</b>	Resident
( --- <i>addr</i> )	
A user variable containing a flag indicating whether <b>FNT</b> should load the default font (flag = 0) or a user-specified font file (flag ≠ 0). This should be set non-zero only after executing <b>USEFFL</b> , <i>q.v.</i> Setting it to 0 will restore the default font the next time <b>FNT</b> is executed.	
<b>SCREEN</b>	Resident
( <i>n</i> --- )	
Changes the display screen color to the color specified <i>n</i> . The foreground (FG) and background (BG) screen colors must be placed in the low-order byte of <i>n</i> , with FG the high-order 4 bits and BG the low-order 4 bits, <i>e.g.</i> , <i>n</i> = 27 ( <b>1Bh</b> ) for black on light yellow. The FG color is only necessary in the text modes.	
<b>SCRN_END</b>	Resident
( --- <i>addr</i> )	
A user variable containing the address <i>addr</i> of the byte immediately following the last byte of the display screen image table to be used as the logical display screen.	
<b>SCRN_START</b>	Resident
( --- <i>addr</i> )	
A user variable containing the address <i>addr</i> of the first byte of the display screen image table to be used as the logical display screen.	
<b>SCRN_WIDTH</b>	Resident
( --- <i>addr</i> )	
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.	
<b>SEED</b>	Resident
( <i>n</i> --- )	
Places a new seed <i>n</i> into the random number generator.	
<b>SET-PAB</b>	Resident
( --- )	
This instruction assumes that <b>PAB-ADDR</b> is set. It then zeroes out the PAB (Peripheral Access Block) pointed to by <b>PAB-ADDR</b> and places the contents of <b>PAB-VBUF</b> into the appropriate word of the PAB. This initializes the PAB.	
<b>SGN</b>	Resident
( <i>n</i> --- -1   0   +1 )	
Returns the sign of <i>n</i> or 0.	

**SIGN** Resident

( *n d* --- *d* )

Stores a minus sign (ASCII 45 or **2Dh**) 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 **#>** removes it from the stack. Must be used between **<#** and **#>**. Using **SIGN** 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 **DABS**. This can be done by pushing *d* to the stack and executing **SWAP OVER DABS** : ( *d* --- *n*  $|d|$  ) prior to **<# ... SIGN ... #>**.

**SIN** Resident

( *f*<sub>1</sub> --- *f*<sub>2</sub> )

Finds the sine *f*<sub>2</sub> of the floating point number *f*<sub>1</sub> on the stack and leaves the result *f*<sub>2</sub> on the stack.

**SLA** Resident

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

Arithmetically shifts the number *n*<sub>1</sub> on the stack *count* bits to the left, leaving the result *n*<sub>2</sub> 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:

**: SLA0 -DUP IF SLA ENDIF ;**

**SLIT** Resident

( --- *addr* )

**SLIT** is similar to **LIT** but acts on strings instead of numbers. **SLIT** places the address *addr* of the string following it on the stack. It modifies the top of the return stack to point to just after the string.

**SMASH** 64-Column Editor [6]

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

The assembly code routine that formats a line of tiny characters. It expects the address *addr*<sub>1</sub> of the line in memory, the number *count*<sub>1</sub> 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 *addr*<sub>2</sub>, a VDP address *vaddr*, and a byte count *count*<sub>2</sub>. See **CLIST** and **CLINE**.

**SMTN** Resident

( --- *vaddr* )

A constant whose value is the VDP address of the Sprite Motion Table. Default value is **780h**. This constant can only be changed via user variable number **26h**.

<b>SMUDGE</b>	Resident
( --- )	
Used during word definition to toggle the smudge bit in the length byte of a definition's name field. This prevents an uncompleted definition from being found during dictionary searches until compilation is completed without error. <b>SMUDGE</b> is simply defined as	
<b>HEX : SMUDGE LATEST 20 TOGGLE ;</b>	
<b>SP!</b>	Resident
( --- )	
A procedure to initialize the parameter stack pointer from <b>S0</b> , the user variable that points to the base of the parameter stack.	
<b>SP@</b>	Resident
( --- <i>addr</i> )	
This word returns the address of the top of the stack as it was before <b>SP@</b> was executed, <i>e.g.</i> ,	
<b>1 2 SP@ @ . . . 2 2 1 ok:0</b>	
<b>SPACE</b>	Resident
( --- )	
Transmit a blank character (ASCII 32  <b>20h</b> ) to the output device.	
<b>SPACES</b>	Resident
( <i>n</i> --- )	
Transmit <i>n</i> blank characters (ASCII 32  <b>20h</b> ) to the output device.	
<b>SPCHAR</b>	Resident
( <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 a character <i>char</i> in the Sprite Descriptor Table to have the pattern composed of the 4 words (cells) on the stack.	
<b>SPDTAB</b>	Resident
( --- <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. This constant can only be changed via user variable number <b>42h</b> .	
<b>SPLIT</b>	Resident
( --- )	
Converts from present display screen mode into standard Split mode configuration.	
<b>SPLIT2</b>	Resident
( --- )	
Converts from present display screen mode into standard Split2 mode configuration.	

<b>SPRCOL</b>	<p>( <i>n spr ---</i> )</p> <p>Changes color of the given sprite number <i>spr</i> to the color <i>n</i> specified.</p>	Resident
<b>SPRDIST</b>	<p>( <i>spr<sub>1</sub> spr<sub>2</sub> --- n</i> )</p> <p>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.</p>	Resident
<b>SPRDISTXY</b>	<p>( <i>dotcol dotrow spr --- n</i> )</p> <p>Places on the stack <i>n</i>, the square of the distance between the point (<i>dotcol, 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.</p>	Resident
<b>SPRGET</b>	<p>( <i>spr --- dotcol dotrow</i> )</p> <p>Returns the dot column <i>dotcol</i> and dot row <i>dotrow</i> position of sprite <i>spr</i>.</p>	Resident
<b>SPRITE</b>	<p>( <i>dotcol dotrow n char spr ---</i> )</p> <p>Defines sprite number <i>spr</i> to have the specified location (<i>dotcol, dotrow</i>), color <i>n</i>, and character pattern <i>char</i>. The size of the sprite will depend on the magnification factor.</p>	Resident
<b>SPRPAT</b>	<p>( <i>char spr ---</i> )</p> <p>Changes the character pattern of a given sprite <i>spr</i> to <i>char</i>.</p>	Resident
<b>SPRPUT</b>	<p>( <i>dotcol dotrow spr ---</i> )</p> <p>Places a given sprite <i>spr</i> at location (<i>dotcol, dotrow</i>).</p>	Resident
<b>SQNTL</b>	<p>( --- )</p> <p>Assigns the attribute SEQUENTIAL to the file whose PAB is pointed to by <b>PAB-ADDR</b>.</p>	Resident
<b>SQR</b>	<p>( <i>f<sub>1</sub> --- f<sub>2</sub></i> )</p> <p>Finds the square root of a floating point number <i>f<sub>1</sub></i> and leaves the result <i>f<sub>2</sub></i> on the stack.</p>	Resident

<b>SRA</b>	Resident
( $n_1$ <i>count</i> --- $n_2$ )	
Arithmetically shifts $n_1$ <i>count</i> bits to the right and leaves the result $n_2$ on the stack. Shifting by <i>count</i> will be modulo 16 except when <i>count</i> = 0, which causes 16 bits to be shifted. To create a word which does not perform a 16-bit shift when <i>count</i> is zero, use the following definition for the same stack contents:	
: <b>SRA0 -DUP IF SRA ENDIF ;</b>	
<b>SRC</b>	Resident
( $n_1$ <i>count</i> --- $n_2$ )	
Performs a circular right shift of <i>count</i> bits on $n_1$ leaving the result $n_2$ on the stack. If <i>count</i> is 0, 16 bits are shifted. To create a word which does not perform a 16-bit shift when <i>count</i> is zero, use the following definition for the same stack contents:	
: <b>SRC0 -DUP IF SRC ENDIF ;</b>	
<b>SRL</b>	Resident
( $n_1$ <i>count</i> --- $n_2$ )	
Performs a logical right shift of <i>count</i> bits on $n_1$ and leaves the result $n_2$ on the stack. If <i>count</i> is 0, 16 bits are shifted. To create a word which does not perform a 16-bit shift when <i>count</i> is zero, use the following definition for the same stack contents:	
: <b>SRL0 -DUP IF SRL ENDIF ;</b>	
<b>SSDT</b>	Resident
( <i>vaddr</i> --- )	
No longer initializes sprites! Use <b>DELALL</b> , <i>q.v.</i> , instead. <b>SSDT</b> 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. See § 6.6.2 “Sprite Initialization” for details.	
<b>STAT</b>	Resident
( --- <i>b</i> )	
Reads the status of the current PAB and returns the status byte <i>b</i> to the stack. See the table in § 8.5 following the explanation of <b>STAT</b> 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.	

**STCR**

CRU Words [5]

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

Performs the TMS9900 STCR instruction. The CRU base address *addr* will be shifted left one bit and stored in workspace register R12 prior to executing the TMS9900 STCR instruction. There will be  $n_1$  bits transferred from the CRU to the stack as  $n_2$ , where the following condition,  $n_1 \leq 15$ , is enforced by  $n_1$  AND 0Fh. If  $n_1 = 0$ , 16 bits will be transferred. For program clarity, you may certainly use  $n_1 = 16$  to transfer 16 bits because  $n_1 = 0$  will be the value actually used by the final machine code. See § 11.3 and CRU documentation in the *Editor/Assembler Manual* for more information.

**SV**

Resident

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

Performs the file I/O save operation. The number of bytes *count* 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 **PAB-ADDR** points, before executing this word.

**SWAP**

Resident

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

Exchange the top two values on the stack.

**SWAP\$**

Stack-based String Library [42]

 $( \text{---} ) ( \text{SS: } str_1 str_2 \text{---} str_2 str_1 )$  “swap string”

The word **SWAP\$** swaps the topmost two strings on the string stack. For example,

```
$" Hello, World!"  ok:0
$" How are you?"  ok:0
SWAP$  ok:0
```

At this point, the string “Hello, World!” is the topmost string on the string stack.

**SWCH**

Printing Routines [19]

 $( \text{---} )$ 

A special purpose word which permits **EMIT** to output characters to an RS232 device rather than to the screen. See **UNSWCH** .

**SWPB**

Resident

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

Reverses the order of the two bytes in  $n_1$  and leaves the new number as  $n_2$ .

**SYS\$**

Resident

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

A user variable that contains the address of the system support entry point.

<b>SYSTEM</b>	Resident
( <i>n</i> --- )	
Calls the system synonyms. You must specify an offset <i>n</i> into a jump table for the routine you wish to call. The offset <i>n</i> must be one of the predefined even numbers. See system Forth block 33 for offsets 0 – 26.	
<b>TAN</b>	Resident
( <i>f</i> <sub>1</sub> --- <i>f</i> <sub>2</sub> )	
Finds the tangent of the floating point number ( <i>f</i> <sub>1</sub> = angle in radians) on the stack and leaves the result <i>f</i> <sub>2</sub> .	
<b>TASK</b>	Resident
( --- )	
A no-operation word or null definition, <b>TASK</b> is the last word defined in the resident Forth vocabulary of <b>fbForth 2.0</b> 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 <b>fbForth 2.0</b> 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 <b>FORGET</b> ting <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>	CRU Words [5]
( <i>addr</i> --- <i>flag</i> )	
<b>TB</b> performs the TMS9900 <b>TB</b> 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 instruction, <b>TB 0</b> , to effect testing the bit. See § 11.3 and CRU documentation in the <i>Editor/Assembler Manual</i> for more information.	
<b>TCHAR</b>	64-Column Editor [6]
( --- <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>	Resident
( --- )	
Converts from present display screen mode into standard Text mode configuration.	
<b>TEXT80</b>	Resident
( --- )	
Converts from present display screen mode into Text80 mode configuration if your computer has that facility.	

