## fbForth 2.0

# A File-Based Cartridge Implementation of TI Forth 

...updated April 28, 2017

## Lee Stewart

Copyright © 2014-2017 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 the original TI Forth Manual

This page left intentionally blank.

## 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 ..... 8
1.5.1 Startup Changes ..... 8
1.5.2 The Opening Menu ..... 8
1.5.3 Enabling 1024 KiB SAMS Mapping ..... 9
1.5.4 Changes to the fbForth 2.0 ISR ..... 9
1.5.5 Changes to COLD ..... 9
1.5.6 Redefinition of BOOT ..... 10
1.6 Acknowledgments ..... 10
2 Getting Started ..... 11
2.1 Stack Manipulation ..... 12
2.2 Arithmetic and Logical Operations ..... 12
2.3 Comparison Operations ..... 13
2.4 Memory Access Operations ..... 14
2.5 Control Structures ..... 14
2.5.1 IF ... THEN ..... 15
2.5.2 IF ... ELSE ... THEN ..... 15
2.5.3 BEGIN ... AGAIN ..... 16
2.5.4 BEGIN ... UNTIL ..... 16
2.5.5 BEGIN ... WHILE ... REPEAT ..... 17
2.5.6 DO ... LOOP ..... 18
2.5.7 DO ... +LOOP ..... 19
2.5.8 CASE ... ENDCASE ..... 20
2.5.8.1 Without ELSEOF ... ENDOF ..... 21
2.5.8.2 With ELSEOF ... ENDOF ..... 21
2.6 Input and Output to/from the Terminal ..... 22
2.7 Numeric Formatting ..... 23
2.8 Block-Related Words ..... 24
2.9 Defining Words ..... 25
2.10 Miscellaneous Words ..... 26
3 How to Use the fbForth 2.0 Editors ..... 27
3.1 Forth Block Layout Caveat ..... 27
3.2 The Two fbForth Editors ..... 28
3.3 Editing Instructions ..... 30
3.4 Changing Foreground/Background Colors of 64-Col Editor. ..... 32
3.5 Block-Copying Utility ..... 32
4 Memory Maps ..... 34
4.1 VDP Memory Map. ..... 34
4.2 CPU Memory ..... 35
4.3 CPU RAM Pad ..... 36
4.4 Low Memory Expansion. ..... 37
4.5 High Memory Expansion ..... 37
5 System Synonyms and Miscellaneous Utilities ..... 38
5.1 System Synonyms ..... 38
5.1.1 VDP RAM Read/Write ..... 39
5.1.2 Extended Utilities: GPLLNK, XMLLNK and DSRLNK. ..... 40
5.1.3 VDP Write-Only Registers. ..... 41
5.1.4 VDP RAM Single-Byte Logical Operations ..... 41
5.2 Disk Utilities ..... 42
5.3 Listing Utilities ..... 42
5.4 Debugging ..... 43
5.4.1 Dump Information to Terminal ..... 43
5.4.2 Tracing Word Execution ..... 44
5.4.3 Recursion. ..... 44
5.5 Random Numbers ..... 45
5.6 Miscellaneous Instructions ..... 46
6 An Introduction to Graphics ..... 47
6.1 Graphics Modes ..... 47
6.2 fbForth 2.0 Graphics Words. ..... 48
6.3 Color Changes. ..... 48
6.4 Placing Characters on the Screen ..... 50
6.5 Defining New Characters ..... 51
6.6 Sprites ..... 52
6.6.1 Magnification. ..... 52
6.6.2 Sprite Initialization ..... 53
6.6.3 Using Sprites in Bitmap Mode ..... 54
6.6.4 Creating Sprites ..... 54
6.6.5 Sprite Automotion. ..... 56
6.6.6 Distance and Coincidences between Sprites ..... 57
6.6.7 Deleting Sprites ..... 58
6.7 Multicolor Graphics ..... 59
6.8 Using Joysticks ..... 59
6.9 Dot Graphics. ..... 62
6.10 Special Sounds ..... 64
6.11 Constants and Variables Used in Graphics Programming. ..... 64
7 The Floating Point Support Package ..... 65
7.1 Floating Point Stack Manipulation ..... 65
7.2 Floating Point Defining Words ..... 66
7.3 Floating Point Fetch and Store. ..... 66
7.4 Floating Point Conversion Words. ..... 66
7.5 Floating Point Number Manipulation ..... 66
7.6 Floating Point Number Entry ..... 67
7.7 Built-in Floating Point Constants ..... 67
7.8 Floating Point Arithmetic ..... 68
7.9 Floating Point Comparison Words ..... 68
7.10 Formatting and Printing Floating Point Numbers ..... 68
7.11 Transcendental Functions. ..... 70
7.12 Interface to the Floating Point Routines ..... 71
7.13 Handling Floating Point Errors. ..... 71
7.14 Floating Point Error Codes ..... 72
8 Access to File I/O Using TI-99/4A Device Service Routines ..... 73
8.1 Switching VDP Modes After File Setup ..... 73
8.2 The Peripheral Access Block (PAB) ..... 73
8.3 File Setup and I/O Variables. ..... 74
8.4 File Attribute Words ..... 75
8.5 Words that Perform File I/O ..... 76
8.6 Alternate Input and Output ..... 79
8.7 File I/O Example 1: Relative Disk File. ..... 80
8.8 File I/O Example 2: Sequential RS232 File. ..... 81
8.9 Disk Catalog Utilities ..... 82
8.9.1 DIR ..... 82
8.9.2 CAT ..... 83
9 The fbForth 2.0 TMS9900 Assembler. ..... 84
9.1 TMS9900 Assembly Mnemonics ..... 84
9.2 fbForth 2.0's Workspace Registers ..... 85
9.3 Loading and Using the Assembler. ..... 85
9.4 fbForth 2.0 Assembler Addressing Modes ..... 86
9.4.1 Workspace Register Addressing. ..... 87
9.4.2 Symbolic Memory Addressing ..... 87
9.4.3 Workspace Register Indirect Addressing ..... 87
9.4.4 Workspace Register Indirect Auto-increment Addressing ..... 88
9.4.5 Indexed Memory Addressing ..... 88
9.4.6 Addressing Mode Words for Special Registers ..... 88
9.5 Handling the fbForth 2.0 Stacks ..... 89
9.6 Structured Assembler Constructs ..... 89
9.7 Assembler Jump Tokens ..... 90
9.8 Assembly Example for Structured Constructs ..... 90
9.9 Assembly Example with DOES $>$ ASM: ..... 91
9.10 ASM: and DOES $>$ ASM: without the Assembler. ..... 92
9.10.1 ASM: without the Assembler. ..... 93
9.10.2 DOES $>$ ASM: without the Assembler. ..... 94
10 Interrupt Service Routines (ISRs) ..... 95
10.1 Overview of fbForth 2.0's ISR ..... 95
10.2 A Detailed Look at fbForth 2.0's ISR ..... 95
10.3 Installing a User ISR. ..... 96
10.4 Example of a User ISR: DEMO ..... 97
10.4.1 Installing the DEMO ISR. ..... 98
10.4.2 Uninstalling the DEMO ISR. ..... 98
10.5 Some Additional Thoughts Concerning the Use of ISRs ..... 99
11 Potpourri ..... 100
11.1 BSAVE and BLOAD ..... 100
11.1.1 Using BSAVE to Customize How fbForth 2.0 Boots Up ..... 101
11.1.2 An Overlay System with BSAVE and BLOAD ..... 101
11.1.3 An Easier Overlay System in Source Code ..... 103
11.2 Conditional Loads ..... 104
11.3 CRU Words ..... 105
11.4 Useful Additional Stack Words ..... 106
12 fbForth 2.0 Dictionary Entry Structure. ..... 108
12.1 Link Field. ..... 108
12.2 Name Field ..... 108
12.3 Code Field. ..... 109
12.4 Parameter Field ..... 110
12.5 Notes on Resident Dictionary Words. ..... 110
13 Screen Fonts and the Font Editor. ..... 111
13.1 Screen Font Changes as of fbForth 2.0:8 ..... 111
13.2 User Fonts. ..... 112
13.3 Using the Font Editor. ..... 112
13.4 Modifying the 64-Column Editor's Font ..... 114
14 The Stack-based String Library. ..... 116
14.1 Introduction-The Concepts behind the Library. ..... 116
14.1.1 Coding Conventions ..... 116
14.1.2 Stack Notation ..... 117
14.1.3 Loading the String Stack Library ..... 117
14.2 String Constant Words ..... 117
14.3 String Stack Words. ..... 119
14.4 The String Stack. ..... 125
14.5 Error Checking ..... 125
14.6 String Stack Format ..... 125
14.7 String Constant Format. ..... 126
14.8 Throw Codes. ..... 126
14.9 Author Information. ..... 126
15 TI Forth Block Utilities. ..... 127
15.1 TIFBLK: Display TI Forth Block ..... 127
15.2 TIFIDX: Display TI Forth Index Lines. ..... 128
15.3 TIF2FBF: Copy TI Forth Blocks to fbForth Blocks. ..... 129
15.4 TIFVU: TI Forth Browser/Copier ..... 129
16 Speech Words ..... 132
16.1 Testing the State of the Speech Synthesizer ..... 132
16.2 Using the Speech Synthesizer's Resident Vocabulary ..... 132
16.3 The Speech Synthesizer's Resident Vocabulary. ..... 132
16.4 Streaming Raw Speech Data ..... 137
17 Sound Words. ..... 138
17.1 Generating Individual Sounds ..... 138
17.2 Playing Sound Lists ..... 138
18 Signed Integer Division ..... 140
18.1 M/ ..... 140
18.2 SM/REM and FM/MOD ..... 141
18.3 Dr. Dobb’s Journal Article: "Signed Integer Division". ..... 142
Appendix A ASCII Keycodes (Sequential Order) ..... 146
Appendix B ASCII Keycodes (Keyboard Order) ..... 148
Appendix C How fbForth 2.0 differs from Starting FORTH (1st Ed.) ..... 150
Appendix D The fbForth 2.0 Glossary. ..... 157
D. 1 Explanation of Some Terms and Abbreviations ..... 157
D. 2 Naming Conventions for Forth Words ..... 158
D. 3 fbForth 2.0 Word Descriptions ..... 161
Appendix E Differences: fbForth 2.0, fbForth 1.0 and TI Forth ..... 250
E. 1 TI Forth Words not in fbForth 2.0 ..... 250
E. 2 fbForth 1.0 Words not in fbForth 2.0 ..... 253
E. 3 New and Modified Words in fbForth 2.0. ..... 255
Appendix F User Variables in fbForth 2.0. ..... 262
F. 1 fbForth 2.0 User Variables (Address Offset Order) ..... 262
F. 2 fbForth 2.0 User Variables (Variable Name Order) ..... 264
Appendix G fbForth 2.0 Load Option Directory ..... 266
G. 1 Option: 64-Column Editor ..... 266
G. 2 Option: CPYBLK -- Block Copying Utility ..... 266
G. 3 Option: Memory Dump Utility. ..... 266
G. 4 Option: TRACE -- Colon Definition Tracing ..... 266
G. 5 Option: Printing Routines. ..... 267
G. 6 Option: TMS9900 Assembler. ..... 267
G. 7 Option: CRU Words. ..... 267
G. 8 Option: More Useful Stack Words etc ..... 267
G. 9 Option: Stack-based String Library ..... 267
G. 10 Option: DIR -- Disk Catalog Utility ..... 267
G. 11 Option: CAT -- Disk Catalog Utility ..... 267
G. 12 Option: TI Forth Block Utilities ..... 268
G. 13 Option: ASM>CODE -- Code Output Utility. ..... 268
G. 14 Option: Compact Flash Utilities ..... 268
Appendix H Assembly Source for CODEd Words. ..... 269
Appendix I Error Messages ..... 274
Appendix J Contents of FBLOCKS ..... 276
Appendix K Diskette Format Details. ..... 297
K. 1 Volume Information Block (VIB) ..... 297
K. 2 File Descriptor Index Record (FDIR) ..... 298
K. 3 File Descriptor Record (FDR). ..... 298
Appendix L Notes on Radix-100 Notation. ..... 300
Appendix M Bug Fixes as of fbForth 2.0:9 ..... 301

## 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:

$$
\begin{aligned}
& \text { Starting FORTH ( } \left.1^{s t} E d .\right) \\
& \text { by Leo Brodie } \\
& \text { published by Prentice Hall } \\
& \text { Using FORTH } \\
& \text { by Forth Inc. } \\
& \text { Invitation to FORTH } \\
& \text { by Katzan } \\
& \text { published by Petrocelli Books }
\end{aligned}
$$

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 DRO, DRI, etc. does not obtain for fbForth because those words have been eliminated from foForth] Disk drives in TI Forth are numbered starting with 0 and are abbreviated with "DR" preceding the drive number: DR0, DR1, etc. Other TI languages (TI Basic, TI Extended Basic, TI Assembler, etc.) and software refer to disk drives starting with 1 and the abbreviation "DSK" preceding the disk (drive) number: DSK1, DSK2, etc. From this you can see that DR0 and DSK1 refer to the same disk drive. When referring to the disk drives by device names, they will always be DSK1, ..., such as part of a complete file reference, e.g., DSK1.MYFILE.

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

### 1.2 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.
Chapters 1-11 have the same topics and much of the same structure as the original TI Forth Manual. The same goes for the seven original appendices, except for the insertion of the current Appendix E "Differences: fbForth 2.0, fbForth 1.0 and TI Forth", which shifts the original Appendices E-I to F - J.
Forth screens are now referred to as blocks, in line with the current Forth convention.
Though not new since fbForth 1.0, Chapter 12 "fbForth 2.0 Dictionary Entry Structure" bears mentioning here to remind you of its existence and to note the addition of § 12.5 "Notes on Resident Dictionary Words", which describes how the resident dictionary's structure in ROM differs from the user's dictionary in RAM.
Also noteworthy additions since the original TI Forth are