<b>THEN</b>	<i>[immediate word]</i>	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 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>		TRACE -- Colon Definition Tracing [18]
	( --- ) Forces colon definitions that follow it to be compiled in such a way that their execution can be traced. Once a routine has been compiled with the <b>TRACE</b> option, it may be executed with or without a trace. To implement a trace, type <b>TRON</b> before execution. To execute without a trace, type <b>TROFF</b> . Colon definitions that have been compiled under the <b>TRACE</b> option must be recompiled under the <b>UNTRACE</b> option to remove the tracing capability. <b>TRACE</b> and <b>UNTRACE</b> can be used alternately to select words to be traced. See <b>TRON</b> , <b>TROFF</b> , <b>UNTRACE</b> and § 5.4 .	
<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.	
<b>TRIAD</b>		Printing Routines [19]
	( <i>blk</i> --- ) Display on the RS232 device the three Forth blocks that include block number <i>blk</i> , beginning with a Forth block evenly divisible by three. Output is suitable for source text records and includes a reference line at the bottom, "fbForth --- a TI-Forth/fig-Forth extension".	
<b>TRIADS</b>		Printing Routines [19]
	( <i>blk</i> <sub>1</sub> <i>blk</i> <sub>2</sub> --- ) May be thought of as a multiple <b>TRIAD</b> , <i>q.v.</i> You must specify a Forth block range. <b>TRIADS</b> will execute <b>TRIAD</b> as many times as necessary to cover that range.	
<b>TRIM\$</b>		Stack-based String Library [42]
	( --- ) ( SS: <i>str</i> <sub>1</sub> --- <i>str</i> <sub>2</sub> ) "trim string" The word <b>TRIM\$</b> removes both leading and trailing spaces from the topmost string. For example, <pre style="margin-left: 40px;">\$"          hello!          " TRIM\$ .\$ <u>hello!</u> <u>ok:0</u></pre>	

<b>TROFF</b>	TRACE -- Colon Definition Tracing [18] ( --- ) Turn off tracing of words compiled with the <b>TRACE</b> option. See <b>TRON</b> , <b>TRACE</b> , <b>UNTRACE</b> and § 5.4 .	
<b>TRON</b>	TRACE -- Colon Definition Tracing [18] ( --- ) Turn on tracing of words compiled with the <b>TRACE</b> option. See <b>TROFF</b> , <b>TRACE</b> , <b>UNTRACE</b> and § 5.4 .	
<b>TRUNC</b>	$( f_1 \text{ --- } f_2 )$ Truncates $f_1$ , leaving the integer portion $f_2$ on the stack.	Resident
<b>TUCK</b>	$( n_1 n_2 \text{ --- } n_2 n_1 n_2 )$ Put a copy of the top number under the top two numbers on the stack.	More Useful Stack Words etc. [41]
<b>TYPE</b>	$( addr \ count \text{ --- } )$ Transmit <i>count</i> characters from <i>addr</i> to the selected output device.	Resident
<b>U</b>	$( \text{--- } n )$ Places the contents <i>n</i> of workspace register UP (R8) on the stack. Register U contains the base address of the user variable area. This is quicker than executing <b>U@</b> , which accomplishes the same thing.	Resident
<b>U*</b>	$( u_1 \ u_2 \text{ --- } ud )$ Leave the unsigned double number product <i>ud</i> of two unsigned numbers, $u_1$ and $u_2$ .	Resident
<b>U.</b>	$( u \text{ --- } )$ Prints an unsigned number <i>u</i> to the output device.	Resident
<b>U.R</b>	$( u \ n \text{ --- } )$ Prints an unsigned number <i>u</i> right justified in a field of width <i>n</i> .	Resident
<b>U/</b>	$( ud \ u_1 \text{ --- } rem \ quot )$ Leave the unsigned remainder <i>rem</i> and unsigned quotient <i>quot</i> from the unsigned double dividend <i>ud</i> and unsigned divisor $u_1$ .	Resident

<b>U0</b>	<p>( --- <i>addr</i> )</p> <p>A user variable that points to the base of the user variable area.</p>	Resident
<b>U&lt;</b>	<p>( <i>u</i><sub>1</sub> <i>u</i><sub>2</sub> --- <i>flag</i> )</p> <p>Leaves a true flag if <i>u</i><sub>1</sub> is less than <i>u</i><sub>2</sub>, else leaves a false flag.</p>	Resident
<b>UCASE\$</b>	<p style="text-align: right;">Stack-based String Library [42]</p> <p>( --- ) ( SS: <i>str</i><sub>1</sub> --- <i>str</i><sub>2</sub> ) “convert to upper case”</p> <p>The word <b>UCASE\$</b> converts all lower case characters in the topmost string to upper case. For example,</p> <p style="text-align: center;"><b>\$" hello world! 1234" UCASE\$ . \$ HELLO WORLD! 1234 ok:0</b></p>	
<b>UCONS\$</b>	<p>( --- <i>addr</i> )</p> <p>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.</p>	Resident
<b>UD.</b>	<p>( <i>ud</i> --- )</p> <p>Prints an unsigned double number <i>ud</i> to the output device.</p>	Resident
<b>UD.R</b>	<p>( <i>ud</i> <i>n</i> --- )</p> <p>Prints an unsigned double number <i>ud</i> right justified in a field of length <i>n</i>.</p>	Resident
<b>UNDRAW</b>	<p>( --- )</p> <p>Sets <b>DMODE</b> to 1. This means that dots are plotted in the off mode.</p>	Resident
<b>UNFORGETTABLE</b> [ <i>sic</i> ]	<p>( <i>addr</i> --- <i>flag</i> )</p> <p>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 <i>is</i> 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 ( <b>TASK</b> ) in the core dictionary such that <b>UNFORGETTABLE</b> [<i>sic</i>] will report false; however, <b>FORGET</b> will still trap that error.</p>	Resident
<b>UNSWCH</b>	<p style="text-align: right;">Printing Routines [19]</p> <p>( --- )</p> <p>Causes the computer to send output to the display screen instead of an RS232 device. See <b>SWCH</b> .</p>	
<b>UNTIL</b>	<p style="text-align: center;">[<i>immediate word</i>]</p> <p>Occurs within a colon-definition in the form:</p>	Resident

**BEGIN ... UNTIL**

Compile time: ( *addr* 1 --- )

**UNTIL** compiles (**OBRANCH**) and an offset from **HERE** to *addr*, which it stores at the space reserved for it at *addr* by **BEGIN**, *q.v.* The value 1 is used for error testing.

Runtime: ( *flag* --- )

**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** TRACE -- Colon Definition Tracing [18]

( --- )

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** Resident

( --- )

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.

**USEBFL** Resident

*[immediate word]*

( --- )

Selects the blocks file from the input stream to be the current blocks file. **USEBFL** is a state-smart word that can be used in either execution or compilation mode.

Usage: **USEBFL DSK1.MYBLOCKS**

**USEFFL** Resident

*[immediate word]*

( --- )

Selects the user-defined font file from the input stream to be the current font file. **USEFFL** is a state-smart word that can be used in either execution or compilation mode. The font file should be 2048 bytes long and define 8-byte character patterns for ASCII characters 0 – 255. It can be shorter than 2048 bytes, but not longer. If the file is found, **SCRFNT**, *q.v.*, will be set to -1 so that the user's font will be loaded the next time **FNT** is executed. If the file is not a "PROGRAM" file or is longer than 2048 bytes, **FNT** will issue an error message and reload the default font.

If the font patterns do not start at byte 0 of the file, as with *TI Writer's* CHARA1 and CHARA2 (offset 6 bytes), the patterns will be illegible. The font editor **FONTEd**, *q.v.*, can be used to change the font file's registration to load properly.

Usage: **USEFFL DSK1.MYFONT**

**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 **6Ch** – **7Eh** for *n*—enough for 10 user variables.

Even if you use odd offsets, storage/retrieval is always on even-address boundaries one byte less. However, **USER** does not check that the definition is within the **80h** size allotted to the user variable table.

**VAL\$**

Stack-based String Library [42]

( --- *d* ) ( SS: *str* --- )

The word **VAL\$** uses **NUMBER** to convert the topmost string on the string stack to a double number *d* (2-cell, 32-bit integer) on the data stack. An error occurs if the string cannot be represented as a double number. An erroneous value (but, without an error report) will result if a convertible number is outside the signed, 32-bit range: -2147483648 – 2147483647. Examples:

```
$" 9900" VAL$ D. 9900 ok:0
$" 9900" VAL$ DROP . 9900 ok:0
$" 1234567890" VAL$ D. 1234567890 ok:0
$" 9.900" VAL$ D. 9.900 ok:0
$" 9.945" $" 1234.0" D. D. 1234.0 994.5 ok:0
```

**VAND**

Resident

( *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.

<b>VCHAR</b>	Resident
( <i>col row count char</i> --- )	
Prints on the display screen a vertical stream of length <i>count</i> of the specified character <i>char</i> . The first character of the stream is located at ( <i>col,row</i> ). Rows and columns are numbered from 0 beginning at the upper left of the display screen.	
<b>VDPMDE</b>	Resident
( --- <i>addr</i> )	
A user variable used by the mode changing words <b>TEXT80</b> , <b>TEXT</b> , <b>GRAPHICS</b> , <b>MULTI</b> , <b>GRAPHICS2</b> , <b>SPLIT</b> and <b>SPLIT2</b> to hold 0 – 6, respectively. <b>VMODE</b> , <i>q.v.</i> , also changes <b>VDPMDE</b> .	
<b>VFILL</b>	Resident
( <i>vaddr count b</i> --- )	
Fills <i>count</i> locations beginning at the given VDP address <i>vaddr</i> with the specified byte <i>b</i> .	
<b>VLIST</b>	Memory Dump Utility [16]
( --- )	
Prints the names of all words defined in the <b>CONTEXT</b> vocabulary. Note that <b>VLIST</b> will display the names of even ill-defined words in the dictionary that cannot be found with ' , <b>-FIND</b> or <b>(FIND)</b> , <i>q.v.</i> , because their smudge bits are set. See <b>SMUDGE</b> and <b>PAUSE</b> .	
<b>VMBR</b>	Resident
( <i>vaddr addr count</i> --- )	
Reads <i>count</i> bytes beginning at the given VDP address <i>vaddr</i> and places them at <i>addr</i> .	
<b>VMBW</b>	Resident
( <i>addr vaddr count</i> --- )	
Writes <i>count</i> bytes from <i>addr</i> into VDP beginning at the given VDP address <i>vaddr</i> .	
<b>VMODE</b>	Resident
( <i>n</i> --- )	
Changes the VDP mode to mode <i>n</i> , corresponding to the values shown in the entry for <b>VDPMDE</b> above.	
<b>VMOVE</b>	Resident
( <i>vaddr<sub>1</sub> vaddr<sub>2</sub> n</i> --- )	
Move a block of <i>n</i> bytes of VRAM from <i>vaddr<sub>1</sub></i> to <i>vaddr<sub>2</sub></i> , all in VRAM, proceeding toward high memory. This is <i>not</i> overlap safe for <i>vaddr<sub>1</sub></i> < <i>vaddr<sub>2</sub></i> .	
<b>VOC-LINK</b>	Resident
( --- <i>addr</i> )	
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 <b>FORGET</b> through multiple vocabularies.	

<b>VOCABULARY</b>	Resident
( --- )	
A defining word used in the form:	
<b>VOCABULARY cccc</b>	
to create a vocabulary definition <b>cccc</b> . Subsequent use of <b>cccc</b> will make it the <b>CONTEXT</b> vocabulary which is searched first by <b>INTERPRET</b> . The sequence <b>cccc DEFINITIONS</b> will also make <b>cccc</b> the <b>CURRENT</b> vocabulary into which new definitions are placed.	
<b>cccc</b> will be so chained as to include all definitions of the vocabulary in which <b>cccc</b> is itself defined. All vocabularies ultimately chain to Forth. By convention, vocabulary names are to be declared <b>IMMEDIATE</b> . See <b>VOC-LINK</b> .	
<b>VOR</b>	Resident
( <i>b vaddr</i> --- )	
Performs a logical OR on the byte at the specified VDP address and the given byte <i>b</i> . The result byte is stored back into the VDP address.	
<b>VRBL</b>	Resident
( --- )	
Assigns the attribute VARIABLE to the file whose PAB is pointed to by <b>PAB-ADDR</b> .	
<b>VSBR</b>	Resident
( <i>vaddr</i> --- <i>b</i> )	
Reads a single byte from the given VDP address <i>vaddr</i> and places its value <i>b</i> on the stack.	
<b>VSBW</b>	Resident
( <i>b vaddr</i> --- )	
Writes a single byte <i>b</i> into the given VDP address <i>vaddr</i> .	
<b>VWTR</b>	Resident
( <i>b n</i> --- )	
Writes the given byte <i>b</i> into the specified VDP write-only register <i>n</i> .	
<b>VXOR</b>	Resident
( <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 (not relevant in <b>fbForth 2.0</b> ). If <b>WARNING</b> = 0, no disk is present and messages will be presented by number ( <b>msg #n</b> ). 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>WBLK</b>	( <i>addr blk ---</i> ) Write a block to the current blocks file.	Resident
<b>WHERE</b>	( <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.	Resident
<b>WHERE</b>	( <i>EDITOR2 Vocabulary</i> ) ( <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.	64-Column Editor [6]
<b>WHILE</b>	<i>[immediate word]</i> Occurs in a colon-definition in the form: <b>BEGIN ... WHILE (true part) ... REPEAT</b> Compile time: ( <i>addr<sub>1</sub> 1 --- addr<sub>1</sub> 1 addr<sub>2</sub> 4</i> ) <b>WHILE</b> emplaces ( <b>0BRANCH</b> ) and leaves <i>addr<sub>2</sub></i> of the reserved offset. The stack values will be resolved by <b>REPEAT</b> . The values 1 and 4 are used for error checking. Runtime: ( <i>flag ---</i> ) <b>WHILE</b> selects conditional execution based on <i>flag</i> . If <i>flag</i> is <i>true</i> (non-zero), <b>WHILE</b> continues execution of the true part through to <b>REPEAT</b> , which then branches back to <b>BEGIN</b> . If <i>flag</i> is <i>false</i> (zero), execution skips to just after <b>REPEAT</b> , exiting the structure.	Resident
<b>WIDTH</b>	( <i>--- addr</i> ) 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 <b>WIDTH</b> . The value may be changed at any time within the above limits.	Resident
<b>WITHIN</b>	( <i>n<sub>1</sub> n<sub>2</sub> n<sub>3</sub> --- flag</i> ) Result <i>flag</i> is true (1) if $n_2 \leq n_1 < n_3$ and false (0) otherwise.	More Useful Stack Words etc. [41]
<b>WLITERAL</b>	<i>[immediate word]</i> ( <i>---</i> ) A compiling word which compiles <b>SLIT</b> and the string which follows <b>WLITERAL</b> into the dictionary. Used in the form: <b>WLITERAL cccc</b>	Resident

<b>WORD</b>		Resident
	( <i>char</i> --- )	
	Read the text characters from the input stream being interpreted until a delimiter <i>char</i> is found, storing the packed character string (see footnote 4 on page 19) beginning at the dictionary buffer <b>HERE</b> . <b>WORD</b> leaves the character count in the first byte followed by the input characters and ends with two or more blanks. Leading occurrences of <i>char</i> are ignored. If <b>BLK</b> is zero, text is taken from the terminal input buffer, otherwise from the disk block stored in <b>BLK</b> . See <b>BLK</b> , <b>IN</b> .	
<b>WRT</b>		Resident
	( <i>count</i> --- )	
	Performs the file I/O write operation. You must specify the number of bytes <i>count</i> to be written.	
<b>XMLLNK</b>		Resident
	( <i>addr</i> --- )	
	Links a Forth program to a routine in ROM or to a routine located in the memory expansion unit. A ROM address <i>addr</i> or XML vector must be specified as in the <i>Editor/Assembler Manual</i> .	
<b>XOR</b>		Resident
	( $n_1$ $n_2$ --- $n_3$ )	
	Leave $n_3$ , the bitwise logical exclusive OR (XOR) of $n_1$ and $n_2$ .	
[	<i>[immediate word]</i>	Resident
	( --- )	
	Used in a colon-definition in the form: : <b>xxxx</b> [ <b>words</b> ] <b>more</b> ;	
	Suspend compilation. The words after [ are executed, not compiled. This allows calculation or compilation exceptions before resuming compilation with ] . See <b>LITERAL</b> and ] .	
[ <b>COMPILE</b> ]	<i>[immediate word]</i>	Resident
	( --- )	
	Used in a colon definition in the form: : <b>xxxx</b> [ <b>COMPILE</b> ] <b>FORTH</b> ;	
	[ <b>COMPILE</b> ] will force the compilation of an immediate definition that would otherwise execute during compilation. The above example will select the Forth vocabulary when <b>xxxx</b> executes rather than at compile time.	
]		Resident
	( --- )	
	Resume compilation to the completion of a colon definition. See [ .	
^		Resident
	( $f_1$ $f_2$ --- $f_3$ )	
	Returns $f_3$ on the stack $f_1$ raised to the $f_2$ power. The operands must be floating point numbers.	

## Appendix E Differences: fbForth 2.0, fbForth 1.0 and TI Forth

This appendix will detail **fbForth 2.0** changes from **fbForth 1.0** and TI Forth. This will include words that have been added, removed, re-purposed and deprecated. All of those words, except those removed, will also be discussed elsewhere in the manual where appropriate, including the **fbForth 2.0** Glossary. Even some of the removed words will be discussed elsewhere as necessary. Words that have been hoisted into the kernel (resident dictionary) will also be discussed.

### E.1 TI Forth Words not in fbForth 2.0

Descriptions of words appearing in the comments here that are part of **fbForth 2.0** may be found in Appendix D “The fbForth 2.0 Glossary”.

!"

(!)"

<b>-64SUPPORT</b>	Now type <b>MENU</b> for options: <b>6 LOAD</b>
<b>-ASSEMBLER</b>	Now type <b>MENU</b> for options: <b>21 LOAD</b>
<b>-BSAVE</b>	Words loaded are now part of resident dictionary.
<b>-CODE</b>	Words loaded are now part of resident dictionary.
<b>-COPY</b>	<b>CPYBLK</b> replaces contents. Now type <b>MENU</b> for options: <b>4 LOAD</b>
<b>-CRU</b>	Now type <b>MENU</b> for options: <b>5 LOAD</b>
<b>-DUMP</b>	Now type <b>MENU</b> for options: <b>16 LOAD</b>
<b>-EDITOR</b>	Words loaded are now part of resident dictionary.
<b>-FILE</b>	Words loaded are now part of resident dictionary.
<b>-FLOAT</b>	Words loaded are now part of resident dictionary.
<b>-GRAPH</b>	Words loaded are now part of resident dictionary.
<b>-GRAPH1</b>	Words loaded are now part of resident dictionary.
<b>-GRAPH2</b>	Words loaded are now part of resident dictionary.
<b>-MULTI</b>	Words loaded are now part of resident dictionary.
<b>-PRINT</b>	Now type <b>MENU</b> for options: <b>19 LOAD</b>
<b>-SPLIT</b>	Words loaded are now part of resident dictionary.
<b>-SYNONYMS</b>	Words loaded are now part of resident dictionary except <b>FORMAT-DISK</b> , which has been removed.

<b>-TEXT</b>	Words loaded are now part of resident dictionary.
<b>-TRACE</b>	Now type <b>MENU</b> for options: <b>18 LOAD</b>
<b>-VDPMODES</b>	Words loaded are now part of resident dictionary.
<b>&gt;ARG</b>	No longer a high-level Forth word
<b>&gt;FAC</b>	No longer a high-level Forth word
<b>ARG</b>	No longer a high-level Forth word
<b>B/BUF\$</b>	User variable no longer used.
<b>B/SCR\$</b>	User variable no longer used.
<b>CHAR-CNT!</b>	No longer a high-level Forth word
<b>CHAR-CNT@</b>	No longer a high-level Forth word
<b>CHK-STAT</b>	No longer a high-level Forth word
<b>CLR-STAT</b>	No longer a high-level Forth word
<b>DDOT</b>	No longer a high-level Forth word
<b>DISK-HEAD</b>	
<b>DISK_HI</b>	User variable no longer used.
<b>DISK_LO</b>	User variable no longer used.
<b>DISK_SIZE</b>	User variable no longer used.
<b>DR0</b>	
<b>DR1</b>	
<b>DR2</b>	
<b>DRIVE</b>	
<b>DTEST</b>	
<b>EDITOR1</b>	No EDITOR1 vocabulary any longer
<b>F.R</b>	Use <b>F.</b> or <b>FFMT.</b> to compose a replacement definition if needed.
<b>FAC</b>	No longer a high-level Forth word
<b>FAC-&gt;S</b>	No longer a high-level Forth word
<b>FAC&gt;</b>	No longer a high-level Forth word
<b>FAC&gt;ARG</b>	No longer a high-level Forth word
<b>FADD</b>	No longer a high-level Forth word
<b>FDIV</b>	No longer a high-level Forth word
<b>FF.</b>	Use <b>FFMT.</b>
<b>FF.R</b>	Use <b>FFMT.</b> to compose a replacement definition if needed.

<b>FMUL</b>	No longer a high-level Forth word
<b>FORMAT-DISK</b>	
<b>FORTH-COPY</b>	
<b>FORTH_LINK</b>	User variable no longer used. Its function is part of <b>FORTH</b> (Forth vocabulary declaration word).
<b>FSUB</b>	No longer a high-level Forth word
<b>GET-FLAG</b>	No longer a high-level Forth word
<b>OFFSET</b>	User variable no longer used.
<b>PUT-FLAG</b>	No longer a high-level Forth word
<b>RDISK</b>	Replaced by <b>RBLK</b> .
<b>S-&gt;FAC</b>	No longer a high-level Forth word
<b>SCOPY</b>	Replaced by <b>CPYBLK</b> .
<b>SCRATCH</b>	Never should have been implemented!
<b>SETFL</b>	No longer a high-level Forth word
<b>SMOVE</b>	Replaced by <b>CPYBLK</b> .
<b>STR</b>	No longer a high-level Forth word
<b>STR.</b>	No longer a high-level Forth word
<b>VAL</b>	No longer a high-level Forth word
<b>WDISK</b>	Replaced by <b>WBLK</b> .

## E.2 fbForth 1.0 Words not in fbForth 2.0

Descriptions of words appearing in the comments here that are part of **fbForth 2.0** may be found in Appendix D “The fbForth 2.0 Glossary”.

<b>&gt;ARG</b>	No longer a high-level Forth word
<b>&gt;FAC</b>	No longer a high-level Forth word
<b>&gt;ROA</b>	No longer needed because the GPL/XML floating point routines that modified the rollout area have been replaced.
<b>ARG</b>	No longer a high-level Forth word
<b>BPOFF</b>	No longer needed
<b>CHAR-CNT!</b>	No longer a high-level Forth word
<b>CHAR-CNT@</b>	No longer a high-level Forth word
<b>CHK-STAT</b>	No longer a high-level Forth word
<b>CLR-STAT</b>	No longer a high-level Forth word
<b>DBF</b>	No longer needed
<b>DDOT</b>	No longer a high-level Forth word
<b>EDITOR1</b>	No EDITOR1 vocabulary any longer
<b>F.R</b>	Use <b>F.</b> or <b>FFMT.</b> to compose a replacement definition if needed.
<b>FAC</b>	No longer a high-level Forth word
<b>FAC-&gt;S</b>	No longer a high-level Forth word
<b>FAC&gt;</b>	No longer a high-level Forth word
<b>FAC&gt;ARG</b>	No longer a high-level Forth word
<b>FADD</b>	No longer a high-level Forth word
<b>FDIV</b>	No longer a high-level Forth word
<b>FF.</b>	Use <b>FFMT.</b>
<b>FF.R</b>	Use <b>FFMT.</b> to compose a replacement definition if needed.
<b>FMUL</b>	No longer a high-level Forth word
<b>FSUB</b>	No longer a high-level Forth word
<b>GET-FLAG</b>	No longer a high-level Forth word
<b>LCT</b>	No longer needed
<b>MGT</b>	No longer needed
<b>PUT-FLAG</b>	No longer a high-level Forth word

---

<b>ROA</b>	No longer needed because the GPL/XML floating point routines that modified the rollout area have been replaced.
<b>ROA&gt;</b>	No longer needed because the GPL/XML floating point routines that modified the rollout area have been replaced.
<b>S-&gt;FAC</b>	No longer a high-level Forth word
<b>SETFL</b>	No longer a high-level Forth word
<b>STR</b>	No longer a high-level Forth word
<b>STR.</b>	No longer a high-level Forth word
<b>TLC</b>	No longer needed
<b>VAL</b>	No longer a high-level Forth word

### E.3 New and Modified Words in fbForth 2.0

This list contains all the new words added since **fbForth 1.0** except for the String Stack Library (see Chapter 14 “The Stack-based String Library”) and the additional words discussed in § 11.4 “Useful Additional Stack Words”. New words have a light gray background and are indented. All of the words in this list are part of the resident dictionary. The words that are not highlighted have been modified by virtue of the fact that they are now part of the resident dictionary. Detailed descriptions of words listed here may be found in Appendix D “The fbForth 2.0 Glossary”.

#### #MOTION

- .BASE**            Display current radix in decimal.
- 0>**                Leaves true *flag* if number on stack is less than 0.
- >DEG**            Converts number on stack from radians to degrees.
- >F**
- >RAD**            Converts number on stack from degrees to radians.

#### ?FLERR

#### APPND

#### ATN

- BANK@**            Returns the contents of the cell in the bank and address on the stack.
- BANKC@**          Returns the contents of the byte in the bank and address on the stack.

#### BEEP

#### BSAVE

- CEIL**             Returns the floating point (FP) integer closest to but > the number on the stack.

#### CHAR

#### CHARPAT

#### CLSE

#### COINC

#### COINCALL

#### COINCXY

#### COLOR

- COLTAB**          Can only change this constant via user variable number **22h**.

#### COS

- DEG/RAD**        FP constant for degrees/radian

<b>DELALL</b>	Use this word instead of <b>SSDT</b> to initialize sprites.
<b>DELSPR</b>	
<b>DLT</b>	
<b>DOT</b>	
<b>DRAW</b>	
<b>DSPLY</b>	
<b>DTOG</b>	
<b>ED@</b>	
<b>EDIT</b>	
<b>EULER_E</b>	FP constant for <i>e</i> .
<b>EXP</b>	
<b>EXP10</b>	Returns 10 raised to the power of the FP number on the stack.
<b>F!</b>	
<b>F*</b>	
<b>F+</b>	
<b>F-</b>	
<b>F-&gt;S</b>	
<b>F-D"</b>	
<b>F.</b>	
<b>F/</b>	
<b>F0&lt;</b>	
<b>F0=</b>	
<b>F&lt;</b>	
<b>F=</b>	
<b>F&gt;</b>	
<b>F&gt;R</b>	Transfers the FP number on the stack to the return stack.
<b>F@</b>	
<b>FABS</b>	Returns absolute value of FP number on stack.
<b>FCONSTANT</b>	Defines an FP constant.
<b>FDROP</b>	
<b>FDUP</b>	
<b>FFMT.</b>	Formats and displays/prints an FP number on the stack.