- Appendix K "Diskette Format Details".
- Appendix L "Notes on Radix-100 Notation", which describes in detail the radix-100 (base-100) notation implemented for floating point numbers on the TI-99/4A.
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 SCRFNT, FNT, USEFFL and FONTED .
- A stack-based string library has been added to FBLOCKS and is fully described in Chapter 14 "The Stack-based String Library".
- Chapter 15 "TI Forth Block Utilities" describes a set of utilities added to FBLOCKS for browsing/reading/writing TI Forth blocks.
- Chapter 18 "Signed Integer Division" discusses signed integer division and the different methods available in fbForth $\mathbf{2 . 0}$ for its implementation. The ANS Forth words, SM/REM and FM/MOD, now part of fbForth 2.0, are discussed in detail, as is the User Variable S|F that allows the user to specify which method of signed integer division fbForth $\mathbf{2 . 0}$ should use.
- 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), >DEG, >RAD, CEIL, DEG/RAD, EXP10, EULER_E, F>R, FABS, FCONSTANT, FMT., FLOOR, FMINUS, FP1, FP10, FRAC, FRŌT, FVARIABLE, LN10INV, LOG10, R>F, RAD/DEG and TRUNC .
- File I/O Library.
- 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 $\mathbf{1 6}$, which is much more informative.
- DIR-new disk cataloging word in FBLOCKS 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 in FBLOCKS 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. Many of them are noted in this "Author's Introduction". See Appendix E "Differences: fbForth 2.0, fbForth 1.0 and TI Forth" for a detailed list.
- SAMS memory expansion ( 1024 KiB ) is supported with SAMS?, SAMS!, S0\&TIB! and >MAP .
- Sound for the four separate sound generators is supported with SOUND.
- fbForth 2.0 ISR has been extensively modified to support interrupt driven speech and sound lists.
- The sound lists include sound list \#1, which can be interrupted by sound list \#2. Sound list \#1 will not be paused, but rather will be muted. These sound lists are implemented with PLAYING? and PLAY .
- The TI Speech Synthesizer is supported with TALKING?, SAY and STREAM .
- Scrollable panels (windows) are supported with PANEL , WRAP and SCROLL .
- ASM: ... ;ASM and CODE: ... ;CODE provide clearer ways of defining words with Assembly Language and machine code, respectively.
- DOES>ASM: ... ;ASM and DOES>CODE: ... ; CODE provide clearer ways of coding the runtime behavior of defining words with Assembly Language and machine code, respectively.
- $\mathbf{N}>\mathbf{S}$ pushes to the stack the next number in the input stream. This word is required to get a number to the stack within CODE: ... ; CODE and DOES>CODE: ... ;CODE constructs.
- The DATA [ ... ]DATA construct allows for quickly compiling a block of cells to HERE or into a word definition. This is particularly useful for character (DCHAR) and sprite ( SPDCHAR ) patterns as well as sound lists, etc.
- Users of the nanoPEB or CF7+ devices manufactured by Jaime Malilong, occasionally available on eBay.com, (see website: webpages.charter.net/nanopeb for description) can use the Compact Flash utilities (CF? , CFVOLS and CFMOUNT ), added to FBLOCKS to
discover what volume numbers are mounted as DSK1, DSK2 and DSK3 as well as to mount a specific volume in DSK1, DSK2 or DSK3.
You will notice at startup that a revision number appears following a ' $:$ ' after the version number. The current version:revision is displayed as fbForth 2.0:9. 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, rightjustified, just below the subtitle-"...updated April 28, 2017" in the present case.
Please note that the SI unit "KiB" is used in this document to denote a byte-multiple of $2^{10}=1024$, where "KB" had been so used. This is because the old unit is now an SI unit that denotes a byte-multiple of $10^{3}=1000$.
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
April, 2017
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 \text { (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 $\mathbf{2 . 0}$ system consists of the fbForth $\mathbf{2 . 0}$ cartridge and the blocks file, FBLOCKS, on the system disk. fbForth $\mathbf{2 . 0}$ 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 2.0:9 (c) 2016 Lee Stewart
    ...a file-based TI Forth implementation
```

FBLDCKS mod: 19RPR2017
Type MENU for load options.
fbForth 2.0:9

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


Loading a block in the "Load Block" column in the above menu loads all routines necessary to perform the task(s) described.
To load a particular package, simply type its load-block number, exactly as it appears in the menu, followed by LOAD. For example, to load the fbForth 2.0 TMS9900 Assembler, type 21 LOAD and press <ENTER>. You may load more than one package at a time.
The list of load options may be displayed at any time by typing the word MENU and pressing <ENTER>. See Appendix 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 $\boldsymbol{n}$ 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:

## 246 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 $\mathbf{2 . 0}$ 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.

### 1.5.1 Startup Changes

This section will detail the startup changes for fbForth 2.0 since the June 20, 2015 edition of this manual was published.

### 1.5.2 The Opening Menu

The opening menu has two choices for fbForth $\mathbf{2 . 0}$ shown in the screen shot below:


TEXAS INSTRUMENTS
HZME CDMPUTER
PRESS
1 FOR TI BASIC
2 FIR FBFIRTH $2.0: 9$
3 FQR FBFRRTH 80 CLLUMN

- Option 2 will open in 40 -column Text mode, TEXT .
- Option 3 will open in 80 -column Text mode, TEXT80, which must not be selected unless the user has an F18A, V9938 or similar video display processor capable of 80 -column Text mode. Otherwise, the display will be corrupted and VRAM will be improperly set up.


### 1.5.3 Enabling 1024 KiB SAMS Mapping

After selection of one of the fbForth $\mathbf{2 . 0}$ options, the first thing that the initialization code does is to set up 1024 KiB SAMS, whether or not a SAMS card is present! Then, it tests for proper SAMS operation by writing a 16 -bit value to an arbitrary address, mapping another SAMS bank to the 4 KiB segment containing the written value and, finally, testing for the written value. If the written value is found, the SAMS mapping did not work. If it is not found, SAMS mapping obviously worked. The SAMS flag is set to 0 or 1 , accordingly. It is this flag that is tested by SAMS? , see Appendix D "The fbForth 2.0 Glossary".

### 1.5.4 Changes to the fbForth 2.0 ISR

The fbForth 2.0 ISR is now enabled at startup so that the new speech ( SAY and STREAM) and sound ( PLAY ) words will work. The speech and sound word ISRs are driven by the fbForth 2.0 ISR. It is easy enough to disable it if the user does not use speech, sound or a user ISR and wants to recover the little bit of time it takes for the fbForth 2.0 ISR to check for non-existent ISRs to service. See Chapter 10 "Interrupt Service Routines (ISRs)" for details.

### 1.5.5 Changes to COLD

COLD is the last routine executed by the fbForth 2.0 startup code. Formerly, it was a high-level Forth word that called another high-level Forth word ( BOOT ) at its conclusion. They have both been combined into a single ALC routine that (re-)sets the Forth environment to the default startup conditions.
Holding down a key immediately upon execution of COLD will force COLD to look for FBLOCKS from that disk. If the user executed COLD , the last loaded font is reloaded regardless of the new disk selection indicated by the held key; whereas, at startup, the held key depressed immediately after the menu selection, will, in fact, also cause the search for FBFONT on the held key's disk. Both invocations of COLD will not attempt to load FBLOCKS if <ENTER> is the held key.

If <ENTER> was held down at powerup or after execution of BOOT (see next section), the default disk drive for both FBLOCKS and FBFONT is DSK1. Though DSK1.FBLOCKS is not loaded, DSK1.FBFONT is loaded (or, at least, attempted). For the next go-round with COLD executed by the user, if no key is held down then, both DSK1.FBLOCKS and DSK1.FBFONT will be loaded if found.

User changes to the following settings will survive a user-executed COLD :

- Display font (see USEFFL, SCRFNT , FNT for how to change font)
- Default colors for all VDP modes (see definition of DCT for how to change)
- Default VDP mode, which should be limited to TEXT80 (0) or TEXT (1) (see definition of DCT for how to change)
- Default S0 and TIB changed by S0\&TIB!.


### 1.5.6 Redefinition of BOOT

BOOT has been redefined to restart fbForth 2.0 at the cartridge startup code. The desired default VDP text mode of TEXT80 or TEXT may be set by pushing to the stack 0 or 1 , respectively, prior to executing BOOT :

0 BOOT
will set the default VDP text mode to TEXT80 and reboot fbForth $\mathbf{2 . 0}$ just as though the user had made the selection on the opening screen.

BOOT may also be executed with nothing on the stack:
BOOT
which will set the default VDP text mode to TEXT as though the user had executed
1 BOOT
Holding a disk-selection key or $\langle E N T E R>$ will have the same effect as at powerup selection.

### 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 $\mathbf{2 . 0}$ 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.
- Matthew Splett for his assistance in debugging the fbForth 2.0 kernel code and proofreading this manual.


## 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, ( $1^{s t}$ 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, ( $1^{s t} E d$.) and lists the differences between the Forth language described in the book (poly-Forth) and fibForth 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 ' 0 ' 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:

$$
\left(n_{1} n_{2}---n_{3}\right)
$$

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:

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

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

## SOME SYMBOLS USED IN THIS DOCUMENT

| $n, n_{1}, \ldots$ | 16-bit signed numbers |
| :--- | :--- |
| $d, d_{1}, \ldots$ | 32-bit signed double numbers |
| $u$ | 16-bit unsigned number |
| $u d$ | 32-bit unsigned double number |
| addr, addr $r_{1}, \ldots$ | 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 $)$ |
| $\mid$ | separates alternate results |

### 2.1 Stack Manipulation

The following are the most common stack manipulation words:

| -DUP | $(n---n n \mid n)$ | Duplicate only if non-zero |
| :---: | :---: | :---: |
| . S | ( --- ) | Non-destructively display stack contents |
| > $\mathbf{R}^{1}$ | ( $n--\mathrm{O}$ ) ( R: --- $n$ ) | Move top item on stack to return stack |
| DEPTH | ( --- stack-depth ) | Number of cells on parameter stack |
| DROP | ( $n---$ ) | Discard top of stack |
| DUP | ( $n--n n$ ) | Duplicate top of stack |
| OVER | $\left(n_{1} n_{2}--n_{1} n_{2} n_{1}\right)$ | Make copy of second item on top |
| R | ( ---n) (R: $n---n$ ) | Copy top item of return stack to stack |
| $\mathrm{R}>^{1}$ | ( ---n) (R: $n--$ ) | Move top item on return stack to stack |
| ROT | $\left(n_{1} n_{2} n_{3}---n_{2} n_{3} n_{1}\right)$ | Rotate third item to top |
| SP! | ( --- ) | Clear stack, resetting it to its base S0 |
| SWAP | $\left(n_{1} n_{2}--n_{2} n_{1}\right)$ | Exchange top two stack items |

### 2.2 Arithmetic and Logical Operations

The following are the most common arithmetic and logical operations:

| $*$ | $\left(n_{1} n_{2}--n_{3}\right)$ | Multiply |
| :--- | :--- | :--- |
| */2 | $\left(n_{1} n_{2} n_{3}--\right.$ quot $)$ | Like $* /$ MOD but giving quot only |
| */MOD ${ }^{2}$ | $\left(n_{1} n_{2} n_{3}--\right.$ rem quot $)$ | $n_{1} * n_{2} / n_{3}$ with 32 bit intermediate |
| $\mathbf{+}$ | $\left(n_{1} n_{2}--n_{3}\right)$ | Add |
| $\mathbf{-}$ | $\left(n_{1} n_{2}--n_{3}\right)$ | Subtract $\left(n_{1}-n_{2}\right)$ |
| $\mathbf{/ L}^{2}$ | $\left(n_{1} n_{2}--n_{3}\right)$ | Divide $n_{1}$ by $n_{2}$ and leave quotient $n_{3}$ |
| /MOD ${ }^{2}$ | $\left(n_{1} n_{2}--\right.$ rem quot $)$ | Divide $n_{1}$ by $n_{2}$ giving remainder \& quotient |
| $\mathbf{1 +}$ | $\left(n_{1}--n_{2}\right)$ | Increment by 1 |
| $\mathbf{2 +}$ | $\left(n_{1}--n_{2}\right)$ | Increment by 2 |
| $\mathbf{1 -}$ | $\left(n_{1}--n_{2}\right)$ | Decrement by 1 |

$1>\mathbf{R}$ and $\mathbf{R}>$ must be used with caution as they may interfere with the normal address stacking mechanism of Forth. Make sure that each $>\mathbf{R}$ in your program has an $\mathbf{R}>$ to match it in the same word definition.

2 The division performed by this word will be symmetric if user variable $\mathbf{S} \mid \mathbf{F}=0$ (the default) or floored if $\mathbf{S} \mid \mathbf{F} \neq 0$. See entry for M/ in Appendix D "The fbForth 2.0 Glossary" and Chapter 18 "Signed Integer Division" for details.

| 2- | $\left(n_{1}--n_{2}\right)$ | Decrement by 2 |
| :---: | :---: | :---: |
| ABS | ( $n---\|n\|$ ) | Absolute value |
| AND | $\left(n_{1} n_{2}--n_{3}\right)$ | Bitwise logical AND giving $n_{3}$ |
| D+ | $\left(d_{1} d_{2}--d_{3}\right)$ | Add double precision numbers |
| D+- | $\left(d_{1} n--d_{2}\right)$ | Negate double number $d_{1}$ if sign of $n$ is negative |
| DABS | ( $d$--- \|d| ) | Absolute value of 32-bit number |
| DMINUS | $\left(d_{1}--d_{2}\right)$ | Leave two's complement of 32-bits |
| FM/MOD | ( $d^{\text {n --- rem quot }}$ ) | Mixed-magnitude, floored divide |
| MAX | $\left(n_{1} n_{2}---n_{1} \mid n_{2}\right)$ | Maximum |
| MIN | $\left(n_{1} n_{2}---n_{1} \mid n_{2}\right)$ | Minimum |
| M* | ( $n_{1} n_{2}---d$ ) | Multiply 2 single numbers giving double result |
| M/ ${ }^{2}$ | ( $d$ n --- rem quot ) | Mixed-magnitude divide |
| M/MOD | ( ud u --- urem udquot) | Unsigned mixed-magnitude divide |
| MINUS | $\left(n_{1}--n_{2}\right)$ | Leave two's complement |
| MOD ${ }^{2}$ | $\left(n_{1} n_{2}---n_{3}\right)$ | Modulo ( remainder from $n_{1} / n_{2}$ ) |
| OR | $\left(n_{1} n_{2}---n_{3}\right)$ | Bitwise logical OR $n_{3}$ |
| SGN | ( $n---1\|0\|+1$ ) | Sign of $n$ as $-1\|0\|+1$ |
| SLA | $\left(n_{1} n_{2}---n_{3}\right)$ | Shift $n_{1}$ left arithmetic $n_{2}$ bits giving $n_{3}$ |
| SM/REM | ( $d$ n--- rem quot ) | Mixed-magnitude, symmetric divide |
| SRA | $\left(n_{1} n_{2}---n_{3}\right)$ | Shift $n_{1}$ right arithmetic $n_{2}$ bits giving $n_{3}$ |
| SRC | $\left(n_{1} n_{2}---n_{3}\right)$ | Shift $n_{1}$ right circular $n_{2}$ bits giving $n_{3}$ |
| SRL | $\left(n_{1} n_{2}---n_{3}\right)$ | Shift $n_{1}$ right logical $n_{2}$ bits giving $n_{3}$ |
| SWPB | $\left(n_{1}---n_{2}\right)$ | Swap the bytes of $n_{1}$ producing $n_{2}$ |
| XOR | $\left(n_{1} n_{2}---n_{3}\right)$ | Bitwise logical exclusive OR $n_{3}$ |
| U* | $\left(u_{1} u_{1}---u d_{2}\right)$ | Unsigned * with double product |
| U/ | ( $u_{1} u_{2}$--- urem uquot) | Unsigned / with remainder |

### 2.3 Comparison Operations

The following are the most common comparisons:
$\begin{array}{ll}< & \left(n_{1} n_{2}--- \text { flag }\right) \\ = & \left(n_{1} n_{2}--- \text { flag }\right)\end{array}$
True if $n_{1}$ less than $n_{2}$ (signed)
True if top two numbers are equal

| $>$ | $\left(n_{1} n_{2}--\right.$ flag $)$ | True if $n_{1}$ greater than $n_{2}$ |
| :--- | :--- | :--- |
| $\mathbf{0}<$ | $(n---$ flag $)$ | True if top number is negative |
| $0=$ | $(n--$ flag $)$ | True if top number is 0 (i.e., NOT) |
| $0>$ | $(n---$ flag $)$ | True if top number is positive |
| $\mathbf{U}<$ | $\left(u_{1} u_{2}--\right.$ flag $)$ | Unsigned integer compare |

### 2.4 Memory Access Operations

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

| ! | ( $n$ addr --- ) | Store $n$ at address (store a cell) |
| :---: | :---: | :---: |
| +! | ( $n$ addr ---- ) | Add $n$ to contents of address |
| >MAP | ( bank addr --- ) | Maps SAMS memory bank to addr. |
| ? | ( addr --- ) | Print the contents of address (same as @ . ) |
| @ | ( $a d d r$--- $n$ ) | Replace word address by its contents |
| C! | ( b addr --- ) | Store $b$ at address (store a byte) |
| C@ | ( $a d d r$--- $b$ ) | Fetch the byte at $a d d r$ |
| CMOVE | ( from_addr to_addr $u$---) | Block move $u$ bytes. |
| BLANKS | ( addr $u$--- ) | Fill $u$ bytes with blanks beginning at $a d d r$ |
| ERASE | ( addr $u$--- ) | Fill $u$ bytes beginning at $a d d r$ with 0s |
| FILL | ( addr u b--- ) | Fill $u$ bytes with $b$ beginning at $a d d r$ |
| MOVE | ( from_addr to_addr $u$---) | Block move $u$ 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 $\mathbf{2 . 0}$ while others are from nonresident words that are part of the default system blocks file, FBLOCKS.

### 2.5.1 IF ... THEN

$\begin{aligned} & \text { IF ... THEN } \\ & \text { IF } \\ &\text { ( flag --- })\end{aligned}$
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
```



```
    DUP 1 AND
    IF
        1+
    THEN
;
```

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

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

| IF ... | ELSE | ... |
| :---: | :---: | :---: |
|  | THEN |  |
|  |  | $($ flag --- |

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.

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 $\mathbf{2 0}$ is the block that will be loaded from the current blocks file if MYWORD is not found in the dictionary.

```
: CLOAD
    ( blk# --- )
    [COMPILE] WLITERAL
    STATE @
    IF
        COMPILE <CLOAD>
```

« Define CLOAD to conditionally load a block from blocks file.
« Load blk\# if word after CLOAD not found.
«Force immediate word WLITERAL to compile into definition of CLOAD so it executes when CLOAD executes.
« Get compilation state for IF to test.
"Are we compiling?
« Yes. Defer execution of runtime procedure <CLOAD> by compiling it into word invoking CLOAD in its definition.

```
    ELSE
    <CLOAD> « No. Execute it.
    THEN
; IMMEDIATE
```


### 2.5.3 BEGIN ... AGAIN

BEGIN ... AGAIN
Creates an infinite loop, continually re-executing the words between BEGIN and AGAIN ${ }^{3}$.

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. ${ }^{3}$ 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 ( --- )
    0 BLK !
    [COMPILE] [
    BEGIN
        RP! CR
        QUERY
        INTERPRET
        STATE @
        0= IF
            ." ok:" DEPTH . «Yes. Echo "ok:" to the terminal followed by stack depth.
        THEN
    AGAIN
;
```


### 2.5.4 BEGIN ... UNTIL

```
BEGIN ... UNTIL
    UNTIL (flag --- )
```

END

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

Synonym for UNTIL .
The following example is from the resident dictionary 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 $l f a$ ) of 0 . This topmost word will always be EXECUTE in fbForth. See

[^0]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 $p f a$, you can get the other three field addresses for a given word. You can get the pfa if you know the $n f a$. These facts are used in the following example:

```
: VLIST
    ( --- )
    80 OUT !
    CONTEXT @ @
    0 SWAP
    BEGIN
        DUP C@ 3F AND
        OUT @ +
        SCRN WIDTH @ 3 -
        > IF
            CR 0 OUT !
        THEN
        DUP ID.
        SWAP 1+ SWAP
        PFA LFA @
        SPACE
        DUP 0=
        PAUSE
    OR UNTIL
    DROP CR . ." words listed"
;
```

« Define VLIST to list the CONTEXT vocabulary.
«Takes no parameters and leaves none.
«Store maximum expected character count in OUT.
« Get $n f a$ of last defined word in CONTEXT vocabulary.
«Start word counter at 0 and swap $n f a$ to top of stack.
«Start indefinite loop.
«Dup $n f a$. Get length byte's least-significant 5 bits.
«Add name length to OUT .
«Get screen width - 3 for spaces and end of line.
«Will line be too long?
« Yes. Go to next line and zero character count.
«Dup nfa. Display name.
« Get word count to top. Increment it. Swap nfa back.
«Get lfa from pfa. Get next word's nfa from lfa.
«Emit a space (updates OUT in the process).
«Dup new nfa. Leave true if 0 , else false.
«Pause if keystroke. Return true if <BREAK>, else false.
«OR above flags. Exit loop if true, else repeat.
«Drop leftover $n f a$. Display word count on next line.

``` ;
```


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

BEGIN ... WHILE ... REPEAT
WHILE (flag --- )

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

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:

```
: FIREDOWN
    ( --- )
    BEGIN
        l JOYST DROP DROP
18 = UNTIL
0
```

«Define FIREDOWN to display loop iterations between press and release of left joystick's fire button.
«No parameters in or out.
«Start indefinite loop awaiting fire button press.
« Get state of joystick/keyboard \#1. Save only char value.
«Repeat loop until char is fire-button value (18).
« Initialize counter on stack.

```
    BEGIN
    1 JOYST DROP DROP
18 = WHILE
        1+
    REPEAT
    CR . ." iterations."
;
«Start indefinite loop awaiting release of fire button. « Get state of joystick/keyboard \#1. Save only char value.
" Continue with loop while char value \(=18\), else exit. « Increment loop counter on stack.
«Repeat loop.
« Display \# of iterations on next screen line.


I \((--n) \quad\) Used between DO and LOOP. Places value of loop
DO 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. LOOP causes a return to the word following DO unless termination is reached.

J ( ---n)

LEAVE ( --- )
counter on stack.

Used when DO LOOPs are nested. Places value of next outer loop counter on the stack.

Causes loop to terminate at next LOOP or +LOOP.

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 \(8 \times 8\) array \(a d d r\) 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 )
DEPTH >R
80 DO
80 DO
OVER OVER
J 8 * I +
+ @
$=\mathrm{IF}$
DROP DROP
I J 1 LEAVE
ELSE

```
: 8X8SRCH
( n addr --- F | c r T )

DEPTH >R
80 DO
80 DO
OVER OVER
J 8 * I +
+ @
= IF
I J 1 LEAVE
ELSE
```

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

```

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 \(\mathbf{I}\) (containing loop's index) on the fourth line is the same index as \(\mathbf{J}\) (next outer loop's index) on the eighth line and not the same as \(\mathbf{I}\) on the eighth line. The definitions of \(\mathbf{I}\) and \(\mathbf{J}\) 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+ < Increment column counter left on stack
LOOP «Inner loop end.
DROP «DROP column counter still on stack.
LOOP
;

```

\subsection*{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 from the resident dictionary is the definition of the fbForth word . \(\mathbf{S}\), 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 \(\mathbf{S 0}\).
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 \(\mathbf{S 0}\) to get its address. The upshot of all this is that we need a loop that decrements the stack address by 2 :


\subsection*{2.5.8 CASE ... ENDCASE}

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 catchall ELSEOF ... ENDOF clause (see § 2.5.8.2 below) was added as of fbForth 2.0:8 to make it easier for the programmer to deal with the default case.

\subsection*{2.5.8.1 Without ELSEOF ... ENDOF}

CASE
\(n_{1}\) OF ... ENDOF
\(n_{2}\) OF ... ENDOF
...
\(n_{m}\) OF ... ENDOF
ENDCASE
CASE ( \(n---\) )

Looks for a number ( \(n_{1}, n_{2}, \ldots, n_{m}\) ) matching \(n\). If there is a match, executes the code between the \(\mathbf{O F}\)... ENDOF set that immediately follows the matching number, proceeding then to the code following ENDCASE. If there is no match, the code after the last ENDOF is executed, with ENDCASE dropping \(n\) from the stack. Execution then continues after ENDCASE . Code after the last ENDOF may use \(n\), which is still available; but, it must not consume \(n\). Otherwise, ENDCASE will drop whatever was under \(n\), adversely affecting program logic and possibly causing a stack underflow.

\subsection*{2.5.8.2 With ELSEOF ... ENDOF}

CASE
```

        n
        nz OF ... ENDOF
        nm
        ELSEOF ... ENDOF
    ENDCASE
CASE ( n--- )

```

Looks for a number \(\left(n_{1}, n_{2}, \ldots, n_{m}\right)\) matching \(n\). If there is a match, executes the code between the \(\mathbf{O F}\)... ENDOF set that immediately follows the matching number, proceeding then to the code following ENDCASE. If there is no match before reaching ELSEOF, ELSEOF forces a match by duplicating \(n\) and calling \(\mathbf{O F}\). This has the effect of preventing ENDCASE or any code immediately preceding it from ever executing and is obviously a lot easier on the programmer.

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:

\section*{HEX}
: JKBD ( kbd --- chr xst yst )

8374 C!
?KEY DROP 8375 C@
DUP 12 =
OVER OFF =
OR IF
8377 C@ 8376 C@ ELSE

DUP
CASE
\begin{tabular}{lll}
04 & OF & OFC \\
05 & 4 & 4 \\
05 & 0 & 4 \\
ENDOF \\
06 & EF & 4 \\
\hline
\end{tabular}
«Use radix 16.
« Define JKBD to scan for joystick input.
« In: Keyboard \(k b d=1\) or 2. Out: Value \(c h r\) of key struck, joystick \(\boldsymbol{x}\)-status \(x s t\) and \(\boldsymbol{y}\)-status yst.
« Store \(k b d\) for keyboard \# to scan.
«Check for keystroke. DROP char returned and get KSCAN's returned value.
« Duplicate \(c h r\) and check for fire button.
«Duplicate chr again and check for "no keystroke".
" Was fire-button or no key depressed?
" Yes. Leave \(x s t\) and \(y s t\) on stack on top of chr.
«No.
« Duplicate \(c h r\) for input to CASE .
\[
\begin{array}{lll}
« c h r=\mathbf{4}(\mathrm{NW}) ? & x s t=\mathbf{F C h}, & y s t=\mathbf{4} \\
« c h r=\mathbf{5}(\mathrm{N}) ? & x s t=0, & y s t=\mathbf{4} \\
« c h r=\mathbf{6}(\mathrm{NE}) ? & x s t=\mathbf{4}, & y s t=\mathbf{4}
\end{array}
\]
```

        0 2 ~ 0 F ~ 0 F C ~ 0 ~ E N D O F ~ < c h r = 2 ( W ) ? ~ x s t = ~ F C h , ~ y s t = 0
        0 3 ~ O F ~ 4 ~ 0 ~ E N D O F ~ < c h r = 3 ( E ) ? ~ x s t = 4 , ~ y s t = 0
        0F OF OFC 0FC ENDOF <chr=Fh(SW)? xst=FCh, yst= FCh
        0 0 ~ O F ~ 0 ~ 0 F C ~ E N D O F ~ < c h r = 0 ( S ) ? ~ x s t = 0 , ~ y s t = ~ F C h
        OE OF 4 0FC ENDOF <chr=Eh(SE)? xst=4, yst= FCh
        ELSEOF DROP 0 0 0 ENDOF «Illegal chr: Drop chr and leave three 0s.
        ENDCASE «Never executed due to use of ELSEOF
    THEN
    0 8374 C!
    ;

```

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 0 Fh , so the OF clause that follows 0Fh executes its contents, ultimately executing QUIT to get back to the terminal command line interpreter.

\subsection*{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 .
\begin{tabular}{|c|c|c|}
\hline . & ( \(n---\) ) & Print a signed number \\
\hline . \({ }^{\prime}\) & ( --- ) & Print a string terminated by " \\
\hline . BASE & ( ---n) & Print the decimal value of the current radix (number base) \\
\hline . R & ( \(n_{1} n_{2}---\) ) & Print \(n_{1}\) right-justified in field of width \(n_{2}\) \\
\hline ?KEY & ( ---n) & Read keyboard. No key? \(n=0\). Key? \(n=\) ASCII keycode . \\
\hline ?TERMINAL & ( --- flag ) & Test if <BREAK> (<CLEAR> on TI-99/4A) pressed \\
\hline BASE & ( --- addr ) & System variable containing number base. To set some base (e.g., Octal) use the following sequence from any base above Octal: 8 BASE ! \\
\hline CLS & ( --- ) & Clears screen. \\
\hline COUNT & ( \(a d d r---a d d r+1 n)\) & Move length byte from a packed character string \({ }^{4}\) at \(a d d r\) to stack and increment \(a d d r\)-suitable for TYPE \\
\hline CR & ( --- ) & Perform a Carriage Return + Line Feed \\
\hline D. & ( \(d\)--- ) & Print double-precision (DP) number \\
\hline D.R & ( \(d n---\) ) & Print DP number right-justified in field of width \(n\) \\
\hline DECIMAL & ( --- ) & Sets the base to Decimal (Base 10) \\
\hline
\end{tabular}

\footnotetext{
4 A packed character string is a string of characters with a leading length byte. Several fbForth words expect or produce such strings.
}
\begin{tabular}{lll} 
EMIT & \((c---)\) & Type character from stack to terminal \\
EXPECT & \((\) addr \(n---)\) & Read \(n\) characters (or until CR) from terminal to \(a d d r\) \\
GOTOXY & \((\) col row --- \()\) & Places cursor at designated column and row of screen. \\
HEX & \((---)\) & Sets the base to Hexadecimal (Base 16) \\
KEY & \((--c)\) & Wait for a keystroke and put its ASCII value on the stack. \\
PAGE & \((---)\) & Clears screen and places cursor at top left of screen. \\
PANEL & \((x y w h---)\) & Sets up panel (window) on screen for SCROLL . \\
SCROLL & \((\) dir --- \()\) & Scrolls screen panel set up with PANEL in direction dir. \\
SPACE & \((---)\) & Type 1 space \\
SPACES & \((n---)\) & Type \(n\) spaces \\
TYPE & \((\) addr \(n---)\) & Type \(n\) characters from addr to terminal \\
U. & \((u---)\) & Print an unsigned number \\
WORD & \((c---)\) & Read one word from input stream delimited by \(c\) \\
WRAP & \((---w r a p)\) & A user variable containing the wrapping flag for SCROLL.
\end{tabular}

\subsection*{2.7 Numeric Formatting}

Advanced numeric formatting control is possible with the following words:
\begin{tabular}{lll} 
NUMBER & \((a d d r---d)\) & Convert string at \(a d d r\) to a double number \(d\) \\
<\# & \((---)\) & Start output string conversion \\
\# & \(\left(d_{1}--d_{2}\right)\) & Convert next, least-significant digit of \(d_{1}\) leaving \(d_{2}\) \\
\#S & \(\left(d--0_{0}\right)\) & Convert all significant digits from right to left \\
SIGN & \((n d--d)\) & Insert sign of \(n\) into number \\
HOLD & \((c---)\) & Insert ASCII character \(c\) into string \\
\#> & \((d---a d d r u)\) & Terminate conversion, ready for TYPE
\end{tabular}

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

```

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.

\subsection*{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:
\begin{tabular}{|c|c|c|}
\hline B/BUF & (---n) & Constant: Block size in bytes (always 1024 in fbForth) \\
\hline BLK & ( --- addr ) & User variable containing current block number (contains 0 for terminal input) \\
\hline BLOAD & ( blk --- flag ) & Loads binary image at blk created by BSAVE and returns flag \(=0\) for successful load; otherwise, flag \(=1\). \\
\hline BLOCK & ( \(n---\) addr ) & Leave address of block \(n\), reading it from the current blocks file if necessary \\
\hline BSAVE & ( addr blk \(---b l k_{2}\) ) & Copies to block blk \(f f\). of current blocks file the binary image from addr to HERE, leaving the next available block blk. \\
\hline CLEAR & ( \(n---\) ) & Fill block \(n\) with blanks \\
\hline CLR_BLKS & ( \(n_{1} n_{2}---{ }^{\text {) }}\) & CLEAR a range of blocks from block \(n_{1}\) to block \(n_{2}\) \\
\hline CLOAD & ( blk --- ) (IS:word) & Load block blk if word not in CONTEXT vocabulary. \\
\hline CPYBLK & ( --- ) & Copy a range of blocks from one blocks file to the same or a different blocks file from information in input stream \\
\hline EMPTY-BUFFERS & ( --- ) & Erase all buffers \\
\hline FLUSH & ( --- ) & Write all updated (dirty) buffers to disk \\
\hline LIST & ( \(n---\) ) & List block \(n\) to terminal \\
\hline LOAD & ( \(n---\) ) & Interpret block \(n\) \\
\hline MKBFL & ( --- ) & Create a blocks file from string and number in input stream \\
\hline SCR \({ }^{5}\) & ( --- addr ) & User variable containing block number most recently referenced by LIST or EDIT \\
\hline UPDATE & ( --- ) & Mark last buffer accessed as updated (dirty) \\
\hline USEBFL & ( --- ) & Select a different blocks file from input stream \\
\hline
\end{tabular}

\footnotetext{
5 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.
}

\subsection*{2.9 Defining Words}

The following are defining words. They are used not only to create new Forth words; but, in the case of words using <BUILDS, to create new defining words.
\begin{tabular}{|c|c|c|}
\hline xxx & ( --- ) & Begin colon definition of \(\mathbf{x x x}^{6}\) \\
\hline ; & ( --- ) & End colon definition \\
\hline VARIABLE xxx & ( \(n---\) ) & Create variable with initial value \(n\) \\
\hline xxx & ( --- addr ) & Returns address when executed \\
\hline FVARIABLE xxx & ( \(f\)--- ) & Create floating-point (FP) variable with initial value \(f\) \\
\hline xxx & ( --- addr ) & Returns address when executed \\
\hline CONSTANT xxx & ( \(n---\) ) & Create constant with value \(n\) \\
\hline xxx & (---n) & Returns \(n\) when executed \\
\hline FCONSTANT xxx & ( \(f---\) ) & Create FP constant with value \(f\) \\
\hline xxx & ( ---n) & Returns \(f\) when executed \\
\hline FILE \(x x x\) & ( \(\left.v a_{1} a d v a_{2}---\right)\) & Define a file reference word and associate PAB address \(v a_{1}\), RAM buffer address \(a d\) and VRAM buffer address \(v a_{2}\) with it \\
\hline xxx & & Makes current the file referenced by \(\mathbf{x x x}\) by setting PAB-ADDR, PAB-BUF, PAB-VBUF to \(v a_{1}, a d, v a_{2}\), respectively \\
\hline USER xxx & ( \(n---\) ) & Create user variable with offset \(n\) bytes from base address of user variable table \\
\hline xxx & ( --- addr ) & Returns address \(a d d r\) of user variable \(\mathbf{x x x}\) \\
\hline ASM: \(x x x\)... ; ASM & ( --- ) & Define assembly-language primitive named xxx \\
\hline CODE: \(x x x\)... ; CODE & ( --- ) & Define machine-code primitive named \(\mathbf{x x x}\) \\
\hline \[
\begin{aligned}
& \text { : } x x x \text { <BUILDS ... } \\
& \text { DOES>ASM: ... ;ASM }
\end{aligned}
\] & & Create new defining word \(\mathbf{x x x}\) with execution-time, assembly-language routine \\
\hline ```
: xxx <BUILDS .... ... ;CODE
``` & & Create new defining word \(\mathbf{x x x}\) with execution-time, machine-code routine \\
\hline \[
\begin{gathered}
\text { : xxx <BUILDS ... } \\
\text { DOES> ... ; }
\end{gathered}
\] & & Create new defining word \(\mathbf{x x x}\) with execution-time, high-level-Forth routine \\
\hline
\end{tabular}

\footnotetext{
6 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 FORGETting it.
}

\subsection*{2.10 Miscellaneous Words}

The following words are relatively common, but don't fit well into any of the above categories:
\begin{tabular}{|c|c|c|}
\hline xxx & ( --- addr ) & Leave parameter field address (pfa) of \(\mathbf{x x x}\). If compiling, compile address. (tick) \\
\hline ( & ( --- ) & Begin comment. Terminated by ) \\
\hline \(\backslash\) & ( --- ) & Begin line comment. \\
\hline , & ( \(n---\) ) & Compile \(n\) into the dictionary (comma) \\
\hline ABORT & ( --- ) & Error termination \\
\hline ALLOT & ( \(n---\) ) & Leave \(n\)-byte gap in dictionary \\
\hline CONTEXT & ( --- addr ) & Leave address of pointer to context vocabulary (searched first) \\
\hline CURRENT & ( --- addr ) & Leave address of pointer to current vocabulary (new definitions placed there) \\
\hline DATA [ & \[
\begin{aligned}
& (--- \text { addr } n) \\
& \left(\text { IS: } n_{1} \ldots n_{\mathrm{n}}\right)
\end{aligned}
\] & Compile numbers until JDATA. Leave address and number of cells \(n\) on stack or in word definition. \\
\hline DEFINITIONS & (---) & Set CURRENT to CONTEXT \\
\hline FORGET xxx & ( --- ) & Forget all definitions back to and including \(\mathbf{x x x}^{6}\) \\
\hline FORTH & ( --- ) & Set CONTEXT to main Forth vocabulary. \\
\hline HERE & ( --- addr ) & Leaves address of next unused byte in the dictionary \\
\hline IN & ( --- addr ) & User variable containing offset into input buffer. \\
\hline PAD & ( --- addr ) & Leaves address of scratch area (68 bytes above HERE ) \\
\hline PLAY & ( addr flag --- ) & Starts sound list at \(a d d r\), depending on flag. \\
\hline S" & \[
\begin{aligned}
& \text { ( --- addr } \mid \text { [ ] ) } \\
& \text { (IS:string" })
\end{aligned}
\] & Store string as packed string at PAD or within a word definition. Leaves address of length byte. \\
\hline SAY & ( addr \(n\)--- ) & Speaks \(n\) Speech-Synthesizer words from addr. \\
\hline SOUND & ( pitch vol ch\# --- ) & Starts sound generator ch\# at pitch and volume vol. \\
\hline SP@ & ( --- addr ) & Leaves address of top stack item. \\
\hline STREAM & ( addr \(n---)\) & Speaks \(n\) cells of raw speech data from addr. \\
\hline VOCABULARY xxx & ( --- ) & Define new vocabulary. \\
\hline ]DATA & ( --- ) & Ends number compilation started with DATA[ and updates cell count on stack or in word definition. \\
\hline
\end{tabular}

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 ) 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.

\section*{\(3 \quad\) How to Use the fbForth 2.0 Editors}

Words introduced in this chapter:
\begin{tabular}{lll} 
CLEAR & EDIT & TEXT \\
CLR_BLKS & EMPTY-BUFFERS & TEXT80 \\
CPYBLK & FLUSH & USEBFL \\
ED@ & MKBFL & WHERE
\end{tabular}

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 \({ }^{7}\) blocks file that fills the disk can hold 89 Forth blocks (numbered \(1^{8}-89\) ). There will actually be one sector ( 256 bytes) left because disk and file overhead occupy 3 sectors and the blocks file occupies 356 sectors ( \(89 \cdot 4\) ), 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.

\subsection*{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 linefeed 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 \(\boldsymbol{+}\) LOOP .

\footnotetext{
7 DF128 refers to the file format: Display data type, Fixed record length, 128-byte logical record length
8 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 .
}

\subsection*{3.2 The Two fbForth Editors}

There are two Forth editors available in fbForth 2.0. The first, which is in the resident dictionary, operates in TEXT or TEXT80 \({ }^{9}\) mode. It will be referred to as the \(40 / 80\)-column editor \({ }^{10}\). Each block is displayed in roughly two halves (left and right) in normal sized characters in TEXT mode.


\footnotetext{
9 TEXT80 mode should only be invoked if your computer is equipped with a VDP that can display 80 columns of text. No harm is done to VRAM except that what shows on the screen will be unpredictable. You can easily restore 40 -column mode by executing TEXT, even though you may not be able to see what you are typing.

10 The 40/80-column Forth editor may only be used when the computer is in TEXT or TEXT80 mode (see Chapter 6). For example, if the \(40 / 80\)-column editor is loaded, don't type EDIT while you are in SPLIT or SPLIT2 mode because the screen will be corrupted and the computer will likely need to be restarted.
}

The full block is displayed in TEXT80 mode.


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.


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

\section*{FORGET TCHAR ok:0}
or by restarting with COLD or MON. Use whichever editor you prefer. Editing instructions are identical for each.

\subsection*{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:

\section*{15 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 :

\section*{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 ( \({ }^{11}\) and end with a). You may also use \(\backslash\) to start a line comment.

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) and the idea and some code was used with his permission.

\footnotetext{
11 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:
\begin{tabular}{|c|c|}
\hline key & function \\
\hline <FCTN + S>, ( \(\leftarrow\) ) & moves the cursor one position to the left. \\
\hline <FCTN + D>, \((\rightarrow)\) & moves the cursor one position to the right. \\
\hline \(<F C T N+E\rangle\), ( \(\uparrow\) ) & moves the cursor up one position. \\
\hline <FCTN+X>, ( \(\downarrow\) ) & moves the cursor down one position. \\
\hline <DELETE> (<FCTN+1>) & deletes the character on which the cursor is placed. \\
\hline <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. \\
\hline <AID> (<FCTN+7>) & erases from the cursor to the end of a line and saves the erased characters in PAD. 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 PAD there. \\
\hline <BEGIN> (<FCTN+5>) & 40/80-column editor: in TEXT 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 TEXT80 mode, places the cursor in the upper left corner. \\
\hline & 64-column editor: places the cursor in the upper left corner \\
\hline \[
\begin{aligned}
& \text { <ERASE> }(<\text { FCTN }+3>) \\
& \text { <REDO> }(<F C T N+8>)
\end{aligned}
\] & 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> 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. \\
\hline <CTRL +8 > & will insert a blank line just above the line the cursor is on. \\
\hline <CTRL \(+V>\) & will tab forward by words. \\
\hline <FCTN + V> & will tab backwards by words. \\
\hline
\end{tabular}

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.

\subsection*{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,
( 01000040 VFILL)
by removing the parentheses:

\section*{01000040 VFILL}

For some other combination, change \(\mathbf{0 4 0}\) to \(\mathbf{0 X Y}\), where \(\mathbf{X}\) is the desired hexadecimal digit for the foreground color and \(\mathbf{Y}\) 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:

\section*{0F 7 VWTR}

If you do not want white, change the \(\mathbf{0 F}\) to the screen color of your choice.
The final chunk of commented code,
( 1000800 01B VFILL)
only requires un-commenting to get black on yellow for the bottom 8 -line section:
1000800 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 01 F 50 SPRITE to 01 new_color 5 0 SPRITE, where new_color is your new color (see § 6.3 "Color Changes").

\subsection*{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:

\section*{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 .

\section*{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. \({ }^{12}\)

The VDP memory can be configured in many ways by the user. The fbForth \(\mathbf{2 . 0}\) 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 8300 h . 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 \(\mathbf{2 . 0}\) system. The use of the remainder of the RAM in this area is dictated by the TI-99/4A's resident operating system.

The 32 KiB memory expansion is divided into an 8 KiB piece at 2000 h and a 24 KiB piece at A000h. The small piece contains BIOS and utility support for fbForth \(\mathbf{2 . 0}\) as well as 4 KiB 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.