**FILE****FLERR**

**FLOOR** Returns the floating point (FP) integer closest to but < the number on the stack.

**FMINUS** Negates the FP number on the stack.

**FNT** Loads the current font (default in cartridge ROM or user-specified).

**FONTEd** Starts the new Font Editor.

**FOVER**

**FP1** FP constant for 1.

**FP10** FP constant for 10.

**FPB** Pushes VRAM address of user screen font file PAB to stack.

**FRAC** Returns fractional part of FP number on the stack.

**FROT** Rotates the third FP number on the stack to the top of the stack.

**FSWAP**

**FVARIABLE** Defines an FP variable.

**FXD****GCHAR****GRAPHICS****GRAPHICS2****HCHAR****HONK****INPT****INT****INTRNL****JCRU****JKBD****JMODE****JOYST****LD****LINE**

**LN10INV** FP constant for  $1/\ln(10)$

**LOG**

**LOG10** Returns the decimal logarithm of the FP number on the stack.

<b>MAGNIFY</b>	
<b>MCHAR</b>	
<b>MINIT</b>	
<b>MOTION</b>	
<b>MULTI</b>	
<b>OPN</b>	
<b>OUTPT</b>	
<b>PAB-ADDR</b>	
<b>PAB-BUF</b>	
<b>PAB-VBUF</b>	
<b>PABS</b>	
<b>PDT</b>	Can only change this constant via user variable number <b>28h</b> .
<b>PI</b>	
<b>R&gt;F</b>	Transfers the FP number on the return stack to the stack.
<b>RAD/DEG</b>	FP constant for radians/degree.
<b>RD</b>	
<b>REC-LEN</b>	
<b>REC-NO</b>	
<b>RLTV</b>	
<b>RSTR</b>	
<b>S-&gt;F</b>	
<b>SATR</b>	Can only change this constant via user variable number <b>24h</b> .
<b>SCREEN</b>	
<b>SCRFONT</b>	User variable containing a flag indicating whether to load the default font (zero flag) or a user-defined font (nonzero flag).
<b>SET-PAB</b>	
<b>SIN</b>	
<b>SMTN</b>	Can only change this constant via user variable number <b>26h</b> .
<b>SPCHAR</b>	
<b>SPDTAB</b>	Can only change this constant via user variable number <b>42h</b> .
<b>SPLIT</b>	
<b>SPLIT2</b>	

**SPRCOL****SPRDIST****SPRDISTXY****SPRGET****SPRITE****SPRPAT****SPRPUT****SQNTL****SQR****SSDT**

This word is now optional. Use **DELALL** to initialize sprites.

**STAT****SV****TAN****TEXT****TEXT80****TRUNC**

Returns integer part of FP number on the stack.

**USEFFL**

Sets system to use a user-specified font file.

**UNDRAW****UPDT****VCHAR****VDPMDE****VMODE**

Sets the VDP mode to the mode of the number on the stack (0 – 6)

**VRBL****WHERE****WRT****^**

## Appendix F User Variables in fbForth 2.0

The purpose of this appendix is to detail the User Variables in **fbForth 2.0** 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 **6Ch** through **7Eh** to create his/her own user variables. See the definition of **USER** in Appendix D .

### F.1 fbForth 2.0 User Variables (Address Offset Order)

Name	Offset	Initial Value	Description
<b>UCONS\$</b>	6h	3618h	Base of User Var initial value table
<b>S0</b>	8h	FFA0h	Base of Stack
<b>R0</b>	Ah	3FFEh	Base of Return Stack
<b>U0</b>	Ch	3660h	Base of User Variables
<b>TIB</b>	Eh	FFA0h	Terminal Input Buffer address
<b>WIDTH</b>	10h	31	Name length in dictionary
<b>DP</b>	12h	A000h	Dictionary Pointer
<b>SYS\$</b>	14h	308Ch	Address of System Support
<b>CURPOS</b>	16h	0	Cursor location in VDP RAM
<b>INTLNK</b>	18h	3020h	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	2010h	Beginning of Disk Buffers
<b>LIMIT\$</b>	20h	3020h	End of Disk Buffers
<b>COLTAB</b>	22h	380h	Color Table address in VRAM. <b>COLTAB</b> gets addr.
<b>SATR</b>	24h	300h	Sprite Attribute Table address in VRAM. <b>SATR</b> gets addr.
<b>SMTN</b>	26h	780h	Sprite Motion Table address in VRAM. <b>SMTN</b> gets addr.
<b>PDT</b>	28h	800h	Pattern Descriptor Table addr in VRAM. <b>PDT</b> gets addr.
<b>FPB</b>	2Ah	80h	User font file PAB offset from FRB. <b>FPB</b> gets addr.
<b>DISK_BUF</b>	2Ch	1000h	VDP location of 128B Forth Record Buffer (FRB)
<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	3020h	Interrupt Service Pointer
<b>ALTIN</b>	38h	0	Alternate Input Pointer
<b>ALTOUT</b>	3Ah	0	Alternate Output Pointer
<b>VDPMDE</b>	3Ch	1	VDP Mode
<b>BPB</b>	3Eh	C6h	Blocks PABs offset from FRB. <b>BPB</b> gets address.
<b>BPOFF</b>	40h	0	Current Blocks file offset from <b>BPB</b> . (0 or 70h)
<b>SPDTAB</b>	42h	800h	Sprite Descriptor Table addr in VRAM. <b>SPDTAB</b> gets addr.
<b>SCRFONT</b>	44h	0	Flag for default font (=0) or user-defined font (≠0)
<b>JMODE</b>	46h	0	Flag for whether <b>JOYST</b> executes <b>JKBD</b> (=0) or <b>JCRU</b> (≠0)

Name	Offset	Initial Value	Description
<b>FENCE</b>	<b>48h</b>		Dictionary Fence
<b>BLK</b>	<b>4Ah</b>		Block being interpreted
<b>IN</b>	<b>4Ch</b>		Byte offset in text buffer
<b>OUT</b>	<b>4Eh</b>		Incremented by <b>EMIT</b>
<b>SCR</b>	<b>50h</b>		Last Forth Block (Screen) referenced
<b>CONTEXT</b>	<b>52h</b>		Pointer to Context Vocabulary
<b>CURRENT</b>	<b>54h</b>		Pointer to Current Vocabulary
<b>STATE</b>	<b>56h</b>		Compilation State
<b>BASE</b>	<b>58h</b>		Number Base for Conversions
<b>DPL</b>	<b>5Ah</b>		Decimal Point Location
<b>FLD</b>	<b>5Ch</b>		Field Width (unused)
<b>CSP</b>	<b>5Eh</b>		Stack Pointer for error checking
<b>R#</b>	<b>60h</b>		Editing Cursor location
<b>HLD</b>	<b>62h</b>		Holds address during numeric conversion
<b>USE</b>	<b>64h</b>		Next Block Buffer to Use
<b>PREV</b>	<b>66h</b>		Most recently accessed disk buffer
<b>ECOUNT</b>	<b>68h</b>		Error control
<b>VOC-LINK</b>	<b>6Ah</b>		Vocabulary linkage
<i>[user to define]</i>	<b>6Ch</b>		<i>—available to user—</i>
<i>[user to define]</i>	<b>6Eh</b>		<i>—available to user—</i>
<i>[user to define]</i>	<b>70h</b>		<i>—available to user—</i>
<i>[user to define]</i>	<b>72h</b>		<i>—available to user—</i>
<i>[user to define]</i>	<b>74h</b>		<i>—available to user—</i>
<i>[user to define]</i>	<b>76h</b>		<i>—available to user—</i>
<i>[user to define]</i>	<b>78h</b>		<i>—available to user—</i>
<i>[user to define]</i>	<b>7Ah</b>		<i>—available to user—</i>
<i>[user to define]</i>	<b>7Ch</b>		<i>—available to user—</i>
<i>[user to define]</i>	<b>7Eh</b>		<i>—available to user—</i>

## F.2 fbForth 2.0 User Variables (Variable Name Order)

Name	Offset	Initial Value	Description
ALTIN	38h	0	Alternate Input Pointer
ALTOUT	3Ah	0	Alternate Output Pointer
BASE	58h		Number Base for Conversions
BLK	4Ah		Block being interpreted
BPB	3Eh	C6h	Blocks PABs offset from FRB. <b>BPB</b> gets address.
BPOFF	40h	0	Current Blocks file offset from <b>BPB</b> . (0 or 70h)
C/L\$	1Ch	64	Characters per Line
COLTAB	22h	380h	Color Table address in VRAM. <b>COLTAB</b> gets addr.
CONTEXT	52h		Pointer to Context Vocabulary
CSP	5Eh		Stack Pointer for error checking
CURPOS	16h	0	Cursor location in VDP RAM
CURRENT	54h		Pointer to Current Vocabulary
DISK_BUF	2Ch	1000h	VDP location of 128B Forth Record Buffer (FRB)
DP	12h	A000h	Dictionary Pointer
DPL	5Ah		Decimal Point Location
ECOUNT	68h		Error control
FENCE	48h		Dictionary Fence
FIRST\$	1Eh	2010h	Beginning of Disk Buffers
FLD	5Ch		Field Width (unused)
FPB	2Ah	80h	User font file PAB offset from FRB. <b>FPB</b> gets addr.
HLD	62h		Holds address during numeric conversion
IN	4Ch		Byte offset in text buffer
INTLNK	18h	3020h	Pointer to Interrupt Service Linkage
ISR	36h	3020h	Interrupt Service Pointer
JMODE	46h	0	Flag for whether <b>JOYST</b> executes <b>JKBD</b> (=0) or <b>JCRU</b> (≠0)
LIMIT\$	20h	3020h	End of Disk Buffers
OUT	4Eh		Incremented by <b>EMIT</b>
PABS	2Eh	460h	VDP location for PABs
PDT	28h	800h	Pattern Descriptor Table addr in VRAM. <b>PDT</b> gets addr.
PREV	66h		Most recently accessed disk buffer
R#	60h		Editing Cursor location
R0	Ah	3FFEh	Base of Return Stack
S0	8h	FFA0h	Base of Stack
SATR	24h	300h	Sprite Attribute Table address in VRAM. <b>SATR</b> gets addr.
SCR	50h		Last Forth Block (Screen) referenced
SCRFNT	44h	0	Flag for default font (=0) or user-defined font (≠0)
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
SMTN	26h	780h	Sprite Motion Table address in VRAM. <b>SMTN</b> gets addr.
SPDTAB	42h	800h	Sprite Descriptor Table addr in VRAM. <b>SPDTAB</b> gets addr.
STATE	56h		Compilation State
SYS\$	14h	308Ch	Address of System Support
TIB	Eh	FFA0h	Terminal Input Buffer address

Name	Offset	Initial Value	Description
<b>U0</b>	<b>C</b> h	<b>3660h</b>	Base of User Variables
<b>UCONS\$</b>	<b>6</b> h	<b>3618h</b>	Base of User Var initial value table
<b>USE</b>	<b>64</b> h		Next Block Buffer to Use
<b>VDPMDE</b>	<b>3</b> Ch	<b>1</b>	VDP Mode
<b>VOC-LINK</b>	<b>6</b> Ah		Vocabulary linkage
<b>WARNING</b>	<b>1</b> Ah	<b>1</b>	Message Control
<b>WIDTH</b>	<b>10</b> h	<b>31</b>	Name length in dictionary
[ <i>user to define</i> ]	<b>7E</b> h		—available to user—
[ <i>user to define</i> ]	<b>7C</b> h		—available to user—
[ <i>user to define</i> ]	<b>7A</b> h		—available to user—
[ <i>user to define</i> ]	<b>78</b> h		—available to user—
[ <i>user to define</i> ]	<b>76</b> h		—available to user—
[ <i>user to define</i> ]	<b>74</b> h		—available to user—
[ <i>user to define</i> ]	<b>72</b> h		—available to user—
[ <i>user to define</i> ]	<b>70</b> h		—available to user—
[ <i>user to define</i> ]	<b>6E</b> h		—available to user—
[ <i>user to define</i> ]	<b>6C</b> h		—available to user—

## Appendix G fbForth 2.0 Load Option Directory

The load options are displayed by typing **MENU** . The load options allow you to load only the Forth extensions you wish to use.

You will notice that some of the load options first load other Forth blocks upon which they depend. For example, option, 64-Column Editor, depends on the words loaded by block 13, which displays “loading compact list words” as block 13 starts to load. If, by chance, the prerequisite words were already in the dictionary at the time you type **6 LOAD** , they would not be loaded again. This is called a conditional load. *Note:* As of this writing, the 64-column editor is the only option that does a conditional load of other blocks.

### ***G.1 Option: 64-Column Editor***

Starting screen: 6

Words loaded: **EDIT**            **ED@**            **WHERE**  
                   **CLIST**            **CLINE**

### ***G.2 Option: CPYBLK -- Block Copying Utility***

Starting screen: 4

Words loaded: **SCMP**            **CPYBLK**

### ***G.3 Option: Memory Dump Utility***

Starting screen: 16

Words loaded: **DUMP**            **VLIST**

### ***G.4 Option: TRACE -- Colon Definition Tracing***

Starting screen: 18

Words loaded: **TRACE**            **UNTRACE**        **TRON**  
                   **TROFF**            **:** (*alternate*)

### ***G.5 Option: Printing Routines***

Starting screen: 19

Words loaded: **SWCH**            **UNSWCH**        **?ASCII**  
                   **TRIAD**            **TRIADS**        **INDEX**

### ***G.6 Option: TMS9900 Assembler***

Starting screen: 21

Words loaded: Entire Assembler vocabulary. See Chapter 9.

### ***G.7 Option: CRU Words***

Starting screen: 5

Words loaded: **SB0**            **SBZ**            **TB**  
                  **LDCR**            **STCR**

### ***G.8 Option: Stack-based String Library***

Starting Screen: 42

Words loaded: Entire String Stack Library. See Chapter 14.

### ***G.9 Option: DIR -- Disk Catalog Utility***

Starting screen: 36

Words loaded: **DIR**

### ***G.10 Option: CAT -- Disk Catalog Utility***

Starting screen: 58

Words loaded: **CAT**

## Appendix H Assembly Source for CODEd Words

Several words in FBLOCKS 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>JCRU</b>	—joystick access via the CRU

These words have been coded in hexadecimal in FBLOCKS, thus they do not require that the **fbForth 2.0** Assembler be in memory before they can be loaded. Their Assembly source code (written in **fbForth 2.0** TMS9900 Assembler) is listed on the following pages.