\subsection*{4.1 VDP Memory Map}
\begin{tabular}{|c|c|c|c|c|c|c|}
\hline \multicolumn{6}{|l|}{Address} & \multirow[t]{9}{*}{\[
\begin{aligned}
& \text { Address } \\
& 0000 \mathrm{~h}
\end{aligned}
\]} \\
\hline 0000h & Graphics \& Multicolor Screen Image Table bytes: 300h & \begin{tabular}{l}
Text \\
Screen Ta
\end{tabular} & Mode Image ble & \multirow[t]{4}{*}{Bitmap Color Table} & \multirow[t]{4}{*}{1800h} & \\
\hline 0300h & Sprite Attribute List 80h & 40 Columns & 80 Columns & & & \\
\hline 0380h & Color Table 20h & TEXT & TEXT80 & & & \\
\hline 03A0h & Unused 20h & 3C0h & 780h & & & \\
\hline 03C0h & VDP Rollout Area 20h & & & & & \\
\hline 03E0h & Value Stack 80h & & & & & \\
\hline 0460h & PABS etc. 320h & & & & & \\
\hline 0780h & Sprite Motion Table 80h & Value Stack for & TEXT80] & & & \\
\hline
\end{tabular}

\footnotetext{
12 Hexadecimal (base 16) notation for integers in this manual is indicated when a string of \(1-4\) hexadecimal digits ( \(0-9, \mathbf{A}-\mathbf{F}\) ) 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.
}
\begin{tabular}{|c|c|c|c|}
\hline \multicolumn{3}{|l|}{Address} & \multirow[t]{19}{*}{Address} \\
\hline 0800h & Pattern \& Sprite Descriptor Tables
\[
\begin{equation*}
0-127 \tag{400h}
\end{equation*}
\] & & \\
\hline 0C00h & 128-255 400h & & \\
\hline 1000h & fbForth's Disk Buffer 80h & & \\
\hline 1080h & PAB: User Screen Font File 46h & & \\
\hline 10C6h & PAB: Current Blocks File 46h & & \\
\hline 110Ch & PAB: Second Blocks File 46h & & \\
\hline 1152h & Unused [PABS points here for TEXT80]
2686h & & \\
\hline & & Bitmap Screen Image Tab. 300h & \\
\hline & & Sprite Attribute List 80h & \\
\hline & & fbForth's Disk Buffer 80h & \\
\hline & & PAB: User Screen Font File 46h & \\
\hline & & PAB: Current Blocks File 46h & \\
\hline & & PAB: Second Blocks File 46h & \\
\hline & & User PABs, etc. 2EEh & \\
\hline & & Stack for VSPTR 40h & \\
\hline & & Bitmap Pattern Descriptor Table 1800h & \\
\hline 37D8h & Disk Buffer Region for 3 Simultaneous Disk Files 828h & Sprite Descriptor Table 1DEh & \\
\hline 3FFFh & & Disk Buffer Region: 2 Files 622h & \\
\hline
\end{tabular}

\subsection*{4.2 CPU Memory}
\begin{tabular}{|c|c|}
\hline Address & \\
\hline 0000h & Console ROM \\
\hline 2000h & Low Memory Expansion Loader, Your Program, REF/DEF Table \\
\hline 4000h & Peripheral ROMs for DSRs \\
\hline 6000h & fbForth 2.0 ROMs (including Resident Dictionary) in Command Module \\
\hline 8000h & \begin{tabular}{l}
Memory-mapped Devices for VDP, GROM, SOUND, SPEECH. \\
CPU RAM at \(8300 \mathrm{~h}-83 \mathrm{FFh}\)
\end{tabular} \\
\hline A000h & High Memory Expansion \\
\hline FFFFh & Your Program (up to parameter stack \& TIB at high end) \\
\hline
\end{tabular}

\subsection*{4.3 CPU RAM Pad}
\begin{tabular}{|c|c|}
\hline Address \({ }^{13}\) & \\
\hline \[
\begin{aligned}
& 8300 \mathrm{~h} \\
& 831 \mathrm{Fh}
\end{aligned}
\] & fbForth's Workspace (see § 9.2) \\
\hline \[
\begin{aligned}
& 8320 \mathrm{~h} \\
& 832 \mathrm{Dh}
\end{aligned}
\] & -FREE- Eh \\
\hline \[
\begin{aligned}
& \text { 832Eh } \\
& 8347 \mathrm{~h}
\end{aligned}
\] & fbForth's Inner Interpreter, etc. \\
\hline \[
\begin{aligned}
& 8348 \mathrm{~h} \\
& 8349 \mathrm{~h}
\end{aligned}
\] & -FREE- 2 \\
\hline \[
\begin{aligned}
& \text { 834Ah } \\
& \text { 8351h }
\end{aligned}
\] & FAC (Floating Point Accumulator) \\
\hline 8354h & Floating Point Error \\
\hline 8355h & Floating Point String \(\leftrightarrow\) Number Conversion Options \\
\hline \[
\begin{aligned}
& 8356 h \\
& 8357 \mathrm{~h}
\end{aligned}
\] & Subroutine Pointer for DSRs use these 3 bytes \\
\hline \[
\begin{aligned}
& \text { 835Ch } \\
& \text { 8363h }
\end{aligned}
\] & ARG (Floating Point Argument Register) \\
\hline \[
\begin{aligned}
& \text { 836Eh } \\
& \text { 836Fh }
\end{aligned}
\] & VSPTR (Value Stack Pointer) \\
\hline \[
\begin{aligned}
& \text { 8370h } \\
& \text { 8371h }
\end{aligned}
\] & Highest Available Address of VDP RAM \\
\hline 8372h & Least Significant Byte of Data Stack Pointer \\
\hline 8373h & Least Significant Byte of Subroutine Stack Pointer \\
\hline 8374h & Keyboard Number to be Scanned \\
\hline 8375h & ASCII Keycode Detected by Scan Routine \\
\hline 8376h & Joystick Y-status \\
\hline 8377h & Joystick X-status \\
\hline 8379h & VDP Interrupt Timer \\
\hline 837Ah & Number of Sprites that can be in Automotion \\
\hline 837Bh & \begin{tabular}{|lll} 
VDP Status Byte & Bit \(0^{14}\) & 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 \({ }^{\text {th }}\) Sprite on a Line
\end{tabular} \\
\hline 837Ch & \begin{tabular}{lll} 
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 \\
\hline
\end{tabular} \\
\hline 837Dh & VDP Character Buffer \\
\hline 837Eh & Current Screen Row Pointer \\
\hline 837Fh & Current Screen Column Pointer \\
\hline 8380h & Default Subroutine Stack \\
\hline 83A0h & Default Data Stack \\
\hline 83C0h & Random Number Seed (Begin Interrupt Workspace) \\
\hline 83C2h & \begin{tabular}{l}
Flag Bit \(0 \quad\) Disable All of the Following \\
Bit 1 Disable Sprite Motion
\end{tabular} \\
\hline
\end{tabular}

13 Locations omitted are not used by fbForth, but may be used by system routines.
14 Bit \(0=\) high order bit.
\begin{tabular}{|c|c|c|c|}
\hline \multicolumn{4}{|l|}{Address} \\
\hline & \multicolumn{3}{|r|}{\begin{tabular}{ll} 
Bit 2 & Disable Auto Sound \\
Bit 3 & Disable System Reset Key (Quit)
\end{tabular}} \\
\hline 83C4h & \multicolumn{3}{|l|}{Link to ISR Hook} \\
\hline 83C6h & \multicolumn{3}{|l|}{Default keyboard argument - 3 (i.e., \(0-2\) )} \\
\hline 83C7h & \multicolumn{3}{|l|}{Keyboard column 0 (special keys)} \\
\hline 83C8h & \multicolumn{3}{|l|}{Scan code of current key, whatever keyboard type} \\
\hline 83C9h & \multicolumn{3}{|l|}{Ditto for keyboard type 4 (Pascal)} \\
\hline 83CAh & \multicolumn{3}{|l|}{Ditto for keyboard type 5(Standard) [Keyboard Debounce?]} \\
\hline 83CCh & \multicolumn{3}{|l|}{Sound List Pointer (VDP RAM)} \\
\hline 83CEh & \multicolumn{3}{|l|}{Sound List Initiation (set to 01h) \& Countdown Byte} \\
\hline 83D0h & \multicolumn{3}{|l|}{Search Pointers for GROM \& ROM} \\
\hline 83D4h & \multicolumn{3}{|l|}{Contents of VDP Register 1} \\
\hline 83D6h & \multicolumn{3}{|l|}{Screen Timeout Counter} \\
\hline 83D8h & \multicolumn{3}{|l|}{Return Address Saved by Scan Routine} \\
\hline 83DAh & \multicolumn{3}{|l|}{Player Number Used by Scan Routine} \\
\hline 83E0h & G & R0 & «Data (Src) \\
\hline 83E2h & P & R1 & «Address (Src) \\
\hline 83E4h & L & R2 & «Data (Dst) \\
\hline 83E6h & W & R3 & «Address (Dst) \\
\hline 83E8h & O & R4 & «MSB: (Src Flag) LSB: (Dst Flag) \\
\hline 83EAh & r & R5 & «MSB: Word Command Flag \\
\hline 83ECh & k & R6-R8 & \\
\hline 83F2h & s & R9 & «MSB: GPL Code \\
\hline 83F4h & p & R10-R12 & \\
\hline 83FAh & a & R13 & «Current GROM Port (9800h) \\
\hline 83FCh & c & R14 & «Timer Tick \& Flags \\
\hline 83FEh & e & R15 & «VDPWA (8C02h) \\
\hline
\end{tabular}

\subsection*{4.4 Low Memory Expansion}
\begin{tabular}{l|ll|}
\cline { 2 - 3 } 2000h & XML Vectors & 0010 h bytes \\
2010h & fbForth Block Buffers (4) & 1010 h \\
3020h & 99/4 Support for fbForth & 0640 h \\
\cline { 2 - 3 } 3660 & User Variable Area & \(\mathbf{0 0 8 0 \mathrm { h }}\) \\
36E0h & Assembler Support, Trampoline Code, \(\ldots\) & \(\mathbf{0 3 2 2 h}\) \\
3A04h & \(\uparrow\) & 05 FCh \\
3FFFh & Return Stack & \\
\cline { 2 - 3 } & &
\end{tabular}

\subsection*{4.5 High Memory Expansion}
\begin{tabular}{|c|c|c|}
\hline \multirow[t]{2}{*}{A000h} & End of Resident fbForth Vocabulary & \\
\hline & & 0030h \\
\hline \multirow[t]{2}{*}{A030h} & \multicolumn{2}{|l|}{User Dictionary Space \(\downarrow\)} \\
\hline & \[
\underset{\text { Parameter Stack }}{\uparrow}
\] & \\
\hline FFA0h FFF1h & Terminal Input Buffer & 0052h \\
\hline
\end{tabular}

\section*{5 System Synonyms and Miscellaneous Utilities}

Words introduced in this chapter:
\begin{tabular}{lll} 
' & RANDOMIZE & VFILL \\
, & RND & VLIST \\
.S & RNDW & VMBR \\
: (traceable) & SEED & VMBW \\
C, & TRACE & VMOVE \\
CLS & TRIAD & VOR \\
DSRLNK & TRIADS & VSBR \\
DUMP & TROFF & VSBW \\
GPLLNK & TRON & VWTR \\
INDEX & UNTRACE & VXOR \\
MYSELF & VAND & XMLLNK
\end{tabular}

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 \({ }^{15}\).
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".

\subsection*{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.

\footnotetext{
15 The term "system synonym" was coined by the developers of TI Forth and likely means "system utilities identical to Editor/Assembler utilities in name and function". A handful of the system synonyms here are actually enhanced utilities, but are still based on Editor/Assembler utilities.
}

\subsection*{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 gpllnk
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.
\[
\begin{array}{llll} 
& \text { base } & \text { byte } & \text { vaddr } \\
\text { instr } \\
\hline \text { HEX } & \text { A3 } & \mathbf{3 8 0} & \text { VSBW }
\end{array}
\]

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 \(\mathbf{3 8 0 h}\).

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.
\begin{tabular}{lllll} 
base & addr & vaddr & count & instr \\
\hline HEX & PAD & \(\mathbf{8 0 8}\) & \(\mathbf{4}\) & VMBW
\end{tabular}
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.
\begin{tabular}{lll} 
base & vaddr & instr \\
\hline HEX & \(\mathbf{7 8 1}\) & VSBR
\end{tabular}
places the contents of VDP address 781h on the stack.
VMBR ( vaddr addr count --- )
Reads multiple bytes from VDP and places them at a specified address. You must specify the VDP source address, a destination address and a byte count.
```

base vaddr addr count instr
HEX 300 PAD 20 VMBR

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

\section*{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.
\begin{tabular}{ccccc} 
base & vaddr & count & byte & instr \\
\hline HEX & 300 & \(\mathbf{2 0}\) & \(\mathbf{0}\) & VFILL
\end{tabular}
fills 32 ( 20 h ) locations, starting at \(\mathbf{3 0 0 h}\), with zeroes.
VMOVE ( vaddr \(_{1}\) vaddr \(_{2}\) count --- )
Copies count bytes from one location ( vaddr \(_{1}\) ) in VDP RAM to another ( \(\mathrm{vaddr}_{2}\) ).
\begin{tabular}{lllll} 
base & vaddr \(_{l}\) & vaddr \(_{2}\) & count & instr \\
\hline HEX & 1500 & 1640 & 100 & VMOVE
\end{tabular}
copies \(256(100 \mathrm{~h})\) bytes from \(\mathrm{vaddr}_{1}\) to \(\mathrm{vaddr}_{2}\). 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.

\subsection*{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 ( \(a d d r---)\)
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.
\begin{tabular}{lllll} 
base & set up FAC for call & \(a d d r\) & instr \\
\hline HEX & \(\mathbf{9 0 0}\) & 834 A & \(\mathbf{1 6}\) & GPLLNK
\end{tabular}
branches to the GPL routine located at 16 h which loads the TI-99/4A standard character set into VDP RAM. It then returns to your program.

\section*{XMLLNK ( \(a d d r---)\)}

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 ( 8000 h - 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.
\begin{tabular}{cll} 
base & \(a d d r\) & instr \\
\hline HEX & \(\mathbf{8 0 0}\) & XMLLNK
\end{tabular}
accesses the floating-point (FP) multiplication routine, located in console ROM. The \(a d d r\) value ( 800 h ) in this case is a reference to offset 10 h 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 10 h (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.

\section*{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):

\section*{: DSRLNK-SP 1014 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 .

\subsection*{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.

\section*{VWTR (bn---)}

VWTR requires 2 parameters; a byte \(b\) to be written and a VDP register number \(n\).
\begin{tabular}{cccc} 
base & \(b\) & \(n\) & instr \\
\hline HEX & F5 & \(\mathbf{7}\) & VWTR
\end{tabular}

The above instruction writes F5h into VDP write only register number 7. This particular register controls the foreground and background colors in text and text 80 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.

\subsection*{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.
\begin{tabular}{llll} 
base & \(b\) & vaddr & instr \\
\hline HEX & F0 & \(\mathbf{8 0 4}\) & VAND \\
HEX & F0 & \(\mathbf{8 0 4}\) & VOR \\
HEX & F0 & \(\mathbf{8 0 4}\) & VXOR
\end{tabular}

Each of the above instructions reads the byte stored at \(\mathbf{8 0 4 h}\) in VDP RAM, performs an AND, OR or XOR on that byte and F0h, and places the result back into VDP RAM at 804h.

\subsection*{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 \(\S 3.5\) "Block-Copying Utility".

\subsection*{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 ( \(\left.b l k_{1} b l k_{2}---\right)\)
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.

\section*{INDEX ( \(\left.b l k_{1} b l k_{2}---\right)\)}

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.

\subsection*{5.4 Debugging}

\subsection*{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 \(a d d r\) and a byte count count. For example,
\begin{tabular}{cccc} 
base & addr & count & instr \\
\hline HEX & \(\mathbf{2 0 1 0}\) & \(\mathbf{2 0}\) & DUMP
\end{tabular}
will list \(32(20 \mathrm{~h})\) bytes of memory beginning at address 2010 h to your terminal:
\begin{tabular}{|c|c|c|c|c|c|}
\hline 2010: & 0001 & 2820 & 6662 & 466F & .. ( fbFo \\
\hline 2018: & 7274 & 6820 & 5745 & 4C43 & rth WELC \\
\hline 2020: & 4F4D & 4520 & 5343 & 5245 & OME SCRE \\
\hline 2028: & 454E & 2D2D & 2D4C & 4553 & EN---LES \\
\hline
\end{tabular}

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.


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 \(\mid\) 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:
\[
\text { íS } \underline{1} \underline{189} \quad \underline{\text { ok: }} 3
\]

\subsection*{5.4.2 Tracing Word Execution}

This section is based on the following article available at 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 \(\mid\) 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)
56TEST (put parameters on stack and execute TEST)
CUBE
\}567\mathrm{ (stack contents upon entering CUBE)
CUBE
1 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 )

```

\subsection*{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 \(\mathbf{2 . 0}\) includes the special word MYSELF .
MYSELF
(---)
The MYSELF instruction places the \(c f a\) 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
+5
FACT
1 5 4
FACT
\$ 4 3
FACT
|54 3 2
FACT
\ [ 4 3 2 w ok:1
.S (check final stack contents)
| 120 ok:1

```

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

\subsection*{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 ( 6 FE5h \(*\) seed +7 AB9h ) 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.

\section*{RNDW ok:1}
will place on the stack a number from 0 to FFFFh.
RND
\[
\left(n_{1}--n_{2}\right)
\]

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 \(n_{2}\).
\begin{tabular}{lll} 
base & \(n_{1}\) & instr \\
\hline DECIMAL & \(\mathbf{1 3}\) & RND
\end{tabular}
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 8802 h 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 83 C 0 h .

SEED
\[
\text { ( } n--- \text { ) }
\]

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

\section*{4 SEED ok:0}
will place the value 4 into the random number generator seed location 83 C 0 h . 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.

\subsection*{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.
\[
\text { ( } n--- \text { ) }
\]
, (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, \(\quad(b---)\)
C, stores \(b\) at HERE . C, is the byte equivalent of , . Care must be taken when using \(\mathbf{C}\), to compile bytes into the dictionary because most storage to the dictionary is celloriented. If HERE is left on an odd address, a word like , will overwrite the previously stored byte!
( --- )

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.

\section*{6 An Introduction to Graphics}

Words introduced in this chapter:
\begin{tabular}{lllll} 
\#MOTION & DELALL & JCRU & SCREEN & SPRPAT \\
BEEP & DELSPR & JKBD & SPCHAR & SPRPUT \\
CHAR & DOT & JMODE & SPDCHAR & SSDT \\
CHARPAT & DRAW & JOYST & SPLIT & TEXT \\
COINC & DTOG & LINE & SPLIT2 & TEXT80 \\
COINCALL & GCHAR & MAGNIFY & SPRCOL & UNDRAW \\
COINCXY & GRAPHICS & MCHAR & SPRDIST & VCHAR \\
COLOR & GRAPHICS2 & MINIT & SPRDISTXY & VDPMDE \\
DATA[ & HCHAR & MOTION & SPRGET & VMODE \\
DCHAR & HONK & MULTI & SPRITE & ]DATA
\end{tabular}

\subsection*{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 graphics 2 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 graphics 2 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:
\begin{tabular}{ll} 
TEXT80 & \((---)\) \\
TEXT & \((---)\) \\
GRAPHICS & \((---)\) \\
MULTI & \((---)\) \\
GRAPHICS2 & \((---)\) \\
SPLIT & \((---)\) \\
SPLIT2 & \((---)\) \\
VMODE & \((n---) \quad\) where \(n\) is one of the VDP mode numbers \((0-6)\).
\end{tabular}

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

```

\section*{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.

\subsection*{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.
\begin{tabular}{lclc} 
Color & \begin{tabular}{c} 
Hex \\
Value
\end{tabular} & Color & \begin{tabular}{c} 
Hex \\
Value
\end{tabular} \\
\hline 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
\end{tabular}

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:
\begin{tabular}{lll} 
base & color & instr \\
\hline HEX & B & SCREEN \\
DECIMAL & 11 & SCREEN
\end{tabular}

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:
\begin{tabular}{llllll} 
base & \(f g\) & \(b g\) & charset & instr \\
\hline HEX & \(\mathbf{4}\) & D & 1A & COLOR & or \\
DECIMAL & \(\mathbf{4}\) & \(\mathbf{1 3}\) & \(\mathbf{2 6}\) & COLOR &
\end{tabular}
(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.

\subsection*{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,
\begin{tabular}{llllll} 
base & col & row & count & char & instr \\
\hline HEX & A & 11 & 5B & 2A & HCHAR \\
or \\
DECIMAL & 10 & 17 & 91 & 42 & \begin{tabular}{l} 
HCHAR \\
(T G)
\end{tabular}
\end{tabular}
will print a stream of \(91 *\) s, starting at column 10 , row 17 , that will wrap from right to left on the screen.

HCHAR does not check to see whether (col,row) is within the screen buffer or whether count will overrun VRAM after the screen buffer. This is the same behavior as in TI Forth. This behavior will be changed in the next build of fbForth 2.0 to conform to how TI Basic and TI Extended Basic implement this function, i.e., in the next build, HCHAR will throw an error if it would start outside the screen buffer and it will wrap to the start of the screen buffer upon reaching the end of the screen buffer.
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 of the same column.

VCHAR does not check to see whether (col,row) is within the screen buffer. Upon reaching the end of the screen buffer, it wraps to the top of the same column. This is different from TI Forth, which wraps to the next column and then to ( 0,0 ), filling the screen buffer if count is high enough. This behavior will be changed in the next build of fbForth 2.0 to conform to how TI Basic and TI Extended Basic implement this function, i.e., in the next build, VCHAR will throw an error if it would start outside the screen buffer and it will wrap to \((0,0)\) upon reaching the end of the screen buffer, as it does now.

\section*{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
\begin{tabular}{lllll} 
base & col & row & instr & \\
\hline HEX & F & \(\mathbf{1 1}\) & GCHAR & or \\
DECIMAL & \(\mathbf{1 5}\) & \(\mathbf{1 7}\) & \begin{tabular}{l} 
GCHAR \\
(T G)
\end{tabular} &
\end{tabular}

2Ah or 42 would be left on the stack.