Block 45 needs a little clarification:

1. It should be noted that the definition of **TCHAR** on line 1 is not actually Assembly source code. It is high-level Forth source code. If you wanted to change the character definitions and copy your new table to block 15 of FBLOCKS, you would need to first load the new character definitions. Let's say you have blocks 45 – 47 in a blocks file named MYBLOCKS on DSK1 with your new character definitions for **TCHAR**. This would require loading block 45 of MYBLOCKS to get the definition of **TCHAR** into memory and then copying the contents of **TCHAR** to lines 3 – 9 of block 15 of FBLOCKS. The following code will do the trick:

```

USEBFL DSK1.MYBLOCKS      <== Make MYBLOCKS current
45 LOAD                   <== Load TCHAR
USEBFL DSK1.FBLOCKS      <== Make FBLOCKS current
TCHAR 15 BLOCK 192 + 194 MOVE <== Copy TCHAR to block 15, line 3
FLUSH                     <== Flush block to FBLOCKS
FORGET TCHAR              <== Recover space in dictionary used by
                             TCHAR

```



**BLOCK #42**

```

0 ( Source for CRU words )   BASE->R  HEX
1 ASM: STCR ( n1 addr --- n2 )
2   *SP+ R12 MOV,
   R12 R12 A,
   *SP R1 MOV,
3   R0 CLR,
   R1 000F ANDI,
   R1 R2 MOV,
4   R1 06 SLA,
   R1 3400 ORI,
   R1 X,
5   R2 R2 MOV,
6   NE IF,
7   R02 0008 CI,
8   LTE IF,
9   R0 SWPB,
10  THEN,
11  THEN,
12  R0 *SP MOV,
13 ;ASM
14
15 R->BASE

```

**BLOCK #43**

```

0 ( Source for DDOT )   BASE->R  HEX
1 8040 VARIABLE DTAB 2010 , 0804 , 0201 , 7FBF , DFEF ,
2   F7FB , FDFF , 8040 , 2010 , 0804 , 0201 ,
3 ASM: DDOT ( dotcol dotrow --- b vaddr )
4   *SP+ R1 MOV,
   *SP R3 MOV,
   R1 R2 MOV,
5   R3 R4 MOV,
   R1 0007 ANDI,
   R3 0007 ANDI,
6   R2 00F8 ANDI,
   R4 00F8 ANDI,
   R2 05 SLA,
7   R2 R1 A,
   R4 R1 A,
   R1 2000 AI,
8   R4 CLR,
   DTAB @(R3) R4 MOVB,
9   R4 SWPB,
   R4 *SP MOV,
   SP DECT,
10  R1 *SP MOV,
11 ;ASM
12
13
14
15 R->BASE

```

```

BLOCK #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 ASM: SMASH ( addr #char line# --- lb vaddr cnt )
4   *SP+ R1 MOV,
   *SP+ R2 MOV,
   *SP R3 MOV,
   R4 LB LI,
   R4 *SP MOV,
5   SP DECT,
   R1 SWPB,
   R1 2000 AI,
   R1 *SP MOV,
   R2 R1 MOV,
   R1 INC,
6   R1 FFFE ANDI,
   SP DECT,
   R1 2 SLA,
   R1 *SP MOV,
   R3 R2 A,
7   BEGIN,
   R2 R3 C,
8   GT WHILE,
   R5 CLR,
   R6 CLR,
   *R3+ R5 MOVB,
9   *R3+ R6 MOVB,
   R5 6 SRL,
   R6 6 SRL,
10  BEGIN,
   TC @(R5) R0 MOV,
   TC @(R6) R1 MOV,
   R1 4 SRC,
   R12 4 LI,
11  BEGIN,
   R0 R11 MOV,
   R11 F000 ANDI,
   R1 R7 MOV,
   R7 F00 ANDI,
12  R11 R7 SOC,
   R7 *R4+ MOVB,
   R0 C SRC,
   R1 C SRC,
   R12 DEC,
13  EQ UNTIL,
   R5 INCT,
   R6 INCT,
   R5 R12 MOV,
   R12 2 ANDI,
14  EQ UNTIL,
15  REPEAT,
   ;ASM
                                     R->BASE

```

**BLOCK #45**

```

0 ( definitions of tiny chars with true lowercase) BASE->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 , 4000 , ( * ) 0044 , E440 , ( + )
6 0000 , 0048 , ( , ) 0000 , E000 , ( - ) 0000 , 0004 , ( . )
7 0224 , 4488 , ( / ) 04AA , EAA4 , ( 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 , AE86 , ( @ )
13 04AA , EAAA , ( A ) 0CAA , CAAC , ( B ) 0688 , 8886 , ( C )
14 0CAA , AAAC , ( D ) 0E88 , C88E , ( E ) 0E88 , C888 , ( F )
15 -->

```

**BLOCK #46**

```

0 ( definitions of tiny chars with true lowercase 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 , AAAE , ( O )
4 0CAA , C888 , ( P ) 0EAA , AAEC , ( Q ) 0CAA , CAAA , ( R )
5 0688 , 422C , ( S ) 0E44 , 4444 , ( T ) 0AAA , AAAE , ( 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 , ( ` ) 000E , 2EAE , ( a )
10 088C , AAAC , ( b ) 0006 , 8886 , ( c ) 0226 , AAA6 , ( d )
11 0004 , AE86 , ( e ) 0688 , E888 , ( f ) 0006 , A62C , ( g )
12 088C , AAAA , ( h ) 0404 , 4442 , ( i ) 0202 , 22A4 , ( j )
13 088A , ACAA , ( k ) 0444 , 4444 , ( l ) 000A , EEAA , ( m )
14 0008 , EAAA , ( n ) 0004 , AAA4 , ( o ) 000C , AC88 , ( p )
15 -->

```

**BLOCK #47**

```

0 ( definitions of tiny chars with true lowercase concluded)
1 0006 , A622 , ( q ) 0008 , E888 , ( r ) 0006 , 842C , ( s )
2 044E , 4442 , ( t ) 000A , AAA6 , ( u ) 000A , AAA4 , ( v )
3 000A , AEEA , ( w ) 000A , A4AA , ( x ) 000A , A62C , ( y )
4 000E , 248E , ( z ) 0644 , 8446 , ( { ) 0444 , 0444 , ( | )
5 0C44 , 244C , ( } ) 02E8 , 0000 , ( ~ ) 0EEE , EEEE , ( DEL )
6 R->BASE ;S
7
8
9
10
11
12
13
14
15

```

**BLOCK #48**

```
0 ( Source for JCRU used by JOYST for CRU access to joysticks)
1 BASE->R    HEX
2 ASM: JCRU ( joystick# --- value )
3   *SP R1 MOV,      ( get unit number)
4   R1 5 AI,        ( use keyboard select 6 for #1, 7 for #2)
5   R1 SWPB,
6   R12 24 LI,
7   R1 3 LDCR,
8   R12 6 LI,
9   R1 5 STCR,
10  R1 SWPB,
11  R1 INV,
12  R1 001F ANDI,
13  R1 *SP MOV,
14  83D6 @() CLR,   ( defeat auto screen blanking without KSCAN)
15 ;ASM              R->BASE
```

## Appendix I 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 because 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.								
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.								
5	FBLOCKS not current	This message is displayed when <b>fbForth 2.0</b> needs to read from the system blocks file, FBLOCKS, and the user has made another blocks file current with <b>USEBFL</b> . This is likely the result of executing <b>MENU</b> without FBLOCKS current.								
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.								
8	block # out of range	A block # has been requested from the current blocks file that is less than 1 or greater than the number of blocks in the file.								
9	file I/O error	Any file I/O operation which results in an error will return this message. The high-order 3 bits of the flag/status byte (PAB + 1) contain the error code, which can be obtained with <b>HEX PAB-ADDR @ 1+ VSBR 0E0 AND 5 SRA</b> . An error code of 0 indicates no error only if the COND bit (bit 2) of the GPL status byte located at <b>837Ch</b> is <i>not</i> set.								
<table border="1"> <thead> <tr> <th>code</th> <th>meaning</th> </tr> </thead> <tbody> <tr> <td>00</td> <td>Bad device name</td> </tr> <tr> <td>01</td> <td>Device is write protected</td> </tr> <tr> <td>02</td> <td>Bad open attribute</td> </tr> </tbody> </table>			code	meaning	00	Bad device name	01	Device is write protected	02	Bad open attribute
code	meaning									
00	Bad device name									
01	Device is write protected									
02	Bad open attribute									

<b>Error#</b>	<b>Message</b>	<b>Probable Causes</b>
		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>
10	floating point error	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.
		<b>code meaning</b>
		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
17	compilation only	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 without a <b>LOOP</b> , an <b>IF</b> has no corresponding <b>ENDIF</b> or <b>THEN</b> , <i>etc.</i>
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, an incomplete conditional construct such as <b>: xx IF ;</b> , will trigger this error message.
21	in protected dictionary	An attempt was made to <b>FORGET</b> a word with an address lower than or equal to that of <b>TASK</b> (last word in resident dictionary) or the contents of <b>FENCE</b> if that is higher.
22	use only when loading	This usually means an attempt was made to use <b>--&gt;</b> on the command line.
25	bad jump token	Improper use of jump tokens or conditionals in the <b>fbForth 2.0</b> TMS9900 Assembler.

## Appendix J Contents of FBLOCKS

The contents of the **fbForth 2.0** system blocks file, FBLOCKS, that follow are derived from TI Forth but are in different blocks. Much of this is due to the fact that the blocks are in a file rather than referenced as sectors on a disk. The blocks are also not necessarily in the same order as in TI Forth; however, the TI Forth block (screen) number is indicated as “(old TIF #...)” where applicable. There are also many changes from TI Forth. Many words have been moved to the resident dictionary and some TI Forth words have been removed. There are new words in **fbForth 2.0**, as well. (*cf.* Appendix E “Differences: fbForth 2.0, fbForth 1.0 and TI Forth”)

Note that blocks are numbered from 1 in **fbForth 2.0** rather than 0 as in TI Forth. There are also 23 blank blocks (blocks 39 – 40, 57, 61 – 80), which you can use as you wish.

Note, also, that the following file is dated 19JUN2015.

### BLOCK #1 ( old TIF #3)

```

0 ( fbForth WELCOME SCREEN---LES 19JUN2015)
1 BASE->R HEX
2 : MENU 1 BLOCK 2+ @ 6662 - 5 ?ERROR 2 LOAD ;
3 ." FBLOCKS mod: 19JUN2015"
4 CR CR ." Type MENU for load options." CR CR R->BASE ;S
5
6
7
8
9
10
11
12
13
14
15

```

### BLOCK #2

```

0 CLS 0 0 GOTOXY ." Load Options (19JUN2015)    fbForth 2.0" CR CR
1 ." Description                                Load Block" CR
2 ." -----" CR
3 ." CPYBLK -- Block Copying Utility.....4" CR
4 ." CRU Words.....5" CR
5 ." 64-Column Editor.....6" CR
6 ." Memory Dump Utility.....16" CR
7 ." TRACE -- Colon Definition Tracing....18" CR
8 ." Printing Routines.....19" CR
9 ." TMS9900 Assembler.....21" CR
10 ." More Useful Stack Words etc.....41" CR
11 ." Stack-based String Library.....42" CR
12 ." DIR -- Disk Catalog Utility.....36" CR
13 ." CAT -- Disk Catalog Utility.....58" CR
14 ." TMS9900 Assembler BLOAD (clears RAM).27" CR
15 ." 64-Column Editor BLOAD (clears RAM)..32" CR    -->

```

**BLOCK #3**

```

0  ." String Library BLOAD (clears RAM)....52" CR CR
1  ." Type <block> LOAD to load." CR  ;S
2
3
4
5
6
7
8
9
10
11
12
13
14
15

```

**BLOCK #4 ( old TIF #39)**

```

0 ( Block Copy 10NOV2013 LES ) CR CR
1 ." CPYBLK copies a range of blocks to the same or another file,
2 e.g.," CR CR ." CPYBLK 5 8 DSK1.F1 9 DSK2.F2" CR CR ." will
3 copy blocks 5-8 from DSK1.F1 to" CR ." DSK2.F2 starting at bloc
4 k 9." CR CR BASE->R DECIMAL 0 CLOAD CPYBLK 0 VARIABLE SFL
5 0 VARIABLE DFL 0 CONSTANT XD : SCMP OVER C@ OVER C@ OVER OVER
6 - SGN >R MIN 1+ 0 SWAP 1 DO DROP OVER I + C@ OVER I + C@ - SGN
7 DUP IF LEAVE THEN LOOP R> OVER 0= IF OR ELSE DROP THEN SWAP
8 DROP SWAP DROP ; : GNUM BL WORD HERE NUMBER DROP ; : GBFL HERE
9 0 BFLNAM SWAP ! ; : CPYBLK 1 ' XD ! HERE BPB BPOFF @ + 9 + DUP
10 VSBR 1+ HERE SWAP DUP =CELLS ALLOT VMBR GNUM GNUM OVER OVER >
11 IF SWAP THEN OVER - 1+ >R SFL GBFL GNUM DFL GBFL SFL @ DFL @
12 SCMP 0= IF OVER OVER - DUP 0< SWAP R MINUS > + 2 = IF SWAP R +
13 1- SWAP R + 1- -1 ' XD ! THEN THEN CR R> 0 DO OVER DUP . OVER
14 SFL @ (UB) SWAP BLOCK 2- ! DFL @ (UB) UPDATE FLUSH XD + SWAP
15 XD + SWAP LOOP DROP DROP DUP (UB) DP ! ; R->BASE