\subsection*{6.5 Defining New Characters}

Each character in graphics mode is \(8 \times 8\) 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:
\begin{tabular}{ccccc} 
& 3C66h & DBE7h & E7DBh & 663Ch \\
Rows & \(0-1\) & \(2-3\) & \(4-5\) & \(6-7\)
\end{tabular}
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
\begin{tabular}{lllllll} 
base & \(n_{1}\) & \(n_{2}\) & \(n_{3}\) & \(n_{4}\) & char & instr \\
\hline HEX & 3C66 & DBE7 & E7DB & 663C & 7B & CHAR \\
or \\
DECIMAL & 15426 & 56295 & 59355 & 26172 & 123 & CHAR \\
& & & & & & (T G)
\end{tabular}

As you can see, it is more natural to use this instruction in HEX than in DECIMAL .
DCHAR ( addr cnt char --- )
DCHAR can be used to create several contiguous characters at once. For example, to create five such characters starting with character number 123, you would first set up an array of 20 16-bit numbers for the required 40 bytes ( 8 bytes or 4 cells for each
character's pattern), place the array's start address \(a d d r\) and cell count cnt on the stack followed by the character code char (123 in this case). You would then type
\begin{tabular}{llllll} 
base & \(a d d r\) & \(c n t\) & char & instr & \\
\hline HEX & AC34 & \(\mathbf{1 4}\) & 7B & DCHAR & or \\
DECIMAL & 44084 & 20 & \(\mathbf{1 2 3}\) & \begin{tabular}{l} 
DCHAR \\
(T G)
\end{tabular}
\end{tabular}

An easy way to set up the array for the above is to use the DATA [ ... ]DATA construct, which will leave the address and cell count of the array on the stack. Add char to the stack and launch DCHAR. The following example defines characters \(123-127\) to each have the same pattern as the CHAR example above:
```

HEX DATA[ 3C66 DBE7 E7DB 663C
3C66 DBE7 E7DB 663C
3C66 DBE7 E7DB 663C
3C66 DBE7 E7DB 663C
3C66 DBE7 E7DB 663C ]DATA ok:2
7B DCHAR ok:0

```

See Appendix D "The fbForth 2.0 Glossary" for the details of DATA [ and ]DATA.

\section*{CHARPAT ( char --- \(\left.n_{1} n_{2} n_{3} n_{4}\right)\)}

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:
HEX 41 CHARPAT 7E CHAR or
DECIMAL
65 CHARPAT
126 CHAR
(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.

\subsection*{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.

\subsection*{6.6.1 Magnification}

Sprites may be defined in 4 different sizes or magnifications:
Magnification
Factor \(\quad\) Description

Factor
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.
\begin{tabular}{cl}
\begin{tabular}{c} 
Magnification \\
Factor
\end{tabular} & \multicolumn{1}{c}{ Description } \\
\hline 1 & \begin{tabular}{l} 
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.
\end{tabular} \\
2 & \begin{tabular}{l} 
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
\end{tabular} \\
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 & \begin{tabular}{l} 
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.
\end{tabular}
\end{tabular}

The default magnification is 0 .
MAGNIFY ( \(n---\) )
To alter sprite magnification, use the fbForth word MAGNIFY .
\begin{tabular}{ll}
\(n\) & instr \\
\hline \(\mathbf{2}\) & \begin{tabular}{l} 
MAGNIFY \\
\((\) G M B)
\end{tabular}
\end{tabular}
will change all sprites to double size and unmagnified.

\subsection*{6.6.2 Sprite Initialization}

DELALL (---)

\section*{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 .
( 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 2 KiB ( 800 h ) boundary will do.
\begin{tabular}{lll} 
base & \(v a d d r\) & instr \\
\hline HEX & \(\mathbf{2 0 0 0}\) & SSDT \\
DECIMAL & \(\mathbf{8 1 9 2}\) & SSDT \\
& & (G M B)
\end{tabular}
will move the Sprite Descriptor Table to 2000h.

\subsection*{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 1 B 00 h . 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

\section*{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 \(\mathbf{2 . 0}\) 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.
\begin{tabular}{|c|c|}
\hline 3800h & Sprite Patterns 0-58 \\
\hline 39DDh & 01DEh \\
\hline 39DEh & Start of Disk Buffer Region for 2 files \\
\hline
\end{tabular}

\subsection*{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.

\section*{SPCHAR \(\quad\left(n_{1} n_{2} n_{3} n_{4}\right.\) 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 .
\begin{tabular}{lllllll} 
base & \(n_{1}\) & \(n_{2}\) & \(n_{3}\) & \(n_{4}\) & char & instr \\
\hline HEX & 0 FFFF & \(\mathbf{2 4 2 4}\) & F0F0 & 4242 & 0 & SPCHAR \\
DECIMAL & 3855 & 9252 & 61680 & 8770 & 0 & \begin{tabular}{l} 
SPCHAR \\
(GM B)
\end{tabular}
\end{tabular}
defines character 0 in the Sprite Descriptor Table. If your Pattern and Sprite Descriptor Tables overlap, use character numbers below 127 with caution.

\section*{SPDCHAR ( addr cnt char --- )}

SPDCHAR can be used to create several contiguous sprite characters at once. SPDCHAR is identical to DCHAR , (see § 6.5 "Defining New Characters" above) but for sprite pattern definitions because SPDTAB does not always start at the same VRAM address as PDT. Here is the same example as for DCHAR in \(\S 6.5\) to create five identical sprite characters numbering \(123-127\), with pattern array of cnt cells starting at addr:
\begin{tabular}{llllll} 
base & \(a d d r\) & cnt & char & instr & \\
\hline HEX & AC34 & 14 & 7B & SPDCHAR & or \\
DECIMAL & 44084 & 20 & 123 & \begin{tabular}{l} 
SPDCHAR \\
(GM B)
\end{tabular} &
\end{tabular}

As with DCHAR , you can facilitate setting up the pattern array with DATA [ ... ]DATA . See the DCHAR example in § 6.5 above.

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 ( \(0-31\) ).
\begin{tabular}{lllllll} 
base & dotcol & dotrow & color & char & spr & instr \\
\hline HEX & 6B & 4C & 5 & 10 & 1 & SPRITE \\
or \\
DECIMAL & 107 & 76 & 5 & 16 & 1 & \begin{tabular}{l} 
SPRITE \\
(G M B)
\end{tabular}
\end{tabular}
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 --- )
\begin{tabular}{llll} 
base & char & spr & instr \\
\hline HEX & 14 & 1 & SPRPAT \\
DECIMAL & 20 & 1 & SPRPAT
\end{tabular}
(G M B)
will change the pattern of sprite \#1 to character number 20.
SPRCOL ( color spr --- )
\begin{tabular}{llll} 
base & color & spr & instr \\
\hline HEX & C & \(\mathbf{2}\) & \begin{tabular}{l} 
SPRCOL \\
DECIMAL \\
\\
12
\end{tabular} \\
& & & \begin{tabular}{l} 
SPRCOL \\
(G M B)
\end{tabular}
\end{tabular}
will change the color of sprite \(\# 2\) to dark green.
SPRPUT ( dotcol dotrow spr --- )
\begin{tabular}{lllll} 
base & dotcol & dotrow & spr & instr \\
\hline HEX & 28 & \(\mathbf{4 F}\) & \(\mathbf{1}\) & SPRPUT \\
DECIMAL & 40 & \(\mathbf{7 9}\) & \(\mathbf{1}\) & \begin{tabular}{l} 
SPRPUT \\
(G M B)
\end{tabular}
\end{tabular}
will place sprite \#1 at column 40 and row 79.

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

MOTION ( xvel yvel spr --- )
\begin{tabular}{llllll} 
base & xvel & \(y\) vel & spr & instr \\
\hline HEX & FC & \(\mathbf{6}\) & \(\mathbf{1}\) & MOTION & or \\
DECIMAL & -4 & \(\mathbf{6}\) & \(\mathbf{1}\) & MOTION
\end{tabular}
(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 .

(GM)
will set sprites \#0 - \#5 in motion.
\[
\begin{array}{ll}
n & \text { instr } \\
\hline 0 & \text { \#MOTION }
\end{array}
\]
will stop all sprite automotion, but motion will resume when another \#MOTION instruction is executed.

\section*{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.

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.

\subsection*{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 \(\boldsymbol{x}\)-distance difference in dot columns) and \(y_{2}-y_{1}\) (the vertical \(\boldsymbol{y}\)-distance difference in dot rows). The squared distance between the two sprites or the sprite and screen point is calculated by squaring the \(\boldsymbol{x}\)-distance difference and adding that to the square of the the \(\boldsymbol{y}\)-distance difference, i.e., \(d^{2}=\left(x_{2}-x_{1}\right)^{2}+\left(y_{2}-y_{1}\right)^{2}\).

SPRDIST \(\quad\left(s p r_{1} s p r_{2}--n\right)\)
\begin{tabular}{lll}
\(s p r_{1}\) & \(s p r_{2}\) & instr \\
\hline \(\mathbf{2}\) & \(\mathbf{4}\) & \begin{tabular}{l} 
SPRDIST \\
\((\mathrm{G} \mathrm{M} \mathrm{B})\)
\end{tabular}
\end{tabular}
returns on the stack the square of the distance between sprite \#2 and sprite \#4.
SPRDISTXY
(dotcol dotrow spr --- n )
\begin{tabular}{lllll} 
base & dotcol & dotrow & spr & instr \\
\hline DECIMAL & \(\mathbf{6 5}\) & \(\mathbf{2 1}\) & \(\mathbf{5}\) & \begin{tabular}{l} 
SPRDISTXY \\
(G M B)
\end{tabular}
\end{tabular}
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 ( \(s p r_{1} s p r_{2}\) tol --- flag )
To find a coincidence between two sprites, the fbForth word COINC is used.
\begin{tabular}{llll}
\(s p r_{1}\) & \(s p r_{2}\) & tol & instr \\
\hline \(\mathbf{7}\) & \(\mathbf{9}\) & \(\mathbf{2}\) & \begin{tabular}{l} 
COINC \\
(G M B)
\end{tabular}
\end{tabular}
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.

\section*{COINCXY ( dotcol dotrow spr tol --- flag )}

Detecting a coincidence between a sprite and a point is similar.
\begin{tabular}{llllll} 
base & dotcol & dotrow & spr & tol & instr \\
\hline DECIMAL & 63 & 29 & \(\mathbf{8}\) & \(\mathbf{3}\) & \begin{tabular}{l} 
COINCXY \\
(G M B)
\end{tabular}
\end{tabular}
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.

\section*{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.

\subsection*{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 ( \(s p r---\) )

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 \(\boldsymbol{x}=1, \boldsymbol{y}=192\). 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 \(\boldsymbol{y}=\mathrm{D} 0 \mathrm{~h}\), the \(\boldsymbol{y}\)-value that undefines all sprites with numbers greater than or equal to the lowest-numbered sprite with that value.

DELALL ( --- )

\section*{DELALL \\ (GMB)}
on the other hand, will remove all sprites from the screen and from memory. See § 6.6.2 Sprite Initialization above for more details.

\subsection*{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 \(1 / 4\) 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 :
\begin{tabular}{llllll} 
base & color & col & row & instr & \\
\hline HEX & B & 1A & 2C & MCHAR & or \\
DECIMAL & \(\mathbf{1 1}\) & \(\mathbf{2 6}\) & \(\mathbf{4 4}\) & MCHAR &
\end{tabular}

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.

\subsection*{6.8 Using Joysticks}

JOYST
\(\left(\begin{array}{lll}\left.\left.n_{1}---\left[\begin{array}{ccc}\text { char } & n_{2} & n_{3}\end{array}\right] \right\rvert\, n_{2}\right)\end{array}\right.\)
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 \(\boldsymbol{x}\) status \(n_{2}\) and the \(\boldsymbol{y}\) 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 26 h of the user variable table. It is used by JOYST to determine whether to execute JKBD \((=0)\) or \(\operatorname{JCRU}(\neq 0)\). The default value is 0 . See JOYST, JKBD and JCRU in this section.
\[
\left(n_{1}--- \text { char } n_{2} n_{3}\right)
\]

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 \(\boldsymbol{x}\) status \(n_{2}\) and the \(\boldsymbol{y}\) 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.

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 \(\boldsymbol{x}\)-joystick status \(n_{2}\) and a \(\boldsymbol{y}\)-joystick status \(n_{3}\) on top of the stack. The joystick positions are illustrated in the diagram on page 62 .
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 \(\boldsymbol{x}\) - and \(\boldsymbol{y}\)-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 \(\boldsymbol{x}\) - and \(\boldsymbol{y}\)-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 \(\boldsymbol{x}\) and \(\boldsymbol{y}\) 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 ' 0 FC' 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
\[
\left(n_{1}--n_{2}\right)
\]

Executed by JOYST when JMODE \(\neq 0\), JCRU allows input from joystick \#1 \(\left(n_{1}=1\right)\) or \#2 \(\left(n_{1}=2\right)\). 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, \(2=\mathrm{W}, 4=\mathrm{E}, 8=\mathrm{S}\) and \(16=\mathrm{N}\). Two-bit directional combinations are \(18=\mathrm{NW}(\mathrm{N}+\mathrm{W}\) or \(16+2), 20=\mathrm{NE}, 10=\mathrm{SW}\) 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 \(=0\) )

\subsection*{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:
\begin{tabular}{llll} 
base & dotcol & dotrow & instr \\
\hline DECIMAL & \(\mathbf{3 4}\) & \(\mathbf{1 2}\) & DOT
\end{tabular}
will plot or unplot a dot at position \((34,12)\), depending on the value of DMODE .

\section*{DCOLOR ( --- addr )}

DCOLOR is short for "dot color" and should contain either one byte of foregroundbackground (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 \(_{1}\) dotrow \(_{1}\) dotcol \(_{2}\) dotrow \(\left._{2}---\right)\)
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.
\begin{tabular}{llllll} 
base & dotcol \(_{1}\) & dotrow \(_{1}\) & dotcol \(_{2}\) & dotrow \(_{2}\) & instr \\
\hline DECIMAL 23 & \(\mathbf{1 2}\) & \(\mathbf{5 6}\) & \(\mathbf{7 8}\) & LINE
\end{tabular}

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.

\subsection*{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.

\section*{BEEP ( --- )}

The first is called BEEP and produces a pleasant high pitched sound.
HONK
( --- )
The other, called HONK, produces a less pleasant low tone.

\subsection*{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.
\begin{tabular}{|c|c|c|c|c|c|c|}
\hline \multirow[b]{2}{*}{name} & \multirow[b]{2}{*}{type} & \multirow[b]{2}{*}{description} & \multirow[b]{2}{*}{default} & \multicolumn{3}{|l|}{bitmap modes} \\
\hline & & & & 4 & 5 & 6 \\
\hline DMODE & variable & Dot graphics drawing mode & 0 & & 0|1/2 & \\
\hline SMTN & constant & VDP address of Sprite Motion Table & 780h & N/A & N/A & N/A \\
\hline COLTAB & constant & VDP address of Color Table & 380h & 0 & 0 & 0 \\
\hline PDT & constant & VDP address of Pattern Descriptor Table & 800h & 2000h & 3000h & 2000h \\
\hline SATR & constant & VDP address of Sprite Attribute Table & 300h & 1800h & 1800h & 1800h \\
\hline SPDTAB & constant & VDP address of Sprite Descriptor Table & 800h & 3800h & 3800h & 3800h \\
\hline
\end{tabular}

\section*{\(7 \quad\) The Floating Point Support Package}

Words introduced in this chapter:
\begin{tabular}{lllll} 
>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 & \(\wedge\) \\
EULER_E & F= & FMINUS & &
\end{tabular}

The floating point package is designed to make it easy to use the Radix 100 floating point package available in ROM in the fbForth \(\mathbf{2 . 0}\) 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 ( \(\pm 9.9999999999999 \cdot 10^{127}\) ) for numbers that are too large and to 0 for numbers that are too small.

\subsection*{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:
\begin{tabular}{ll} 
FDUP & \((f---f f)\) \\
FDROP & \((f---)\) \\
FOVER & \(\left(f_{1} f_{2}---f_{1} f_{2} f_{1}\right)\) \\
FSWAP & \(\left(f_{1} f_{2}---f_{2} f_{1}\right)\) \\
FROT & \(\left(f_{1} f_{2} f_{3}---f_{2} f_{3} f_{1}\right)\) \\
F>R & \((f---)(\) R: \(---f)\) \\
R \(>\mathbf{F}\) & \((---f)(\) R: \(f---)\)
\end{tabular}

\subsection*{7.2 Floating Point Defining Words}

The following words create new floating point variables and constants. They both require an 8byte floating-point number on the stack to place in the parameter field of the newly defined variable or constant:
\begin{tabular}{cll} 
FVARIABLE \(\mathbf{x x x}\) & \((f---)\) & Create variable with initial value \(f\) \\
\(\quad \mathbf{x x x}\) & \((--\) addr \()\) & Returns address when executed \\
FCONSTANT \(\mathbf{x x x}\) & \((f---)\) & Create constant with value \(f\) \\
\(\mathbf{x x x}\) & \((---f)\) & Returns \(f\) when executed
\end{tabular}

\subsection*{7.3 Floating Point Fetch and Store}

Floating point numbers can be stored and fetched by using
```

F! (faddr --- )
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.)

\subsection*{7.4 Floating Point Conversion Words}

The following words convert numbers on the stack to and from floating point numbers:
S->F \(\quad(n---f)\)
A 16-bit number can be converted to floating point by using \(\mathbf{S - > F}\). It functions by replacing the 16 -bit number on the stack by a floating point number of equal value.

\section*{F->S \(\quad(f---n)\)}

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

\subsection*{7.5 Floating Point Number Manipulation}

\section*{FABS \(\quad(f---|f|)\)}
converts \(f\) to its absolute value.
FMINUS ( \(f\)--- \(-f\) )
negates \(f\) by negating the most significant word (topmost cell on the stack).
FLOOR \(\quad\left(f_{1}---f_{2}\right)\)
finds the closest integer \(f_{2}\) less than or equal to \(f_{1}\).

\section*{CEIL \(\quad\left(f_{1}---f_{2}\right)\)}
finds the closest integer \(f_{2}\) greater than or equal to \(f_{1}\).
TRUNC \(\quad\left(f_{1}---f_{2}\right)\)
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.

\subsection*{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

```

\subsection*{7.7 Built-in Floating Point Constants}

DEG/RAD ( --- \(f\) ) pushes the constant 57.295779513082 (degrees/radian) to the stack.

\section*{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 \(>\mathbf{F} \mathbf{1}\).
FP10
(---f)
pushes the constant 10 to the stack as a floating point number. It is equivalent to \(\mathbf{> F} \mathbf{1 0}\).
LN10INV ( --- \(f\) )
pushes the constant \(1 / \ln (10)=1 / 2.302 \ldots=0.43429448190325\) to the stack.

\section*{PI ( ---f)}
pushes the constant \(\pi=3.141592653590\) to the stack.
RAD/DEG ( \(---f\) )
pushes the constant 0.01745329251994 (radians/degree) to the stack.

\subsection*{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:

\section*{F+ \(\quad\left(f_{1} f_{2}---f_{3}\right)\)}
puts on the stack the result \(\left(f_{3}\right)\) of \(f_{1}+f_{2}\).
F- \(\quad\left(f_{1} f_{2}---f_{3}\right)\)
puts on the stack the result \(\left(f_{3}\right)\) of \(f_{1}-f_{2}\).
F* \(^{*} \quad\left(f_{1} f_{2}---f_{3}\right)\)
puts on the stack the result \(\left(f_{3}\right)\) of \(f_{1} \times f_{2}\).
F/ \(\quad\left(f_{1} f_{2}---f_{3}\right)\)
puts on the stack the result \(\left(f_{3}\right)\) of \(f_{1} / f_{2}\).

\subsection*{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
\begin{tabular}{lll} 
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> & \(\left(f_{1} f_{2}--\right.\) flag \()\) & flag is true if \(f_{1}>f_{2}\) \\
F= & \(\left(f_{1} f_{2}--\right.\) flag \()\) & flag is true if \(f_{1}=f_{2}\) \\
F< & \(\left(f_{1} f_{2}--\right.\) flag \()\) & flag is true if \(f_{1}<f_{2}\)
\end{tabular}

\subsection*{7.10 Formatting and Printing Floating Point Numbers}
F. \((f---)\)

The word \(\mathbf{F}\). 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 \(\mathbf{1 . 0}\) 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. \({ }^{16}\)

The following screen shows examples of all the above situations:


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.

\section*{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 Enotation). 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 . :

\footnotetext{
16 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.
}

\section*{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 no 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:
```

2 1\frac{123 % Ok:4 field too big! ok:0}{1}
>F 123 ok:4

```

```

4 12-3 FOK:4
>F 12 123.4567890123.000000000000 ok:0
\F14 -1.2345678901234E120 FDUP ok:8 ok:0
2 149FFMT. -1.2345678901230E+** ok:4

```

\subsection*{7.11 Transcendental Functions}

The following transcendental functions are also available:
>DEG \(\quad\left(f_{1}--f_{2}\right) \quad\) is the conversion of \(f_{1}\) radians to \(f_{2}\) degrees
\begin{tabular}{lll} 
>RAD & \(\left(f_{1}--f_{2}\right)\) & is the conversion of \(f_{1}\) degrees to \(f_{2}\) radians \\
INT & \(\left(f_{1}--f_{2}\right)\) & Returns largest integer not larger than input \\
a & \(\left(f_{1} f_{2}--f_{3}\right)\) & \(f_{3}\) is \(f_{1}\) raised to the \(f_{2}\) power \\
SQR & \(\left(f_{1}--f_{2}\right)\) & \(f_{2}\) is the square root of \(f_{1}\) \\
EXP & \(\left(f_{1}--f_{2}\right)\) & \(f_{2}\) is \(e(2.71828 \ldots)\) raised to the \(f_{1}\) power \\
EXP10 & \(\left(f_{1}--f_{2}\right)\) & \(f_{2}\) is 10 raised to the \(f_{1}\) power \\
LOG & \(\left(f_{1}--f_{2}\right)\) & \(f_{2}\) is the natural \(\log\) of \(f_{1}\) \\
LOG10 & \(\left(f_{1}--f_{2}\right)\) & \(f_{2}\) is the common \(\log (\log 10)\) of \(f_{1}\) \\
COS & \(\left(f_{1}--f_{2}\right)\) & \(f_{2}\) is the cosine of \(f_{1}\) (in radians) \\
SIN & \(\left(f_{1}--f_{2}\right)\) & \(f_{2}\) is the sin of \(f_{1}\) (in radians) \\
TAN & \(\left(f_{1}--f_{2}\right)\) & \(f_{2}\) is the tangent of \(f_{1}\) (in radians) \\
ATN & \(\left(f_{1}--f_{2}\right)\) & \(f_{2}\) is the arctangent (in radians) of \(f_{1}\)
\end{tabular}

\subsection*{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:
\begin{tabular}{lllll}
\(>\) ARG & FAC & FAC>ARG & FMUL & SETFL \\
\(>\) FAC & FAC->S & FADD & FSUB & VAL \\
ARG & FAC> & FDIV & S->FAC &
\end{tabular}

\subsection*{7.13 Handling Floating Point Errors}

FLERR ( \(---n\) )
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.

\section*{?FLERR ( --- )}
?FLERR issues the following error message if the last floating point operation resulted in an error:

\section*{?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.

\subsection*{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:
\begin{tabular}{cl} 
Code & Error Description \\
\hline 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
\end{tabular}

\section*{8 Access to File I/O Using TI-99/4A Device Service Routines}

Words introduced in this chapter:
\begin{tabular}{llll} 
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 \({ }^{17}\) & VRBL \\
FXD & PAB-VBUF & SET-PAB & WRT
\end{tabular}

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 \(\mathbf{2 . 0}\) 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 \(\mathbf{2 . 0}\) 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.

\subsection*{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.

\subsection*{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.

\footnotetext{
17 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:
\begin{tabular}{|c|c|}
\hline Byte 0 & Byte 1 \\
\hline I/O Opcode & Flag/Status \\
\hline \multicolumn{2}{|l|}{Bytes 2 \& 3} \\
\hline \multicolumn{2}{|l|}{Data Buffer Address in VDP} \\
\hline Byte 4 & Byte 5 \\
\hline Logical Record Length & Character Count \\
\hline \multicolumn{2}{|l|}{Bytes 6 \& 7} \\
\hline \multicolumn{2}{|l|}{Record Number} \\
\hline Byte 8 & Byte 9 \\
\hline Screen Offset (Status) & Name Length \\
\hline \multicolumn{2}{|l|}{Byte 10+} \\
\hline File Descriptor & \\
\hline - & \\
\hline - & \\
\hline
\end{tabular}

\subsection*{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 \(\mathbf{2 . 0}\) 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 ( --- \(a d d r)\)
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 \(\quad(---a d d r)\)
Holds address in VDP RAM of the first byte of a region of adequate length to store data temporally 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 8370 h 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 \(_{1}\) addr vaddr \(_{2}---\) )
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:

\section*{Using The Defining Word, FILE}
\begin{tabular}{|c|c|}
\hline 0 VARIABLE MY-BUF 78 ALLOT & (Create 80-character RAM buffer) \\
\hline PABS @ 10 + & (PAB starts 10 bytes into VRAM region for PABS and this address will be stored in PAB-ADDR ) \\
\hline MY-BUF & (RAM address to be stored in PAB-BUF ) \\
\hline 6000 & (A free area at 1770h in VRAM to be stored in PAB-VBUF ) \\
\hline FILE JOE & (Whenever the word JOE is executed, the file I/O variables, PAB-ADDR, PAB-BUF and PAB-VBUF, will be set as defined here.) \\
\hline JOE & (Use the file's identifying word (FID) before using any other file I/O words) \\
\hline
\end{tabular}

\section*{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.

\subsection*{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.
\begin{tabular}{|c|c|c|c|}
\hline \multirow[b]{2}{*}{Attribute Type} & \multicolumn{2}{|l|}{Options From} & \multirow[b]{2}{*}{Description} \\
\hline & TI Basic & fbForth & \\
\hline \multirow[t]{2}{*}{File Type} & SEQUENTIAL & SQNTL* & Records may only be accessed in sequential order \\
\hline & RELATIVE & RLTV & Accessed in sequential or random order. Records must be of fixed length \\
\hline Record Type & FIXED VARIABLE & \[
\begin{aligned}
& \text { FXD* } \\
& \text { VRBL }
\end{aligned}
\] & All records in the file are the same length Records in the same file may have different lengths \\
\hline \multirow[t]{2}{*}{Data Type} & DISPLAY & DSPLY* & File contains printable or displayable characters \\
\hline & INTERNAL & INTRNL & File contains data in machine or binary format \\
\hline \multirow[t]{4}{*}{Mode of Operation} & INPUT & INPT & File contents can be read from, but not written to \\
\hline & OUTPUT & OUTPT & File contents can be written to, but not read from \\
\hline & UPDATE & UPDT* & File contents can be written to and read from \\
\hline & APPEND & APPND & Data may be added to the end of the file, but cannot be read \\
\hline
\end{tabular}
* 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 \(\mathbf{F}-\mathbf{D}\) ":
```

F-D" RS232.BA=9600" ok:0
F-D" DSK2.FILE-ABC" ok:0

```

\subsection*{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.
\begin{tabular}{llc} 
From TI Basic & From fbForth & DSR Opcode \\
\hline OPEN & OPN & 0 \\
CLOSE & CLSE & 1 \\
READ & RD & 2 \\
WRITE & WRT & 3 \\
RESTORE & RSTR & 4 \\
LOAD & LD & 5 \\
SAVE & SV & 6 \\
DELETE & DLT & 7 \\
STATUS & STAT & 9
\end{tabular}

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
\[
\text { ( } n \text {--- ) }
\]
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.

\section*{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:
\begin{tabular}{|c|c|c|}
\hline & \multicolumn{2}{|l|}{Status Byte Information When Value is} \\
\hline Bit & 1 & 0 \\
\hline 0 & File does not exist. & File exists. If device is a printer or similar, always 0 . \\
\hline 1 & Protected file. & Unprotected file. \\
\hline 2 & & Reserved for future use. Always 0 . \\
\hline 3 & INTERNAL data type. & DISPLAY data type or program file. \\
\hline 4 & Program file. & Data file. \\
\hline 5 & VARIABLE record length. & FIXED record length. \\
\hline 6 & At physical end of peripheral. No more data can be written. & Not at physical end of peripheral. Always 0 when file not open. \\
\hline 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. \\
\hline
\end{tabular}

Almost all of the file I/O support words of TI Forth and fbForth \(\mathbf{1 . 0}\) 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:

\section*{: 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 ( 837 Ch ) 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)
\begin{tabular}{cll} 
Bits & Contents & Meaning \\
\hline \(\mathbf{0 - 2}\) & Error Code & \(0=\) no error. Error codes are decoded in table below. \\
\(\mathbf{3}\) & Record Type & \(0=\) fixed-length records; \(1=\) variable-length records. \\
\(\mathbf{4}\) & Data Type & \(0=\) DISPLAY; \(1=\) INTERNAL. \\
\(\mathbf{5 - 6}\) & Mode of Operation & \(0=\) UPDATE; \(1=\) OUTPUT; \(2=\) INPUT; \(3=\) APPEND. \\
\(\mathbf{7}\) & File Type & \(0=\) sequential file; \(1=\) relative file.
\end{tabular}

\section*{Error Codes in Bits 0-2 of Flag/Status Byte of PAB}

\section*{Error}

\section*{Code Meaning}

0 No error unless bit 2 of status byte at address 837 Ch 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.e., 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, etc.

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

\subsection*{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

\section*{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.

\subsection*{8.7 File I/O Example 1: Relative Disk File}
\begin{tabular}{l} 
Instruction \\
\hline HEX \\
0 VARIABLE BUFR 3E ALLOT \\
PABS @ A + \\
BUFR 1700 \\
FILE TESTFIL \\
TESTFIL \\
SET-PAB \\
RLTV \\
DSPLY \\
40 REC-LEN \\
F-D" DSK2.TEST"
\end{tabular}

\section*{Comment}

Change number base to hexadecimal
Create space for a 64 byte buffer which will be the PAB-BUF PAB starts 10 bytes into PABS. This will be the PAB-ADDR Place the PAB-BUF and PAB-VBUF on stack in preparation for FILE
Associates the name TESTFIL with these three parameters
File name must be executed before using any other File I/O words
Create PAB skeleton
Make TESTFIL a relative file
Records will contain printable information
Record length is 64 ( 40 h ) bytes
Will create the file descriptor "DSK2.TEST" in the PAB for TESTFIL .
OPN
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
100 DO

TESTDATA is assumed to be the beginning memory address of the information to be written to the file
Want to write \(16(10 \mathrm{~h})\) records
\begin{tabular}{|c|c|c|}
\hline & DUP & Duplicate address \\
\hline & BUFR 40 CMOVE & Move 64 bytes of information into the PAB-BUF \\
\hline & I REC-NO & Place record number into PAB \\
\hline & 40 WRT & Write one 64-byte record to the disk \\
\hline & 40 + & Increment address for next record \\
\hline & LOOP DROP & Clear stack \\
\hline ; & & End definition \\
\hline - . & - • . - & - ' \({ }^{\text {c }}\) \\
\hline FIL-WRT & & Execute writing procedure \\
\hline 4 REC-NO & RD & 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 BUFR \\
\hline BUFR 40 & DUMP & Print out the read information to the monitor. (DUMP routines must be loaded from block 16 of FBLOCKS) \\
\hline CLSE & & Close the file \\
\hline
\end{tabular}

\subsection*{8.8 File I/O Example 2: Sequential RS232 File}
\begin{tabular}{|c|c|}
\hline Instruction & Comment \\
\hline HEX & Change number base to hexadecimal \\
\hline 0 VARIABLE MY-BUF 4E ALLOT & Create an 80-character PAB-BUF \\
\hline PABS @ 30 + & Skip all previous PABs. This will be the PAB-ADDR \\
\hline MY-BUF 1900 & Place the PAB-BUF and PAB-VBUF on stack in preparation for FILE \\
\hline FILE PRNTR & Associates the name PRNTR with these three parameters \\
\hline PRNTR & File name must be executed before using any other File I/O words \\
\hline SET-PAB & Create a PAB skeleton \\
\hline DSPLY & PRNTR will contain printable information \\
\hline SQNTL & PRNTR may be accessed only in sequential order \\
\hline VRBL & Records may have variable lengths \\
\hline 50 REC-LEN & Maximum record length is 80 char. \\
\hline F-D" RS232.BA=9600" or & PRNTR will be an RS232 serial "file" with baud rate \(=\) \\
\hline F-D" PIO" & 9600 or a parallel printer "file". \\
\hline OPN & Open the file \\
\hline . . . . . & . . . . . \\
\hline
\end{tabular}

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:

\footnotetext{
: PRNT FILE-INFO
200 DO

FILE-INFO is assumed to be the beginning memory address of the information to be sent to the printer Will write 32 records
}


\subsection*{8.9 Disk Catalog Utilities}

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

\subsection*{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 .

\section*{Usage: DIR DSK1.}

Example:


\subsection*{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:


\section*{\(9 \quad\) The fbForth 2.0 TMS9900 Assembler}

The assembler supplied with your fbForth 2.0 system is typical of assemblers supplied with figForth 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:

\section*{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.

Before going on, it would be a good idea to familiarize yourself with Chapter 12 "fbForth 2.0 Dictionary Entry Structure" to ensure you understand the structure of fbForth \(\mathbf{2 . 0}\) words.

\subsection*{9.1 TMS9900 Assembly Mnemonics}
\begin{tabular}{|c|c|c|}
\hline A, & JGT, & RTWP, \\
\hline \(A B\), & JH, & S, \\
\hline ABS , & JHE, & SB, \\
\hline AI, & JL, & SBO, \\
\hline ANDI, & JLE, & SBZ, \\
\hline B , & JLT, & SETO, \\
\hline BL, & JMP, & SLA, \\
\hline BLWP, & JNC, & SOC, \\
\hline C, & JNE, & SOCB, \\
\hline CB , & JNO, & SRA, \\
\hline CI, & JOC, & SRC, \\
\hline CKOF, & JOP, & SRL, \\
\hline CKON, & LDCR, & STCR, \\
\hline CLR, & LI, & STST, \\
\hline COC, & LIMI, & STWP, \\
\hline CZC, & LREX, & SWPB, \\
\hline DEC, & LWPI, & SZC, \\
\hline DECT, & MOV, & SZCB, \\
\hline DIV, & MOVB, & TB, \\
\hline IDLE, & MPY, & THEN, \\
\hline INC, & NEG, & X , \\
\hline INCT, & ORI, & XOP, \\
\hline INV, & RSET, & XOR, \\
\hline JEQ, & RT, & \\
\hline
\end{tabular}

These words are available when the assembler is loaded. Only the words C, and R0 (see later) conflict with the existing fbForth \(\mathbf{2 . 0}\) vocabulary.

Most assembly code in fbForth \(\mathbf{2 . 0}\) 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.

\section*{9.2 fbForth 2.0's Workspace Registers}
\begin{tabular}{|c|c|c|}
\hline \multicolumn{2}{|l|}{Register Name} & \multirow[b]{2}{*}{Usage} \\
\hline Original & Alternate & \\
\hline 0 & R0 & \multirow{8}{*}{These registers are available. They are used only within fbForth words written in CODE .} \\
\hline 1 & R1 & \\
\hline 2 & R2 & \\
\hline 3 & R3 & \\
\hline 4 & R4 & \\
\hline 5 & R5 & \\
\hline 6 & R6 & \\
\hline 7 & R7 & \\
\hline UP & R8 & Points to base of User Variable area \\
\hline SP & R9 & Parameter Stack Pointer \\
\hline W & R10 & Inner Interpreter current Word pointer \\
\hline 11 & R11 & Linkage for subroutines in CODE routines \\
\hline 12 & R12 & Used for CRU instructions \\
\hline IP & R13 & Interpretive Pointer \\
\hline RP & R14 & Return Stack Pointer \\
\hline NEXT & R15 & Points to the next instruction fetch routine \\
\hline
\end{tabular}

\subsection*{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 ASM:, DOES>ASM:, CODE:, DOES>CODE: and ;CODE 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 ASM: 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, ; ASM and its synonym, NEXT , . After defining words that use ASM: 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.
The use of CODE, ; CODE and NEXT, , though still supported, is deprecated in favor of the identical but clearer ASM: , DOES>ASM: and ; ASM, respectively. Please keep this in mind when attempting to compare fbForth \(\mathbf{2 . 0}\) code using them with TI Forth code, which, obviously can only use the former.
An Assembly definition begins with ASM: It is followed by assembly mnemonics and terminated with ; ASM. ASM: is used in the following way:
```

ASM: EXAMPLE <assembly mnemonics> ;ASM

```

Equivalently, machine code, which does not require the Assembler, may be used in place of assembly mnemonics with

CODE: EXAMPLE <machine code> ; CODE
Each defines a Forth word named EXAMPLE with an execution procedure defined by the assembly mnemonics or machine code that follow EXAMPLE, which must terminate with ;ASM. The assembly code ends 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

```

Just as with ASM: , assembly code following DOES>ASM: must 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 .

Just as with CODE:, DOES>CODE: can be used to create the same defining word without needing to load the Assembler (see example in § 9.10 ):
```

: DEF-WRD <BUILDS ... DOES>CODE: <machine code> ;CODE

```

\section*{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 TMS 9900 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 \(\mathbf{2 . 0}\) addressing mode words may also be used. The TI Forth code can be used in fbForth 2.0 with no changes.
; ASM is defined as
```