```

**BLOCK #5 ( old TIF #88)**

```

0 ( CRU WORDS 12OCT82 LA0 ) 0 CLOAD STCR
1 CR ." loading CRU words"
2 BASE->R HEX
3 CODE SB0 C339 , A30C , 1D00 , NEXT,
4 CODE SBZ C339 , A30C , 1E00 , NEXT,
5 CODE TB C319 , A30C , 04D9 , 1F00 , 1601 , 0599 , NEXT,
6
7 CODE LDCR C339 , A30C , C079 , C039 , 0241 , 000F , 1304 ,
8 0281 , 0008 , 1501 , 06C0 , 0A61 , 0261 , 3000 ,
9 0481 , NEXT,
10
11 CODE STCR C339 , A30C , C059 , 04C0 , 0241 , 000F , C081 ,
12 0A61 , 0261 , 3400 , 0481 , C082 , 1304 , 0282 ,
13 0008 , 1501 , 06C0 , C640 , NEXT,
14
15 CR ." See Manual for usage." CR R->BASE

```

```

BLOCK #6 ( old TIF #22)
0 ( 64 COLUMN EDITOR )
1 0 CLOAD EDITOR2 ( ED@)
2 BASE->R DECIMAL 13 R->BASE CLOAD CLIST
3 BASE->R HEX CR ." loading 64-column editor"
4
5
6 VOCABULARY EDITOR2 IMMEDIATE EDITOR2 DEFINITIONS
7 0 VARIABLE CUR
8 : !CUR 0 MAX 3FF 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
13 : PTR CUR @ SCR @ BLOCK + ;
14 : R/C CUR @ C/L /MOD ; ( --- col row ) R->BASE -->
15

BLOCK #7 ( old TIF #23)
0 ( 64 COLUMN EDITOR ) BASE->R HEX ." ."
1
2 : CINIT
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 -->

BLOCK #8 ( old TIF #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 -->

```

```

BLOCK #9 ( old TIF #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 -->

BLOCK #10 ( old TIF #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 ." BLOCK #" SCR @ BASE->R DECIMAL U.
7 R->BASE CR ;
8 : +SCR SCR @ 1+ DUP SCR ! SCRNO CLIST ;
9 : -SCR SCR @ 1- 1 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

BLOCK #11 ( old TIF #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 -->

```

## BLOCK #12 ( old TIF #28 &amp; 29)

```

0 ( 64 COLUMN EDITOR )                               BASE->R HEX      ." ."
1 : EDT  VDPME @ >R  SPLIT ( 0 1000 040 VFILL) ( 0F 7 VWTR)
2 ( 1000 800 01B VFILL)  CINIT !CUR R/C CGOTOXY
3  DUP DUP SCR ! SCRNO CLIST BEGIN RKEY  CASE 08 OF LEFT ENDOF
4  0C OF -SCR ENDOF  0A OF DOWN ENDOF  03 OF DEL RELINE ENDOF
5  0B OF UP ENDOF   04 OF INS RELINE ENDOF  09 OF RIGHT ENDOF
6  07 OF DELLIN REDRAW ENDOF  06 OF INSLIN REDRAW ENDOF
7  0E OF HOME  ENDOF  02 OF +SCR  ENDOF  16 OF TAB ENDOF
8  0D OF 1 +LIN .CUR PLACE ENDOF 1E OF INSLIN BLNKS REDRAW ENDOF
9  01 OF DELHALF BLNKS RELINE ENDOF  7F OF -TAB ENDOF
10 0F OF 5 0 SPRPAT R> VMODE CLS SCRNO DROP QUIT ENDOF
11 DUP 1F > OVER 7F < AND IF DUP !BLK R/C SWAP DROP DUP SCR @
12 (LINE) ROT CLINE 1 +.CUR ELSE 7 EMIT  ENDIF ENDCASE AGAIN ;
13 FORTH DEFINITIONS      : EDIT EDITOR2 0 EDT ;
14 : WHERE EDITOR2 SWAP 2- EDT ; : ED@ EDITOR2 SCR @ SCRNO EDIT ;
15 CR CR ." See Manual for usage." CR  R->BASE

```

## BLOCK #13 ( old TIF #65)

```

0 ( COMPACT LIST )
1 0 CLOAD CLIST BASE->R  CR ." loading compact list words"
2 DECIMAL 0 VARIABLE TCHAR 382 ALLOT
3 15 BLOCK 192 + 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 , NEXT,
15 R->BASE -->

```

## BLOCK #14 ( old TIF #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 ;      R->BASE      ;S
5
6
7
8
9
10
11
12
13
14
15

```

```

BLOCK #15 ( old TIF #67)
0 ( Tiny character patterns for TCHAR array---compact list for
1 64-column editor---388 bytes, lines 3:0-9:0 below )
2
3
4
5
6
7
8
9
10
11
12
13
14
15

BLOCK #16 ( old TIF #42)
0 ( DUMP ROUTINES 12JUL82 LCT...18NOV2013 LES mod)
1 0 CLOAD VLIST BASE->R HEX CR ." loading memory dump utility"
2 : VM+ VDPMD @ 0= IF + ELSE DROP THEN ;
3 : DUMP8 -DUP
4 IF
5 BASE->R HEX 0 OUT ! OVER 4 U.R 3A EMIT
6 OVER OVER 0 DO
7 DUP @ 0 <# # # # BL HOLD BL HOLD #> TYPE 2+ 2
8 +LOOP DROP 1F 18 VM+ OUT @ - SPACES
9 0 DO
10 DUP C@ DUP 20 < OVER 7E > OR
11 IF DROP 2E ENDIF
12 EMIT 1+
13 LOOP
14 CR R->BASE ENDIF ; -->
15

BLOCK #17 ( old TIF #43)
0 ( DUMP ROUTINES 12JUL82 LCT...01SEP2014 LES mod) ." ."
1 : DUMP CR 00 8 8 VM+ U/ >R SWAP R> -DUP
2 IF 0
3 DO 8 8 VM+ DUMP8 PAUSE IF SWAP DROP 0 SWAP LEAVE ENDIF LOOP
4 ENDIF SWAP DUMP8 DROP ;
5 ( .S has been put in resident dictionary)
6 ( maybe should put VLIST there, as well)
7 : VLIST 80 OUT ! CONTEXT @ @ 0 SWAP ( start counter)
8 BEGIN DUP 2 BANKC@ 3F AND OUT @ + SCRN_WIDTH @ 3 - >
9 IF CR 0 OUT ! ENDIF
10 DUP ID.
11 SWAP 1+ SWAP ( increment counter)
12 PFA LFA 2 BANK@ SPACE DUP 0= PAUSE OR
13 UNTIL DROP CR . ." words listed" ; R->BASE
14
15

```

```

BLOCK #18 ( old TIF #44 )
0 ( TRACE COLON WORDS-FORTH DIMENSIONS III/2 P.58 260CT82 LCT)
1 0 CLOAD (TRACE)      CR ." loading colon definition tracing "
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

```

```

BLOCK #19 ( old TIF #72)
0 ( ALTERNATE I/O SUPPORT FOR RS232 PNTR 12JUL82 LCT...mod LES)
1 0 CLOAD INDEX      CR ." loading printing routines"
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 * 1+ 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 ." fbForth --- a TI-Forth/fig-Forth extension" 0C EMIT
15 ENDIF UNSWCH ;          R->BASE -->

```

```

BLOCK #20 ( old TIF #73)
0 ( SMART TRIADS AND INDEX 15SEP82 LA0...mod LES )
1 BASE->R DECIMAL      ." ."
2 : TRIADS ( from to --- )
3 3 / 3 * 2+ SWAP 3 / 3 * 1+ DO I TRIAD 3 +LOOP ;
4 : INDEX ( from to --- ) 1+ SWAP
5 DO I DUP ?ASCII IF CR 4 .R 2 SPACES I BLOCK 64 TYPE ELSE DROP
6     ENDIF PAUSE IF LEAVE ENDIF LOOP ; R->BASE ;S
7
8
9
10
11
12
13
14
15

```

## BLOCK #21 ( old TIF #75)

```

0 ( ASSEMBLER 12JUL82 LCT-LES12DEC2013) 0 CLOAD A$$M BASE->R HEX
1 ASSEMBLER DEFINITIONS CR ." loading TMS9900 assembler" CR ." "
2 : GOP' OVER DUP 1F > SWAP 30 < AND IF + , , ELSE + , ENDIF ;
3 : GOP <BUILDS , DOES> @ GOP' ;
4 0440 GOP B,      0680 GOP BL,      0400 GOP BLWP,
5 04C0 GOP CLR,    0700 GOP SET0,    0540 GOP INV,
6 0500 GOP NEG,    0740 GOP ABS,      06C0 GOP SWPB,
7 0580 GOP INC,    05C0 GOP INCT,    0600 GOP DEC,
8 0640 GOP DECT,   0480 GOP X,
9 : GROU <BUILDS , DOES> @ SWAP 40 * + GOP' ;
10 2000 GROU COC,  2400 GROU CZC,  2800 GROU XOR,
11 3800 GROU MPY,  3C00 GROU DIV,  2C00 GROU XOP,
12 : GGOP <BUILDS , DOES> @ SWAP DUP DUP 1F > SWAP 30 < AND
13     IF 40 * + SWAP >R GOP' R> , ELSE 40 * + GOP' ENDIF ;
14 A000 GGOP A,   B000 GGOP AB,   8000 GGOP C,   9000 GGOP CB,
15 6000 GGOP S,   7000 GGOP SB,   E000 GGOP SOC,  F000 GGOP SOCB, -->

```

## BLOCK #22 ( old TIF #76)

```

0 ( ASSEMBLER 12JUL82 LCT) ." ."
1 4000 GGOP SZC,  5000 GGOP SZCB,  C000 GGOP MOV,  D000 GGOP MOVB,
2 : 0OP <BUILDS , DOES> @ , ;
3 0340 0OP IDLE,  0360 0OP RSET,  03C0 0OP CKOF,
4 03A0 0OP CKON,  03E0 0OP LREX,  0380 0OP RTWP,
5 : ROP <BUILDS , DOES> @ + , ; 02C0 ROP STST,  02A0 ROP STWP,
6 : IOP <BUILDS , DOES> @ , , ; 02E0 IOP LWPI,  0300 IOP LIM1,
7 : RIOP <BUILDS , DOES> @ ROT + , , ; 0220 RIOP AI,
8 0240 RIOP ANDI,  0280 RIOP CI,  0200 RIOP LI,  0260 RIOP ORI,
9 : RCOP <BUILDS , DOES> @ SWAP 10 * + + , ;
10 0A00 RCOP SLA,  0800 RCOP SRA,  0B00 RCOP SRC,  0900 RCOP SRL,
11 : DOP <BUILDS , DOES> @ SWAP 00FF AND OR , ;
12 1300 DOP JEQ,  1500 DOP JGT,  1B00 DOP JH,  1400 DOP JHE,
13 1A00 DOP JL,  1200 DOP JLE,  1100 DOP JLT,  1000 DOP JMP,
14 1700 DOP JNC,  1600 DOP JNE,  1900 DOP JNO,  1800 DOP JOC,
15 1C00 DOP JOP,  1D00 DOP SB0,  1E00 DOP SBZ,  1F00 DOP TB, -->

```

## BLOCK #23 ( old TIF #77)

```

0 ( ASSEMBLER 12JUL82 LCT) ." ." CR ." "
1 : GCOP <BUILDS , DOES> @ SWAP 000F AND 040 * + GOP' ;
2 3000 GCOP LDCR,  3400 GCOP STCR,
3 00 CONSTANT R0  01 CONSTANT R1  02 CONSTANT R2  03 CONSTANT R3
4 04 CONSTANT R4  05 CONSTANT R5  06 CONSTANT R6  07 CONSTANT R7
5 08 CONSTANT R8  09 CONSTANT R9  0A CONSTANT R10  0B CONSTANT R11
6 0C CONSTANT R12  0D CONSTANT R13  0E CONSTANT R14
7 0F CONSTANT R15  08 CONSTANT UP  09 CONSTANT SP  0A CONSTANT W
8 0D CONSTANT IP  0E CONSTANT RP  0F CONSTANT NEXT
9 : @() 020 ; : *? 010 + ; : *?+ 030 + ; : @(?) 020 + ;
10 : @(R0) R0 @(?) ; : *R0 R0 *? ; : *R0+ R0 *?+ ;
11 : @(R1) R1 @(?) ; : *R1 R1 *? ; : *R1+ R1 *?+ ;
12 : @(R2) R2 @(?) ; : *R2 R2 *? ; : *R2+ R2 *?+ ;
13 : @(R3) R3 @(?) ; : *R3 R3 *? ; : *R3+ R3 *?+ ;
14 : @(R4) R4 @(?) ; : *R4 R4 *? ; : *R4+ R4 *?+ ;
15 : @(R5) R5 @(?) ; : *R5 R5 *? ; : *R5+ R5 *?+ ; -->

```

## BLOCK #24 ( old TIF #78)