: ;ASM *NEXT B, ;

```
and is equivalent to the following assembly code:
```

B *R15

```

\subsection*{9.4.1 Workspace Register Addressing}

The registers in the fbForth \(\mathbf{2 . 0}\) code below can be referenced directly by number, however, we are using the alternate, easier to read, R designation:
\begin{tabular}{|c|c|c|c|c|}
\hline fbForth 2.0 & TI Forth & \multicolumn{3}{|l|}{Conventional Assembler} \\
\hline HEX & HEX & & & \\
\hline ASM: EX1 & CODE EX1 & & DEF & EX1 \\
\hline R1 R2 A, & 12 A , & EX1 & A & R1, R2 \\
\hline R3 INC, & 3 INC, & & INC & R3 \\
\hline R3 FFFC ANDI, & 3 FFFC ANDI, & & ANDI & R3, >FFFC \\
\hline ; ASM & NEXT, & & B & *R15 \\
\hline
\end{tabular}

\subsection*{9.4.2 Symbolic Memory Addressing}

Symbolic addressing is done with the @() word (Wycove Forth equivalent: @@ ). It is used after the address.
\begin{tabular}{|c|c|c|c|c|}
\hline fbForth 2.0 & TI Forth & \multicolumn{3}{|l|}{Conventional Assembler} \\
\hline 0 VARIABLE VAR1 & 0 VARIABLE VAR1 & VAR1 & BSS & 2 \\
\hline 5 VARIABLE VAR2 & 5 VARIABLE VAR2 & VAR2 & DATA & 5 \\
\hline ASM: EX2 & CODE EX2 & & DEF & EX2 \\
\hline VAR2 @() R1 MOV, & VAR2 @() 1 MOV, & EX2 & MOV & @VAR2,R1 \\
\hline R1 2 SRC, & 12 SRC, & & SRC & R1,2 \\
\hline R1 VAR1 @() S, & 1 VAR1 @() S, & & S & R1,@VAR1 \\
\hline VAR2 @() VAR1 @() SOC, & VAR2 @() VAR1 @() SOC, & & S0C & @VAR2, @VAR1 \\
\hline ; ASM & NEXT, & & B & *R15 \\
\hline
\end{tabular}

\subsection*{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 *? .
\begin{tabular}{lclll} 
fbForth 2.0 & TI Forth & \multicolumn{2}{c}{ Conventional Assembler } \\
\hline 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
\end{tabular}

\subsection*{9.4.4 Workspace Register Indirect Auto-increment Addressing}

Workspace Register Indirect Auto-increment addressing is done with the *?+ word (Wycove Forth equivalent: \(\boldsymbol{*}_{+}\)). 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 *?+ .
\begin{tabular}{lclll} 
fbForth 2.0 & TI Forth & \multicolumn{2}{c}{ Conventional Assembler } \\
\hline 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
\end{tabular}

\subsection*{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 @(?).
\begin{tabular}{|c|c|c|c|c|}
\hline fbForth 2.0 & TI Forth & \multicolumn{3}{|l|}{Conventional Assembler} \\
\hline HEX 2000 CONSTANT XRAM & HEX 2000 CONSTANT XRAM & XRAM & EQU & >2000 \\
\hline ASM: EX5 & CODE EX5 & & DEF & Ex5 \\
\hline XRAM @(R1) R2 MOV, DECIMAL & XRAM 1 @(?) 2 MOV, DECIMAL & Ex5 & MOV & @XRAM(R1),R2 \\
\hline XRAM \(22+\) @ (R2) & XRAM \(22+2\) @(?) & & mov & @XRAM+22(R2),@XRAM+26(R2) \\
\hline XRAM 26 + @(R2) MOV, & XRAM \(26+2\) @(?) MOV, & & & \\
\hline ;ASM & NEXT, & & B & *R15 \\
\hline
\end{tabular}

\subsection*{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 \(\boldsymbol{n}\) :
\begin{tabular}{llll}
\begin{tabular}{lll} 
Register \\
Address
\end{tabular} & Indirect & Indexed & \begin{tabular}{l} 
Indirect \\
Auto-increment
\end{tabular} \\
\hline W & *W & @(W) & *W+ \\
RP & *RP & @(RP) & *RP+ \\
IP & *IP & @(IP) & *IP+ \\
SP & *SP & @(SP) & *SP+ \\
UP & *UP & @(UP) & *UP+ \\
NEXT & *NEXT & @(NEXT) & *NEXT+ \\
Rn & *Rn & @(Rn) & *Rn+
\end{tabular}

\subsection*{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.

\subsection*{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 \(\mathbf{2 . 0}\) 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.

\subsection*{9.7 Assembler Jump Tokens}
\begin{tabular}{|c|c|c|c|}
\hline Token & Comment & Conventional Assembler Used & Machine Code Generated \\
\hline EQ & True if = & JNE & 1600h \\
\hline GT & True if signed > & JGT \$+1 JMP & 1501h 1000h \\
\hline GTE & True if signed \(>\) or \(=\) & JLT & 1100h \\
\hline H & True if unsigned \(>\) & JLE & 1200h \\
\hline HE & True if unsigned \(>\) or \(=\) & JL & 1A00h \\
\hline L & True if unsigned \(<\) & JHE & 1400h \\
\hline LE & True if unsigned \(<\) or \(=\) & JH & 1800h \\
\hline LT & True if signed \(<\) & JLT \$+1 JMP & 1100h 1000h \\
\hline LTE & True if signed \(<\) or \(=\) & JGT & 1500h \\
\hline NC & True if No Carry & JOC & 1800h \\
\hline NE & True if equal bit not set & JEQ & 1300h \\
\hline NO & True if No overflow & JNO \$+1 JMP & 1901h 1000h \\
\hline NP & True if Not odd Parity & JOP & 1700h \\
\hline OC & True if Carry bit is set & JNC & 1700h \\
\hline 00 & True if Overflow & JN0 & 1900h \\
\hline OP & True if Odd Parity & JOP \$+1 JMP & 1c00h 1000h \\
\hline
\end{tabular}

\subsection*{9.8 Assembly Example for Structured Constructs}

The following example is designed to show how these jump tokens and structured constructs are used:
\begin{tabular}{|c|c|c|c|}
\hline fbForth 2.0 & TI Forth & Conventional & Assembler \\
\hline ( GENERALIZED SHIFTER ) & ( GENERALIZED SHIFTER ) & GENERALIZED & SHIFTER \\
\hline ASM: SHIFT & CODE SHIFT & DEF & SHIFT \\
\hline *SP+ R0 MOV, & *SP+ 0 MOV, & SHIFT MOV & *SP+, R0 \\
\hline NE IF, & NE IF, & JEQ & L3 \\
\hline *SP R1 MOV, & *SP 1 MOV, & MOV & *SP, R1 \\
\hline R0 ABS, & 0 ABS, & ABS & R0 \\
\hline GTE IF, & GTE IF, & JLT & L1 \\
\hline R1 R0 SLA, & 10 SLA, & SLA & R1,0 \\
\hline ELSE, & ELSE, & JMP & L2 \\
\hline R1 R0 SRL, & 10 SRL, & L1 SRL & R1,0 \\
\hline THEN, & ENDIF, & & \\
\hline R1 *SP MOV, & 1 *SP MOV, & L2 MOV & R1, *SP \\
\hline THEN, & ENDIF, & & \\
\hline ; ASM & NEXT, & L3 B & *R15 \\
\hline
\end{tabular}

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.

\subsection*{9.9 Assembly Example with DOES>ASM:}

Before giving an example of defining an fbForth \(\mathbf{2 . 0}\) defining word that uses 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 (with DOES>, DOES>ASM: or DOES>CODE:), ASM:, CODE: and CREATE. 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 \(\mathbf{2 . 0}\) 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 \(\mathbf{2 . 0}\) 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 \(\mathbf{X X X}\) with 9 in its parameter field and the address of the execution code of CONSTANT in its code field. fbForth \(\mathbf{2 . 0}\) defines CONSTANT in high-level Forth essentially as
: CONSTANT <BUILDS , DOES> @ ;
Using DOES>ASM : , it could also be defined with Assembler code as
```

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

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

\section*{HEX}
: CONSTANT <BUILDS , DOES>CODE: 0649 C65A ;CODE
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.

\subsection*{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.

Important Note: ASM>CODE (see entry in Appendix D ) can be used to convert words written using ASM: to their CODE: counterparts. Unfortunately, this convenience does not extend to words that use DOES>ASM: .

This section details how you can convert by hand words that use ASM: or DOES>ASM: to CODE: or DOES \(>\) CODE : , respectively. It is much more painful than with ASM>CODE for ASM: to CODE : , but explains the gory details. And, of course, you have no choice with the conversion of DOES>ASM: to DOES>CODE: .

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 fibForth 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 \(\mathbf{2 . 0}\) Assembler is loaded by executing \(\mathbf{2 1}\) 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.

\subsection*{9.10.1 ASM: without the Assembler}

Refer to the example in § 9.4.5 for the following:
5. Use ' to find the pfa of EX5 and dump from the \(p f a\) 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.
6. The last cell should be 045 Fh , corresponding to the ; ASM instruction.
7. Write the high-level part of the word with CODE: instead of ASM: (CODE: EX5) followed by the machine code after EX5 using the dump above to transcribe the hexadecimal value for each cell starting with the first cell (parameter field) and ending with ; CODE (instead of 045Fh) as follows:

HEX
CODE: EX5 C0A1 2000 C8A2 2016 201A ;CODE
You may have noticed that the machine code between CODE: and ; CODE is not compiled with, as it is in TI Forth. This is because CODE: employs its own interpreter to attempt to first convert tokens in the input stream to numbers, whereas the Forth outer interpreter first tries to find the tokens in the dictionary. The upshot of this is that you must now use N \(>\) S between CODE: and ; CODE if you need a number pushed to the stack.
8. 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 and compile them into the definition yourself. In the above example, the constant XRAM was used, so we need to replace the value 2000 h with the reference that put it there. In this case XRAM is used three times to get the cells with \(2000 \mathrm{~h}, 2016 \mathrm{~h}\) and 201 Ah . We need to replace the 2000 h with XRAM, the 2016 h with XRAM \(16+\) and the 201Ah with XRAM 1 A + to get

HEX
CODE: EX5
C0A1 XRAM , C8A2 XRAM N>S \(16+\), XRAM N>S \(1 A+\),
; CODE
which can now be entered in an fbForth 2.0 block to be loaded without the Assembler overhead.
Notice the use of , and \(\mathbf{N}>\mathbf{S}\) in the code above. XRAM is not recognized as a number by CODE: , so you must compile it with , . 16 is a number we want on the stack to add to XRAM before compiling the result, so we must prevent CODE: from compiling it by using \(\mathrm{N}>\mathbf{S}\) to push it to the stack. The same situation obtains for XRAM \(\mathrm{N}>\mathrm{S} 1 \mathrm{~A}+\),
You should test your new version of the word to verify that it is identical to the original assembly version.

\subsection*{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 \(c f a\) 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:
5. Use ' and CFA to find the \(c f a\) 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 \(p f a\) of CONST2 and dump from the \(p f a\) to the end of the word:

HERE ' CONST2 SWAP OVER - DUMP
will dump this to the screen:
AE4A: 71CC 661672540649 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 045 Fh , corresponding to the ; ASM instruction.
8. Write the high-level part of the word through DOES \(>\) ASM: , replacing DOES \(>\) ASM: with DOES \(>\) CODE: , followed by the machine code after 7254h [the \(c f a\) of (;CODE) we found above in (5)]. Use the dump above for guidance to place the hexadecimal value for each cell as follows, replacing 045Fh with ; CODE :
```

HEX
: CONSTANT <BUILDS , DOES>CODE: 0649 C65A ;CODE

```
which can now be entered on an fbForth 2.0 screen to be loaded with only DOES>CODE: and ; CODE 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.

\section*{10 Interrupt Service Routines (ISRs)}

This chapter has been completely rewritten since the June 20, 2015 edition of this manual because, as of fbForth 2.0:8, the fbForth 2.0 ISR is enabled by default so that it may process the new fbForth \(\mathbf{2 . 0}\) speech and sound routines. Though the fbForth \(\mathbf{2 . 0}\) ISR may be disabled by the user as it used to be by default, doing so will disable the new speech and sound routines.
The method of servicing a user's ISR written in Forth is basically the same as in past builds of fbForth 2.0.

\subsection*{10.1 Overview of fbForth 2.0's ISR}

Though the user may disable \(\mathrm{it}^{18}\), fbForth 2.0's ISR is now hooked at startup and is executed for every interrupt. There are three entry points into fbForth 2.0's ISR. Their ALC (Assembly Language Code) labels are INT1, INT2 and INT3.
INT1 is where the console ISR branches at the end of its interrupt processing. It processes any pending speech (started with SAY or STREAM ) and sound (started with PLAY). It then looks to see whether a user ISR is installed in user variable ISR. If so, it modifies the fbForth \(\mathbf{2 . 0}\) inner interpreter's NEXT (R15) to re-enter at INT2.
Re-entry at INT2 will restore NEXT and set up re-entry yet again at INT3 for cleanup just before branching to the user ISR.
When the user ISR finishes, fbForth 2.0's ISR is re-entered at INT3 for cleanup via the inner interpreter. Upon exit, the inner interpreter will resume processing Forth words where it was interrupted.
A user ISR will be executed only if the user has installed an ISR using the steps detailed in § 10.3 "Installing a User ISR".

\subsection*{10.2 A Detailed Look at fbForth 2.0's ISR}

The console ISR branches to the contents of 83C4h (R2 of the console ISR workspace [83C0h]) if it is non-zero. As of fbForth 2.0:8, 83C4h contains the address of ISR entry point INT1 (currently, 3020h) mentioned in the last section. This same entry point is in user variable INTLNK, as well. This means that the console ISR will branch to the fbForth 2.0 ISR with BL *R12 through the GPL workspace ( 83 E 0 h ), R12 containing the ISR's entry point.
Upon entry at INT1 from the console ISR, the fbForth 2.0 ISR does the following:
- Checks for pending speech and sound. If found, the following ISR branch stack is set up and executed:
- Relevant speech ISR address, if speech pending;
- Sound list \#1 ISR address, if pending;
- Sound list \#2 ISR address, if pending;
- fbForth 2.0 ISR return address.

18 fbForth 2.0's ISR may be disabled by zeroing 83C4h with HEX 0 83C4!.
- Restores interrupted bank.
- Checks user variable ISR for a non-zero value, implying a user ISR is installed. If a user ISR is defined, modifies NEXT to re-enter fbForth 2.0's ISR at INT2 at the next branch through NEXT via B *NEXT or B *R15, which will set up to execute the user ISR.
- Exits the fbForth 2.0 ISR by changing to the ISR workspace (83C0h) and returning to the caller of the console ISR.

Upon entry at INT2 (because we have a user ISR defined), the fbForth \(\mathbf{2 . 0}\) ISR does the following:
- Disables interrupts via LIMI 0.
- Disables VDP interrupt.
- Restores NEXT to its value before it was changed at INT1.
- Sets the fbForth "pending interrupt" flag.
- Pushes current IP (next word pointer) to the return stack.
- Changes IP to INT3 for cleanup re-entry to fbForth 2.0 ISR.
- Copies the value in ISR to W (current word pointer) so inner interpreter will execute the user ISR.
- Branches to inner interpreter to execute user ISR via DOEXEC.

Upon entry at INT3 (because we are returning from executing the user ISR), the fbForth 2.0 ISR does the following:
- The inner interpreter actually branches to the address 4 bytes after INT3, which pops the saved IP from the return stack.
- Clears the fbForth "pending interrupt" flag.
- Clears the pending VDP interrupt by reading VDP status.
- Re-enables VDP interrupt.
- Re-enables interrupts via LIMI 2.
- Branches to inner interpreter via NEXT to continue executing the interrupted list of word addresses.

If the user's ISR (see below) is properly installed, fbForth 2.0's ISR, at interrupt, modifies NEXT so that the very next time B \({ }^{*}\) NEXT or B \({ }^{* R 15}\) is executed from fbForth 2.0's workspace, fbForth 2.0's ISR is re-entered to disable interrupts and to insert execution of the user ISR and its cleanup into the fbForth 2.0 inner interpreter's list of execution addresses (cfas).
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 \(\mathbf{2 . 0}\) 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.

\subsection*{10.3 Installing a User ISR}

The user variables ISR and INTLNK are provided to assist the user in using ISRs. Initially, INTLNK contains the address of the fbForth 2.0 ISR handler and ISR is set to 0 to indicate no user ISR. To correctly use user variable ISR, the following steps should be followed:

\section*{Step}

Forth Code
1) Create and test an fbForth 2.0 routine to perform the function. Let's call it MYISR : : MYISR ... ;
2) Clear the fbForth \(\mathbf{2 . 0}\) ISR hook to temporarily disable it: HEX 0 83C4 !
3) Determine the Code Field Address (cfa) of the routine in (1):
4) Write the \(c f a\) from (3) (still on the stack) into user variable

ISR:
5) Write the contents of INTLNK into \(83 C 4 h(33732)\) to re-
ISR:
5) Write the contents of INTLNK into 83C4h (33732) to reenable the fbForth 2.0 ISR: HEX INTLNK @ 83C4 !
' MYISR CFA

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 instruction whose address is in NEXT (as it does at the end of each code word). In addition, the KEY routine has been coded so that it also executes through NEXT after every keyscan whether or not a key has been pressed. The execution of the "NEXT" instruction in the inner interpreter is actually coded in TI Assembler as B *NEXT or B *R15 because fbForth \(\mathbf{2 . 0}\) workspace register 15 (R15 or NEXT) always contains the address of "NEXT" (MOV *IP+,W) except, of course, when we temporarily force its change by installing a user ISR. This executes the same procedure as the fbForth 2.0 Assembler words ; ASM and NEXT, (see Chapter 9"The fbForth 2.0 TMS9900 Assembler").

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, speech and sound because of missed interrupts. In addition it is possible to bring the fbForth \(\mathbf{2 . 0}\) system to a slow crawl by using about \(\mathbf{9 9 \%}\) 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.

\subsection*{10.4 Example of a User ISR: DEMO}

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.)

The problem listed in (1) is overcome by moving PAD in the interrupt routine to eliminate the interference between the foreground and the background task. An example of problem (2) would be attempting to use the built-in number formatting routines, which are quite general and, hence, pay a performance penalty. DEMO performs this conversion rather crudely, but fast enough that there is adequate time remaining in each \(1 / 60\) second to do meaningful computing.

\section*{0 VARIABLE TIMER}
: UP 100 ALLOT ;
: DOWN -100 ALLOT DROP ;

\section*{: DEMO UP}

1 TIMER +! TIMER @
PAD DUP 5 +
DO
010 U/
SWAP 48 +
I C!
- 1 +LOOP

PAD 1+ SCRN_START @ 5 VMBW DOWN ;
(TIMER will hold the current count)
(move HERE and thus PAD up 100 bytes)
(restore PAD to its original location)
(move PAD to avoid conflict)
(increment TIMER, leave on stack)
(ready to loop from PAD +5 down to PAD +1 )
(make positive double, get \(1^{\text {st }}\) digit)
(generate ASCII digit)
(store to PAD )
(decrement loop counter)
(write to screen)
(restore PAD location)

\subsection*{10.4.1 Installing the DEMO ISR}

To install this ISR, the following code should be executed:
```

HEX 0 83C4 ! (clear console ISR hook)
' DEMO CFA (get cfa of the word to be installed as user ISR)
ISR !
INTLNK @
HEX 83C4 !
(clear console ISR hook)
(get $c f a$ of the word to be installed as user ISR)
(place it in user variable ISR )
(get the fbForth 2.0 ISR address to the stack)
(re-install fbForth 2.0 ISR into console ISR hook)
(Note: the cfa of DEMO must be in user variable ISR before writing to $\mathbf{8 3 C} \mathbf{C h}$ )

```

\subsection*{10.4.2 Uninstalling the DEMO ISR}

To reverse the installation of the ISR, the following code should be executed:
```

HEX 0 83C4 ! (clear console ISR hook)
0 ISR !
INTLNK @
HEX 83C4 !
(clear console ISR hook)
(disable user ISR by zeroing user variable ISR )
(get the fbForth 2.0 ISR address to the stack)
(re-install fbForth 2.0 ISR into console ISR hook)

```

\subsection*{10.5 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.

\section*{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.

\subsection*{11.1 BSAVE and BLOAD}

BSAVE ( \(\left.a d d r b l k_{1}---b l k_{2}\right)\)
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 \(a d d r\) in the dictionary image to be saved to disk.
2) The Forth block number \(b l k_{1}\) 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 \(b l k_{2}\) of the first available Forth block after the image.

Each Forth block of the saved image has the following format:
\begin{tabular}{ll} 
Byte \# & Contents \\
\hline \(0-1\) & \begin{tabular}{l} 
Address at which the first image byte of this Forth block \\
will be placed.
\end{tabular} \\
\(2-3\) & \begin{tabular}{l} 
DP for this memory image. \\
\(4-5\)
\end{tabular} \\
\(6-7\) & Contents of CURRENT . \\
\(8-9\) & Contents of CURRENT @ . \\
\(10-11\) & Contents of CONTEXT CONTEXT @ . \\
\(12-13\) & Contents of VOC-LINK . \\
\(14-15\) & Pointer to last word defined in Forth vocabulary. \\
\(16-17\) & Pointer to last word defined in Assembler vocabulary. \\
18 & The letter ' \(\mathfrak{t}\) '. \\
19 & The letter ' i '. \\
\(20-23\) & Not used. \\
\(24-1023\) & Up to 1000 bytes of the memory image.
\end{tabular}

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 \(18^{\text {th }}\) and \(19^{\text {th }}\) 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 usersupplied 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.

\subsection*{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 \(\mathbf{2 . 0}\) 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 \(\mathbf{2 . 0}\) 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.

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 ; \(\mathbf{S}\) exits loading and interpreting the block. This will load your BSAVEd system and it will happen a lot faster than loading the text blocks because they now don't need to be interpreted.

\subsection*{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
ELSEOF -1 ENDOF \ wrong overlay choice!
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 VLIST to get the name of the word immediately following TASK. Remember that VLIST lists the dictionary from HERE back to older words.
b) 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.)

\subsection*{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 (1st 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 \(\mathbf{2 . 0}\) 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.

\subsection*{11.2 Conditional Loads}

\section*{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.

\section*{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 FOO
This instruction, for example, will load fbForth 2.0 block 27 only if a dictionary search via (FIND) fails to find FOO . FOO 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 \(\mathbf{2 . 0}\) block. This is done by using the command:

\section*{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 ; \(\mathbf{S}\) had been encountered.
Caution must be exercised when using BASE->R and \(\mathbf{R}->\) 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.