```

0 ( ASSEMBLER 12JUL82 LCT)                ." ."
1 : @(R6) R6 @(?); : *R6 R6 *? ; : *R6+ R6 *?+ ;
2 : @(R7) R7 @(?); : *R7 R7 *? ; : *R7+ R7 *?+ ;
3 : @(R8) R8 @(?); : *R8 R8 *? ; : *R8+ R8 *?+ ;
4 : @(R9) R9 @(?); : *R9 R9 *? ; : *R9+ R9 *?+ ;
5 : @(R10) R10 @(?); : *R10 R10 *? ; : *R10+ R10 *?+ ;
6 : @(R11) R11 @(?); : *R11 R11 *? ; : *R11+ R11 *?+ ;
7 : @(R12) R12 @(?); : *R12 R12 *? ; : *R12+ R12 *?+ ;
8 : @(R13) R13 @(?); : *R13 R13 *? ; : *R13+ R13 *?+ ;
9 : @(R14) R14 @(?); : *R14 R14 *? ; : *R14+ R14 *?+ ;
10 : @(R15) R15 @(?); : *R15 R15 *? ; : *R15+ R15 *?+ ;
11 : @(UP) UP @(?); : *UP UP *? ; : *UP+ UP *?+ ;
12 : @(SP) SP @(?); : *SP SP *? ; : *SP+ SP *?+ ;
13 : @(W) W @(?); : *W W *? ; : *W+ W *?+ ;
14 : @(IP) IP @(?); : *IP IP *? ; : *IP+ IP *?+ ;
15 -->

```

## BLOCK #25 ( old TIF #79)

```

0 ( ASSEMBLER 12JUL82 LCT)                ." ."
1 : @(RP) RP @(?); : *RP RP *? ; : *RP+ RP *?+ ;
2 : *NEXT+ NEXT *?+ ; : *NEXT NEXT *? ; : @(NEXT) NEXT @(?);
3 : @@ @() ; : ** *? ; : *+ *?+ ; : () @(?); ( Wycove syntax)
4
5 ( DEFINE JUMP TOKENS )
6 : GTE 1 ; : H 2 ; : NE 3 ; : L 4 ; : LTE 5 ; : EQ 6 ;
7 : OC 7 ; : NC 8 ; : OO 9 ; : HE 0A ; : LE 0B ; : NP 0C ;
8 : LT 0D ; : GT 0E ; : NO 0F ; : OP 10 ;
9 : CJMP ?EXEC
10     CASE LT OF 1101 , 0 ENDOF     GT OF 1501 , 0 ENDOF
11     NO OF 1901 , 0 ENDOF     OP OF 1C01 , 0 ENDOF
12     DUP 0< OVER 10 > OR IF 19 ERROR ENDF DUP
13     ENDCASE 100 * 1000 + , ;
14 : IF, ?EXEC [COMPILE] CJMP HERE 2- 42 ; IMMEDIATE
15 -->

```

## BLOCK #26 ( old TIF #80)

```

0 ( ASSEMBLER 12JUL82 LCT)                ." ."
1 : ENDIF, ?EXEC
2     42 ?PAIRS HERE OVER - 2- 2 / SWAP 1+ C! ; IMMEDIATE
3 : ELSE, ?EXEC 42 ?PAIRS 0 [COMPILE] CJMP HERE 2- SWAP 42
4     [COMPILE] ENDIF, 42 ; IMMEDIATE
5 : BEGIN, ?EXEC HERE 41 ; IMMEDIATE
6 : UNTIL, ?EXEC SWAP 41 ?PAIRS [COMPILE] CJMP HERE - 2 / 00FF
7     AND HERE 1- C! ; IMMEDIATE
8 : AGAIN, ?EXEC 0 [COMPILE] UNTIL, ; IMMEDIATE
9 : REPEAT, ?EXEC >R >R [COMPILE] AGAIN,
10     R> R> 2- [COMPILE] ENDIF, ; IMMEDIATE
11 : WHILE, ?EXEC [COMPILE] IF, 2+ ; IMMEDIATE
12 ( : NEXT, *NEXT B, ; ) ( <--now in kernel )
13 : RT, R11 ** B, ; ( RT pseudo-instruction )
14 : THEN, [COMPILE] ENDIF, ; IMMEDIATE ( ENDIF, synonym )
15 FORTH DEFINITIONS : A$$M ; R->BASE

```

**BLOCK #27**

```
0 ( TMS9900 Assembler BLOAD)
1 ." loading TMS9900 assembler "
2 BASE->R 28 R->BASE BLOAD
3 : BLERR IF ." BLOAD error!" THEN ; BLERR FORGET BLERR
4 FORTH DEFINITIONS ;S
5
6
7
8
9
10
11
12
13
14
15
```

**BLOCK #28 – BLOCK #31 TMS9900 Assembler Binary**

```
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
```

**BLOCK #32**

```
0 ( 64-Column Editor BLOAD)
1 ." loading 64-column editor "
2 BASE->R 33 R->BASE BLOAD
3 : BLERR IF ." BLOAD error!" THEN ; BLERR FORGET BLERR
4 FORTH DEFINITIONS ;S
5
6
7
8
9
10
11
12
13
14
15
```

**BLOCK #33 – BLOCK #35 64-Column Editor Binary**

```

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

```

**BLOCK #36**

```

0 ( TurboForth [MRW] Disk Catalog Utility..mod 19JUN2015 LES)
1 0 CLOAD DIR 0 CLOAD CAT CR ." loading DIR catalog utility"
2 BASE->R HEX 0 VARIABLE CatRec 24 ALLOT
3 1152 @ CatRec OVER 46 + FILE Cat 0 VARIABLE Total
4 0 VARIABLE FCount 0 VARIABLE LC 0 VARIABLE bpr
5 0 VARIABLE sect 0 VARIABLE prot 0B10 VARIABLE Tabs 1C00 ,
6 : Tab ( n --- ) Tabs + C@ CURPOS @ SCRN_WIDTH @ / GOTOXY ;
7 : @R100 9 * CatRec DUP C@ + 2+ + PAD 8 CMOVE PAD F@ F->S ;
8 : DskInfo RD DROP CR CatRec COUNT ." Disk Name: " TYPE CR
9 ." Total: " 1 @R100 DUP U.
10 ." Free: " 2 @R100 DUP U. ." Used: " - U. CR ;
11 : Ftype ( ftype --- ) 2 @R100 bpr ! CASE 1 OF ." DIS/FIX"
12 ENDOF 2 OF ." DIS/VAR" ENDOF 3 OF ." INT/FIX" ENDOF 4 OF
13 ." INT/VAR" ENDOF 5 OF ." PROGRAM" 0 bpr ! ENDOF
14 ." ????????" 0 bpr ! ENDCASE
15 bpr @ -DUP IF 4 U.R THEN ; R->BASE -->

```

**BLOCK #37**

```

0 BASE->R DECIMAL ." ."
1 : Head1 ( --- ) ." ----- ---- ----- --- -" CR ;
2 : Head ( --- ) ." Name Size Type B/R P" CR Head1 ;
3 : DoDIR 0 LC ! 0 Total ! 0 FCount ! Head BEGIN
4 LC @ 20 MOD 19 = IF KEY DROP CR Head THEN RD DROP
5 CatRec COUNT DUP WHILE TYPE
6 1 @R100 DUP 1- sect ! DUP 0 Tab 4 U.R Total +! 0 @R100 DUP
7 prot ! ABS 1 Tab Ftype prot @ 0< IF 2 Tab ." Y" THEN CR
8 1 LC +! 1 FCount +! REPEAT
9 DROP DROP Head1
10 FCount @ . ." files" 0 Tab Total @ 4 U.R ." sectors" CR ;
11 R->BASE -->
12
13
14
15

```

**BLOCK #38**

```

0 BASE->R                                ." ."
1 : DIR
2   Cat SET-PAB      ( Initialize PAB skeleton)
3   INTRNL FXD  RLTV  INPT  38 REC-LEN
4   ( Get directory name from input stream)
5   PAB-ADDR @ 10 + 32 WORD HERE COUNT >R SWAP R VMBW R> N-LEN!
6   ( Get the catalog and display it)
7   OPN            ( open the catalog)
8   DskInfo       ( display disk info)
9   DoDIR         ( display file list)
10  CLSE          ( close the catalog) ;
11 R->BASE  CR
12 ." DIR - Catalogs a disk." CR
13 ." E.g., DIR DSK1." CR
14
15

```

**BLOCK #39 – BLOCK #40**

```

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

```

**BLOCK #41**

```

0 ( Useful words--most are required by fbForth String Library)
1 ( written by Mark Wills, Lee Stewart & Marshall Linker)
2 0 CLOAD $.      CR ." Loading useful additional words--" CR
3 ." 2DUP 2DROP NIP TUCK CELLS -ROT PICK ROLL WITHIN <> $. EXIT"
4 : 2DUP ( a b -- a b a b ) OVER OVER ;
5 : 2DROP ( a b -- ) DROP DROP ; : NIP ( a b -- b ) SWAP DROP ;
6 : TUCK ( a b -- b a b ) SWAP OVER ; : CELLS ( n -- 2n ) 2 * ;
7 : -ROT ( a b c -- c a b ) ROT ROT ;
8 : PICK ( +n -- [n]) 1+ CELLS SP@ + @ ;
9 ( The source for ROLL was Marshall Linker via
10 George Smyth's Forth Forum)
11 : ROLL ( [n]..[0] +n -- [n-1]..[0][n] )
12   -DUP IF 1- SWAP >R MYSELF R> SWAP THEN ;
13 : WITHIN ( n low high -- true|false ) OVER - >R - R> U< ;
14 : <> ( a b -- 1|0 ) = 0= ; : $. BASE->R HEX U. R->BASE ;
15 : EXIT ( -- ) [COMPILE] ;S ; IMMEDIATE

```

## BLOCK #42

```

0 ( Portable, Stack Based String Library for fbForth V2.0 )
1 ( V 1.0 - Mark Wills Sept 2014.)
2 ( Ported from the original TurboForth code by Mark Wills )
3 ( Modified by Lee Stewart October 2014)
4 BASE->R DECIMAL 41 R->BASE CLOAD $.
5 CR ." Loading String Library"
6 0 CONSTANT ($sSize)
7 HERE CONSTANT ($sEnd)
8 ($sEnd) VARIABLE ($sp)
9 0 VARIABLE ($temp1)
10 0 VARIABLE ($depth)
11 0 VARIABLE ($temp0)
12 0 VARIABLE ($temp2)
13 0 VARIABLE ($temp3)      -->
14
15

```

## BLOCK #43

```

0 ( Throw codes for string library, mod: Lee Stewart)
1 BASE->R DECIMAL          ." ."
2 : (throw) ( code -- )
3   CASE
4     ($sSize) 0= IF DROP 9999 THEN
5     9900 OF ." String stack underflow" ENDOF
6     9901 OF ." String too large to assign" ENDOF
7     9902 OF ." String stack is empty" ENDOF
8     9903 OF ." Need at least 2 strings on string stack" ENDOF
9     9904 OF ." String too large for string constant" ENDOF
10    9905 OF ." Illegal LEN value" ENDOF
11    9906 OF ." Need at least 3 strings on string stack" ENDOF
12    9908 OF ." Illegal start value" ENDOF
13    9999 OF ." String stack not initialized" ENDOF
14  ENDCASE
15  CR ABORT ;              R->BASE  -->

```

## BLOCK #44

```

0 ( String stack words, mod: Lee Stewart [INIT$ added])
1 : ($depth+) ( -- ) 1 ($depth) +! ;    BASE->R DECIMAL ." ."
2 : ($sp@) ( -- addr ) ($sp) @ ;
3 : ($rUp) ( n -- n|n+1 ) 1+ -2 AND ;
4 : cell+ ( n -- n+2 ) COMPILE 2+ ; IMMEDIATE
5 : (sizeOf$) ( $addr - $size ) @ ($rUp) cell+ ;
6 : (set$SP) ( $size -- ) MINUS DUP ($sp@) + ($sEnd)
7   < IF 9900 (throw) THEN ($sp) +! ;
8 : (addrOf$) ( index -- addr ) ($sp@) SWAP DUP IF 0 DO
9   DUP (sizeOf$) + LOOP ELSE DROP THEN ;
10 : (lenOf$) ( $addr -- len )
11   STATE @ IF COMPILE @ ELSE @ THEN ; IMMEDIATE
12 : INIT$ ( stack_size -- ) ' ($sSize) ! HERE ' ($sEnd) !
13   ($sEnd) ($sSize) + ($sp) ! ($sSize) ALLOT ;
14 : RESET$ ( -- ) 0 ($depth) ! ($sEnd) ($sSize) + ($sp) ! ;
15 : DEPTH$ ( -- $sDepth ) ($depth) @ ;    R->BASE  -->

```

## BLOCK #45

```

0 ( String constant words etc.)          BASE->R DECIMAL      ." ."
1 : $CONST ( max_len tib:"name" -- ) ( runtime: -- $Caddr)
2   <BUILDS ($rUp) DUP , 0 , ALLOT DOES> NOP ;
3 : CLEN$ ( $Caddr -- len ) cell+ @ ;
4 : MAXLEN$ ( $Caddr -- max_len ) (lenOf$) ;
5 : .$CONST ( $Caddr -- ) cell+ DUP (lenOf$)
6   SWAP cell+ SWAP TYPE ;
7 : := ( $Caddr tib:"string" -- ) DUP @ 34 WORD HERE COUNT
8   SWAP >R 2DUP < IF 9901 (throw) THEN NIP 2DUP SWAP cell+
9   ! >R [ 2 CELLS ] LITERAL + R> R> -ROT CMOVE ;
10 : ($") ( addr len -- ) ( ss: -- str ) DUP ($rUp) cell+ (set$SP)
11   DUP ($sp@) ! ($sp@) cell+ SWAP CMOVE ($depth+) ;
12 : (COMPILE$) ( addr len -- ) DUP >R PAD SWAP CMOVE HERE 6 CELLS
13   COMPILE LIT + , COMPILE LIT R , COMPILE BRANCH HERE R
14   ($rUp) + HERE - 2+ , PAD 12 - R HERE SWAP CMOVE R> ($rUp)
15   ALLOT COMPILE ($) ;                      R->BASE  -->

```

## BLOCK #46

```

0 ( String stack words)                  BASE->R DECIMAL      ." ."
1 : $" 34 WORD HERE COUNT STATE @ IF (COMPILE$) ELSE ($") THEN ;
2   IMMEDIATE : >$ cell+ DUP (lenOf$) SWAP cell+ SWAP ($) ;
3 : PICK$ DEPTH$ 0= IF 9902 (throw) THEN
4   (addrOf$) DUP (lenOf$) SWAP cell+ SWAP ($) ;
5 : DUP$ DEPTH$ 0= IF 9902 (throw) THEN 0 PICK$ ;
6 : DROP$ DEPTH$ 0= IF 9902 (throw) THEN
7   ($sp@) (sizeOf$) MINUS (set$SP) -1 ($depth) +! ;
8 : SWAP$ DEPTH$ 2 < IF 9903 (throw) THEN ($sp@) DUP (sizeOf$)
9   HERE SWAP CMOVE 1 (addrOf$) DUP (sizeOf$) ($sp@) SWAP CMOVE
10  HERE DUP (sizeOf$) ($sp@) DUP (sizeOf$) + SWAP CMOVE ;
11 : NIP$ DEPTH$ 2 < IF 9903 (throw) THEN SWAP$ DROP$ ;
12 : OVER$ DEPTH$ 2 < IF 9903 (throw) THEN 1 PICK$ ;
13 : (rot$) ($sp@) 3 (addrOf$) ($sp@) (sizeOf$)
14   1 (addrOf$) (sizeOf$) 2 (addrOf$) (sizeOf$) + + CMOVE
15   3 (addrOf$) ($sp) ! -3 ($depth) +! ;      R->BASE  -->

```

## BLOCK #47

```

0 ( String stack words)                  BASE->R DECIMAL      ." ."
1 : ROT$ DEPTH$ 3 < IF 9906 (throw) THEN
2   1 PICK$ 1 PICK$ 4 PICK$ (rot$) ;
3 : -ROT$ DEPTH$ 3 < IF 9906 (throw) THEN
4   0 PICK$ 3 PICK$ 3 PICK$ (rot$) ;
5 : LEN$ DEPTH$ 1 < IF 9902 (throw) THEN ($sp@) @ ;
6 : >$CONST >R DEPTH$ 1 < IF 9902 (throw) THEN LEN$ R @ > IF 9904
7   (throw) THEN ($sp@) DUP (sizeOf$) R> cell+ SWAP CMOVE DROP$ ;
8 : +$ DEPTH$ 2 < IF 9903 (throw) THEN 1 (addrOf$) cell+ HERE 1
9   (addrOf$) (lenOf$) CMOVE ($sp@) cell+ 1 (addrOf$) (lenOf$) HERE
10  + LEN$ CMOVE HERE LEN$ 1 (addrOf$) (lenOf$) + DROP$ DROP$ ($) ;
11 : MID$ DEPTH$ 1 < IF 9902 (throw) THEN DUP LEN$ > OVER 1 < OR
12   IF 9905 (throw) THEN OVER DUP LEN$ > SWAP 0< OR IF 9908
13   (throw) THEN SWAP ($sp@) cell+ + SWAP ($) ;
14 : LEFT$ DEPTH$ 1 < IF 9902 (throw) THEN DUP LEN$ > OVER 1 < OR
15 IF 9905 (throw) THEN 0 ($sp@) cell+ + SWAP ($) ; R->BASE  -->

```



**BLOCK #51**

```

0 ( String stack words)          BASE->R DECIMAL      ." ." CR
1 : $.S CR DEPTH$ 0 > IF ($sp@) DEPTH$ ." Index|Length|String"
2   CR ." -----+-----+-----" CR 0 BEGIN DEPTH$ 0 > WHILE DUP
3   6 .R ." |" LEN$ 6 .R ." |" .$ 1+ CR REPEAT DROP ($depth) !
4   ($sp) ! CR ELSE ." String stack is empty." CR THEN
5   ." Allocated stack space:" ($sEnd) ($sSize) + ($sp@) - 4 .R
6   ." bytes" CR ."      Total stack space:" ($sSize) 4 .R
7   ." bytes" CR ." Stack space remaining:" ($sp@) ($sEnd) - 4
8   .R ." bytes" CR ;          R->BASE
9 ." You MUST initialize the string stack before you can use the
10 string library:" CR
11 ."      512 INIT$" CR
12 ." will create a string stack with 512 bytes available." CR
13 ." Example: $" 34 EMIT ." RED" 34 EMIT ." $" 34 EMIT
14 ." GREEN" 34 EMIT ." $" 34 EMIT ." BLUE" 34 EMIT ." $.S"
15 CR

```

**BLOCK #52**

```

0 ( String Library BLOAD)
1 ." loading string library " CR
2 ." You MUST initialize the string stack before you can use the
3 string library:" CR
4 ."      512 INIT$" CR
5 ." will create a string stack with 512 bytes available." CR
6 ." Example: $" 34 EMIT ." RED" 34 EMIT ." $" 34 EMIT
7 ." GREEN" 34 EMIT ." $" 34 EMIT ." BLUE" 34 EMIT ." $.S"
8 CR
9 BASE->R DECIMAL 53 R->BASE BLOAD
10 : BLERR IF ." BLOAD error!" THEN ; BLERR FORGET BLERR
11 FORTH DEFINITIONS      ;S
12
13
14
15

```

**BLOCK #53 – BLOCK #56 String Library Binary**

```

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

```

## BLOCK #57

```

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

```

## BLOCK #58

```

0 ( Catalog program that uses VIB, FDIR and FDRs..LES 19JUN2014)
1 0 CLOAD CAT 0 CLOAD DIR
2 BASE->R CR          ." loading CAT catalog program"
3 HEX 0 VARIABLE Buf1 0FE ALLOT 0 VARIABLE Buf2 12 ALLOT
4 0 VARIABLE Total 0 VARIABLE FCount 0 VARIABLE LC
5 0 VARIABLE bpr 0 VARIABLE sect 0 VARIABLE prot
6 1154 CONSTANT VBuf 0B10 VARIABLE Tabs 181E ,
7 0110 VARIABLE CATPAB
8 : RdErr? ( err --- ) -DUP IF CR ." Disk I/O error "
9  BASE->R [COMPILE] HEX . R->BASE ABORT THEN ;
10 : DSRLNK10 0A 0E SYSTEM 8350 C@ RdErr? ;
11 : getBuf ( bufadr count --- ) VBuf ROT ROT VMBR ;
12 : getSect ( sect# --- ) 8350 ! VBuf 834E ! VBuf 2- 8356 !
13  DSRLNK10 ;
14 : Tab ( n --- ) Tabs + C@ CURPOS @ SCRN_WIDTH @ / GOTOXY ;
15  R->BASE -->

```

## BLOCK #59

```

0 BASE->R DECIMAL          ." ."
1 : getFree ( --- n ) 0 Buf2 ! Buf1 56 + DUP 200 + SWAP DO
2  I @ 65535 XOR -DUP IF 16 0 DO DUP 1 AND Buf2 +! 1 SRL LOOP
3  DROP THEN 2 +LOOP Buf2 @ ; : Head1 ( --- )
4  ." ----- -" CR ; : Head ( --- )
5  ." Name Size Typ B/R Bytes P" CR Head1 ; HEX
6 : DskInfo ( dsk# --- ) SWPB 1+ 834C ! 0 getSect Buf1 100 getBuf
7  CR Buf1 0A ." Disk Name: " TYPE CR ." Total: " Buf1 0A + @ 2-
8  DUP U. ." Free: " getFree DUP U.
9  ." Used: " - U. CR ; DECIMAL
10 : Ftype ( --- ) Buf2 17 + C@ bpr ! Buf2 12 + C@ DUP 8 AND
11  prot ! 247 AND CASE 0 OF ." D/F" ENDOF 128 OF
12  ." D/V" ENDOF 2 OF ." I/F" ENDOF 130 OF ." I/V"
13  ENDOF 1 OF ." PGM" sect @ 256 * Buf2 16 + C@ -DUP IF +
14  256 - THEN 0 bpr ! 2 Tab 5 U.R ENDOF ." ????" 0 bpr !
15  ENDCASE bpr @ -DUP IF 4 U.R THEN ; R->BASE -->

```

**BLOCK #60**

```

0 BASE->R DECIMAL                ." ."
1 : DoCAT ( --- ) 0 LC ! 0 Total ! 0 FCount ! Head 1 getSect
2   Buf1 256 getBuf Buf1 BEGIN LC @ 20 MOD 19 = IF KEY
3   CR Head THEN DUP @ -DUP WHILE getSect Buf2 20 getBuf Buf2 10
4   TYPE Buf2 14 + @ DUP sect ! 1+ DUP 0 Tab 4 U.R Total +! 1 Tab
5   Ftype prot @ IF 3 Tab ." Y" THEN CR 1 LC +! 1 FCount +! 2+
6   REPEAT DROP Head1 FCount @ . ." files" 0 Tab Total @ 4 U.R
7   ." sectors" CR ;
8 : CAT ( dsk# --- ) BASE->R [COMPILE] DECIMAL
9   CATPAB VBuf 2- 2 VMBW DskInfo DoCAT R->BASE ;
10 CR ." n CAT - Catalogs a disk. n = disk #." CR
11 ." E.g., 1 CAT catalogs DSK1." CR R->BASE ;S
12
13
14
15

```

**BLOCK #61 – BLOCK #80**

```

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

```

## 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 original TI Forth system was designed around and supplied in this disk format. Though the TI Forth system could not readily be moved to a disk of another size, **fbForth 2.0** consists of only one file, which can easily be moved to a disk of any size. Different disk formats are possible; however, 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>21</sup>	2	double	20	1600 <sup>22</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	Allocation Bitmap (room for 1600 sectors)		57
254			255

<sup>21</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>22</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>23</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>23</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-nybble<sup>24</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 nybbles of the cluster pointer as  $n_1 - n_3$  and the 3 nybbles of the cumulative sector offset as  $m_1 - m_3$ , with the subscripts indicating the significance of the nybble, 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>25</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>26</sup>

24 A nybble (also nibble) is half of one byte (8 bits) and is equal to 4 bits. The editor prefers “nybble” to “nibble” because of its obvious relationship to “byte”. 2 nybbles = 1 byte.

25 The subscript, 10, indicates base 10 (decimal).

26 This example is taken from one of my (Lee Stewart’s) Compact Flash volumes.

## Appendix L Notes on Radix-100 Notation

**fbForth 2.0** floating-point math routines use radix-100 format for floating-point numbers. The term “radix” is used in mathematics to mean “number base”. We will use “radix 100” to describe the base-100 or centimal number system and “radix 10” to describe the base-10 or decimal number system. Radix-100 format is the same format used by the XML and GPL routines in the TI-99/4A console. Each floating-point number is stored in 8 bytes (4 cells) with a sign bit, a 7-bit, excess-64 (64-biased) integer exponent of the radix (100) and a normalized, 7-digit (1 radix-100 digit/byte) significand for a total of 8 bytes per floating point number. The signed, radix-100 exponent can be -64 to +63. (Keep in mind that the exponent is for radix-100 notation. Those same exponents radix 10 would be -128 to +126.) The exponent is stored in the most significant byte (MSB) biased by 64, *i.e.*, 64 is added to the actual exponent prior to storing, *i.e.*, -64 to +63 is stored as 0 to 127.

The significand (significant digits of the number) must be normalized, *i.e.*, if the number being represented is not zero, the MSB of the significand must always contain the first non-zero (significant) radix-100 digit, with the radix exponent of such a value that the radix point immediately follows the first digit. This is essentially scientific notation for radix 100. Each byte contains one radix-100 digit of the number, which, of course, means that each byte can have a value from 0 to 99 (**0** to **63h**) except for the first byte of a non-zero number, which must be 1 to 99. It is easy to view a radix-100 number as a radix-10 number by representing the radix-100 digits as pairs of radix-10 digits because radix 100 is the square of radix 10. In the following list of largest and smallest possible 8-byte floating point numbers, the radix-100 representation is on the left with spaces between pairs of radix-100 digits. The radix-16 (hexadecimal) internal representation of each byte of the number is also shown:

- Largest positive floating point number [hexadecimal: **7F 63 63 63 63 63 63 63**]:  
 $99 . 99 99 99 99 99 \times 100^{63} = 99.999999999999 \times 10^{126}$   
 $= 9.99999999999999 \times 10^{127}$
- Largest negative floating point number [hexadecimal: **80 9D 63 63 63 63 63 63**]:  
 $-99 . 99 99 99 99 99 \times 100^{63} = -99.999999999999 \times 10^{126}$   
 $= -9.99999999999999 \times 10^{127}$
- Smallest positive floating point number [hexadecimal: **00 01 00 00 00 00 00 00**]:  
 $01 . 00 00 00 00 00 00 \times 100^{-64} = 1.000000000000 \times 10^{-128}$
- Smallest negative floating point number [hexadecimal: **FF FF 00 00 00 00 00 00**]:  
 $-01 . 00 00 00 00 00 00 \times 100^{-64} = -1.000000000000 \times 10^{-128}$

The only difference in the internal storage of positive and negative floating point numbers is that only the first word (2 bytes) of negative numbers is negated or complemented (two's complement).

A floating point zero is represented by zeroing only the first word. The remainder of the floating point number does not need to be zeroed for the number to be treated as zero for all floating point calculations.