\subsection*{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 \(\mathbf{2 . 0}\) 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 \(a d d r\) in all of these words such that it is the actual CRU base bit number. For TB , SBO 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 SBO 9:
\[
\begin{array}{ll}
\text { LI } & \mathrm{R} 12,>1100 \\
\text { SB0 } & 9
\end{array}
\]
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 SBO :

HEX 889 SBO ok:0
which actually executes the following TMS9900 Assembler code:
\begin{tabular}{lll} 
M0V & *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
\end{tabular}

LDCR \(\quad\left(n_{1} n_{2} a d d r---\right)\)
Performs a TMS9900 LDCR instruction. The CRU base address \(a d d r\) 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 \(n_{2}\) AND 0 Fh . 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
\[
\left(n_{1} a d d r---n_{2}\right)
\]

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.
TB ( addr --- flag )
TB performs the TMS9900 TB instruction. The bit at CRU address addr is tested by this instruction. Its value (flag \(=1 \mid 0)\) is returned to the stack. The CRU base address \(a d d r\) 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.
( addr --- )

This word expects to find on the stack the CRU address \(a d d r\) 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, SB0 0, to effect setting the bit.
( addr --- )
This word expects to find on the stack the CRU address \(a d d r\) 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.

\subsection*{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
\[
\left(n_{1} n_{2}--n_{1} n_{2} n_{1} n_{2}\right)
\]

Duplicate the top two numbers on the stack.

\section*{2DROP \(\quad\left(n_{1} n_{2}---\right)\)}

Drop the top two numbers from the stack.

NIP

TUCK

CELLS
\[
\left(n_{1} n_{2}---n_{2}\right)
\]

Remove from the stack the number that is under the top number.
\[
\left(n_{1} n_{2}---n_{2} n_{1} n_{2}\right)
\]

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

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

\section*{PICK \(\quad(+n---[n])\)}

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 \(\quad([n] \ldots[0]+n---[n-1] \ldots[0][n])\)
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 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 .
\[
\left(n_{1} n_{2} n_{3}--- \text { flag }\right)
\]

Result flag is true (1) if \(n_{2} \leq n_{1}<n_{3}\) and false (0) otherwise.
<> \(\quad\left(n_{1} n_{2}--\right.\) 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


EXIT is a synonym for ; \(\mathbf{S}\), which stops interpretation of a Forth block or ends the current word's execution and returns to the calling procedure.

\section*{12 fbForth 2.0 Dictionary Entry Structure}

The structure of an entry (a Forth word) in the fbForth \(\mathbf{2 . 0}\) dictionary is briefly described in this chapter to give the reader a better understanding of fbForth \(\mathbf{2 . 0}\) 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).

\subsection*{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.

\subsection*{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 \(n f a\) points to this field and may be retrieved by pushing the \(p f a\) (see § 12.4 ) onto the stack and executing NFA.

The name field is a packed character string (see footnote 4 on page 22) 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 ( 80 h ), the precedence bit ( 40 h ) and the smudge
bit \((20 \mathrm{~h})\). 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 \(\mathbf{2 . 0}\) 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 ( 80 h ) 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 \((+1 \mid-1)\) 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.

\subsection*{12.3 Code Field}

The code field immediately follows the last cell of the name field. The code field address \(c f a\) points to this field and may be retrieved by pushing the \(p f a\) (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:
\begin{tabular}{l|l|l}
\begin{tabular}{l} 
Word \\
Defined by
\end{tabular} & \begin{tabular}{l} 
Code Field Contains \\
Address of
\end{tabular} & What the Runtime Code Does \\
\hline VARIABLE & Runtime code of VARIABLE & Pushes word's \(p f a\) onto stack \\
CONSTANT & Runtime code of CONSTANT & \begin{tabular}{l} 
Pushes contents of word's pfa onto \\
stack
\end{tabular} \\
\(\mathbf{S}\) & Runtime code of : & \begin{tabular}{l} 
Executes the list of previously \\
defined words, the addresses of \\
which are stored beginning at this \\
word's \(p f a\)
\end{tabular} \\
CODE & \(p f a\) of word & \begin{tabular}{l} 
Executes machine code stored \\
beginning at this word's \(p f a\)
\end{tabular} \\
ASM: & \(p f a\) of word & \begin{tabular}{l} 
Executes machine code stored \\
beginning at this word's \(p f a\)
\end{tabular}
\end{tabular}

\subsection*{12.4 Parameter Field}

The parameter field follows the code field. The parameter field address \(p f a\) points to this address, which can be retrieved by using ' :
' cccc
where \(\mathbf{c c c c}\) 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 \(p f a\) 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 \(p f a\), so it is more suited to colon definitions:
```

-FIND cccc DROP DROP

```

If you know only the \(n f a\), you can retrieve the \(p f a\) by executing PFA .
The contents of the parameter field depend on the type of word defined. The following table shows common situations:
\begin{tabular}{l|l} 
Word Defined by & Parameter Field Contains \\
\hline VARIABLE & Value of variable \\
CONSTANT & \begin{tabular}{l} 
Value of constant
\end{tabular} \\
\(\mathbf{:}\) & \begin{tabular}{l} 
Mostly a list of the addresses (usually their \(c f a s)\) of \\
previously defined words that comprise this word's \\
definition
\end{tabular} \\
CODE & \begin{tabular}{l} 
Machine code comprising this word's runtime code \\
Machine code comprising this word's runtime code
\end{tabular} \\
ASM: & Man
\end{tabular}

\subsection*{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 32 KiB 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 \(c f a\); the second, the \(p f a\); the third, the address of the previous word's \(p f a\) 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-pfapointer chain, mentioned in the previous paragraph. The only way to find the \(n f a\), given a \(p f a\) 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 \(n f a\).

To find the lfa, LFA first finds the \(n f a\) via NFA and then backs up two bytes, i.e., lfa \(=n f a-2\).
Finally, CFA finds the \(c f a\) from the \(p f a\) in the same way for both ROM and RAM: \(c f a=p f a-2\).

\section*{13 Screen Fonts and the Font Editor}

Words introduced in this chapter:
FONTED FNT SCRFNT USEFFL

\subsection*{13.1 Screen Font Changes as of fbForth 2.0:8}

The default Screen Font File with descenders for ASCII character values \(0-127\) (1024 bytes) is no longer in ROM. Now, it is to be found in DSK1.FBFONT unless the default disk has been changed at bootup. The boot DSK \#, \(n\), is saved as the \(5^{\text {th }}\) byte of the packed string for the default blocks filename, "DSK \(n\).FBLOCKS", the \(1^{\text {st }}\) byte of which is the string length.

At powerup,
- SCRFNT is set to its new default value of -1 ;
- The default font is loaded from DSK \(n\).FBFONT by FNT .

At non-powerup COLD , the font file loaded before COLD was invoked is reloaded, unless the user changed the default value of SCRFNT to 0 . The default value of SCRFNT can be changed to 0 to force loading the console font, with its small caps for lowercase, with the following code:

0 UCONS\$ @ 68 + !
UCONS\$ is the address of the default-value table of User Variables and 68 (44h) is SCRFNT 's position in the table.

If the default font file cannot be found,
- SCRFNT and its default value are set to 0 ;
- The console font, with small caps for lowercase, is loaded.

If SCRFNT \(\neq 0\), FNT loads the default font file, the PAB for which follows the fbForth \(\mathbf{2 . 0}\) disk buffer, DISK_BUF , in VRAM.

The user can change the default font to come from a binary font file of the user's choosing with USEFFL. USEFFL will set up the font-file PAB (immediately follows DISK_BUF in VRAM). The default font filename will be copied to the font PAB in VRAM.

The fbForth 2.0 word FNT loads either the default font file (can be changed by user) or the console font into the Pattern Descriptor Table (PDT) depending on the value of the user variable SCRFNT. The default font is loaded from DSK1.FBFONT by FNT (or from DSK \(n\).FBFONT if key \(n\) is held down) at fbForth \(\mathbf{2 . 0}\) startup because SCRFNT \(=-1\) at startup. The fbForth 2.0 system default font contains the patterns for ASCII character codes \(0-127\). The font pattern for each character is 8 bytes, which means that 1 KiB 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 (re)loading the console screen font. Changing SCRFNT back to a non-zero value will switch font loading to the currently stored font-file name, be it the system or user font file.

\subsection*{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 2 KiB , i.e., it should not code for more than 256 characters. Attempting to load font files larger than 2 KiB 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 1 KiB (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:

\section*{PDT 1024 + 10240 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:

\section*{USEFFL DSK1.MYFONT}

Once USEFFL executes without an error, the user's font file will be installed at the next execution of FNT, unless SCRFNT \(=0\).

\subsection*{13.3 Using the Font Editor}

Typing FONTED opens the font editor with 2 KiB of the current font loaded from the PDT and with the uppercase 'A' (ASCII 65) in the edit box. A full 2 KiB 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:
\begin{tabular}{ll} 
Function & Keystroke \\
\hline <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 \\
<CTRL+W> & Toggle output file size between 1 KiB (ASCII 1 - 127) and 2 KiB \\
& (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
\end{tabular}

To aid in editing, the current pattern is shown actual size in a \(6 x 8\)-pixel, text-mode character box and an \(8 \times 8\)-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 \(0-127\) ( 1 KiB file) or all patterns \(0-255\) ( 2 KiB file). The blue ' + ' character marks which of these occurs with the next file save and is toggled with <CTRL+W>.

\subsection*{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:


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 \(3 \times 7\) matrix was quite a challenge. The ' \(\&\) ' should probably be re-designed.
To design your own \(3 \times 7\) font, use a \(4 \times 8\)-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, ( \({ }^{\wedge} 0\) ) (Shift+0), on line 5 is a substitute for ( ) ) , a syntax error]:
\begin{tabular}{|c|c|c|c|c|c|c|c|c|c|}
\hline \multicolumn{10}{|l|}{BLOCK \#10} \\
\hline \multicolumn{3}{|r|}{0 ( DEFINITIONS} & \multicolumn{2}{|l|}{FOR true} & TINY & \multicolumn{2}{|l|}{CHARACTERS} & ) BA & >R HEX \\
\hline 1 & OEEE & VARIABLE & TCHAR & DATA & & & & & \\
\hline 2 & 0000 & 0000 & 0444 & 4404 ( ! ) & 0AA0 & 0000 & & 08AE & AEA2 \\
\hline 3 & 04EC & 46E4 ( \$) & 0A24 4 & 448A ( \%) & 06AC & 4A86 & \&) & 0480 & 0000 \\
\hline 4 & 0248 & 8842 ( () & 08422 & 2248 ( ^0) & 04EE & 4000 & & 0044 & E440 ( + \\
\hline 5 & 0000 & 0048 ( , ) & 0000 E & E000 ( -) & 0000 & 0004 & & 0224 & 4488 ( /) \\
\hline 6 & 04AA & EAA4 ( 0) & 04C4 4 & 4444 ( 1) & 04A2 & 488E & 2) & 0 C 22 & C22C ( 3) \\
\hline 7 & 02AA & AE22 ( 4) & 0E8C 2 & 222C ( 5) & 0688 & CAA4 & & 0 E 22 & 4488 ( 7) \\
\hline 8 & 04AA & 4AA4 ( 8) & 04AA 6 & 622C ( 9) & 0004 & 0040 & & 0004 & 0048 \\
\hline 9 & 0024 & 8420 ( <) & 000E 0 & 0E00 ( =) & 0084 & 2480 & >) & 04A2 & 4404 ( ? ) \\
\hline 10 & 04AE & AE86 ( @) & 04AA E & EAAA ( A) & 0CAA & CAAC & B) & 0688 & 8886 ( C) \\
\hline 11 & 0CAA & AAAC ( D) & 0E88 C & C88E ( E) & 0E88 & C888 & & 04A8 & 8AA6 ( G) \\
\hline 12 & 0AAA & EAAA ( H) & 0E44 4 & 444E ( I) & 0222 & 22A4 & & 0AAC & CAAA ( K) \\
\hline 13 & 0888 & 888E ( L) & OAEE A & AAAA ( M) & OAAE E & EEAA & & 0EAA & AAAE ( 0) \\
\hline 14 & 0CAA & C888 ( P) & 0EAA A & AAEC ( Q) & 0CAA & CAAA & & 0688 & 422C ( S \\
\hline 15 & & & & & & & & & \\
\hline
\end{tabular}
```

BLOCK \#11
0 ( DEFINITIONS FOR true lowercase TINY CHARACTERS concluded)
1 0E44 4444 ( T) 0AAA AAAE ( U) OAAA AA44 ( V) 0AAA AEEA ( W)
2 0AA4 44AA ( X) 0AAA E444 ( Y) 0E24 488E ( Z) 0644 4446 ( [)
30884 4422 ( \) 0C44 444C ( ]) 044A A000 ( \$) 0000 000F ( _)
40420 0000 ( `) 000E 2EAE ( a) 088C AAAC ( b) 0006 8886 ( c)
50226 AAA6 ( d) 0004 AE86 ( e) 0688 E888 ( f) 0006 A62C ( g)
6 088C AAAA ( h) 0404 4442 ( i) 0202 22A4 ( j) 088A ACAA ( k)
70444 4444 ( l) 000A EEAA ( m) 0008 EAAA ( n) 0004 AAA4 ( o)
8 000C AC88 ( p) 0006 A622 ( q) 0008 E888 ( r) 0006 842C ( s)
9 044E 4442 ( t) 000A AAA6 ( u) 000A AAA4 ( v) 000A AEEA ( w)
10 000A A4AA ( x) 000A A62C ( y) 000E 248E ( z) 0644 8446 ( {)
11 0444 0444 ( |) 0C44 244C ( }) 02E8 0000 ( ~) 0EEE EEEE ( DEL)
12 ]DATA DROP DROP
1 3
14 DECIMAL TCHAR 15 BLOCK 192 + 194 MOVE FLUSH FORGET TCHAR
15
R->BASE

```

\section*{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 1.2 (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 1.2 (cf. his website) for this chapter. In addition, the author rendered some minor assistance in this fbForth- \(\mathbf{2 . 0}\) 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 \({ }^{19}\) presented at the end of this paper has been modified where appropriate for compatibility with the Forth- \(83^{20}\) standard, and, specifically, TurboForth V1.2. The original ANS paper can be downloaded as a PDF:
turboforth.net/downloads/docs/ANS_String_Lib.pdf. ]

\section*{Mark Wills' abstract for TurboForth 1.2:}

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 \(83^{20}\) 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.

\subsection*{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.

\subsection*{14.1.1 Coding Conventions}

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

19 The code for the fbForth \(\mathbf{2 . 0}\) version of the String Stack Library is located in Appendix J starting at block 42.
20 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\$).

\subsection*{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,

\section*{\$CONST ( max_len IS:name" --- )}

\subsection*{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:0}\mp@subsup{}{}{21

```
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.

\subsection*{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

21 Note that computer responses are underlined as is the case here for ok: \(\mathbf{0}\).

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.

\subsection*{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 \(r_{1}--\) str \(_{1}\) str \(r_{1}\) ) "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 \(r_{1}\) str - -- str \(r_{2}\) str \(r_{1}\) )"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.

NIP\$ ( --- ) (SS: str str \(_{2}---\) str \(_{2}\) ) "nip string"
The word NIP\$ removes the string underneath the topmost string from the string stack.
For example,
\[
\begin{aligned}
& \text { \$" red" ok:0 } \\
& \text { \$" blue" ok:0 }
\end{aligned}
\]

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

\section*{NIP\$}

At this point, "red" has been removed from the string stack, leaving "blue" as the topmost string.
OVER\$ ( --- ) (SS: \(\left.s t r_{1} s t r_{2}---s t r_{1} s t r_{2} s t r_{1}\right)\) "over string"
The word OVER\$ pushes a copy of the string \(s t r_{1}\) to the top of the string stack, above \(s t r_{2}\). 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: \(s t r_{1} s t r_{2} s t r_{3}--\) str \(\left._{2} s t r_{3} s t r_{1}\right)\) "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 str \(_{2}\) str \(r_{3}--\) str \(_{3}\) str \(r_{1}\) str \(r_{2}\) "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:

\section*{COLOR . \(\$\) CONST red ok: 0}
+\$ (--- )
( SS: str str \(_{2}-\) str \(r_{1} \& s t r_{2}\) ) "concatenate strings"
The word \(\boldsymbol{+}\) \$ 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 - -- str \(_{1} s t r_{2}\) ) "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 \(\left(s t r_{1}\right)\) is retained. Note also that the first character in the string (the leftmost character) is character number 0 .
LEFT\$ (len --- ) (SS: str \(\left.---s t r_{1} s t r_{2}\right)\) "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 - --- \(s t r_{1} s t r_{2}\) ) "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 8 th 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.

\section*{}

The word REPLACE\$ searches string \(s t r_{2}\) for the first occurrence of string \(s t r_{3}\). If it is found, it is replaced with the string \(s t r_{1}\), the position of \(s t r_{3}\) within \(s t r_{2}\) is pushed to the data stack, \(s t r_{1}\) and \(s t r_{3}\) are removed from the string stack and the new string \(s t r_{4}\) is left on the string stack. For example,
```

512 INITE. ok:G
\, \$12 INITE" ok:0
SiS
Index:Length:String
Allocated stack space: }16\mathrm{ bytes
Stack space remaining: 412 bytes
ok:0

```

If the search string \(s t r_{3}\) is not found, -1 is pushed to the data stack, \(s t r_{1}\) and \(s t r_{2}\) are left on the string stack, ready for another search if desired.

\section*{. \$ ( --- ) ( 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 \(r_{1}--\) 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: stri --- 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 leading 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 \(r_{2}\) )"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 - -- str \(r_{2}\) "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 str \(_{2}---\) str \(_{1}\) str \(r_{2}\) )"compare strings"
The word CMP\$ performs a case-sensitive comparison of the topmost two strings on the string stack and returns -1 if \(s t r_{1}<s t r_{2}, 0\) if \(s t r_{1}=s t r_{2}\) and +1 if \(s t r_{1}>s t r_{2}\). 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\$ . \(\underline{0}\) ok: 0
Displays " 0 " since the strings are identical.
\$" hell" \$" hello" CMP\$ . 正 ok:0
Displays " -1 " since the first string is less than the second.
A case in-sensitive comparison can easily be built as follows:
: CMPCI\$ ( --- flag ) ( SS: str \(_{1}\) str \(_{2}--\) str \(_{1}\) str \(_{2}\) )
OVER\$ OVER\$ UCASE\$ SWAP\$ UCASE\$ CMP\$ DROP\$ DROP\$ ; ok: 0
The above code creates copies of \(\operatorname{str}_{1}\) and str \(_{2}\) (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 \(s t r_{1}\) and \(s t r_{2}\) are removed from the string stack; thus, \(s t r_{1}\) and \(s t r_{2}\) 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" \(\underline{\text { ok: } 0}\)
\$" red" ok: 0
2 PICK \(\$ \underline{\text { ok: } 0}\)

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: -21474836482147483647.

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. g900 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 \(\$ \mathbf{S}\) 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\$.

\section*{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.

\subsection*{14.4 The String Stack}

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

\subsection*{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.

\subsection*{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:
\[
\begin{array}{l|l}
\text { Actual length }(1 \text { cell }) & \text { String payload ( } 1 \text { char=1 byte) } \\
\hline
\end{array}
\]

\subsection*{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:
\begin{tabular}{|c|c|c|}
\hline \begin{tabular}{c} 
Maximum length \\
\((1\) cell \()\)
\end{tabular} & \begin{tabular}{c} 
Actual length \\
\((1\) cell \()\)
\end{tabular} & \begin{tabular}{c} 
String payload \\
\((1\) char= \(=1\) byte \()\)
\end{tabular} \\
\hline
\end{tabular}

\subsection*{14.8 Throw Codes}

The words in the library perform sanity checks on input parameters where necessary. In particular, the string stack, being statically ALLOTed 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:
\begin{tabular}{|c|c|c|c|c|}
\hline Throw Code & Nature of Error & \multicolumn{3}{|c|}{Thrown By} \\
\hline 9900 & String stack underflow & \multicolumn{3}{|l|}{(set\$SP)} \\
\hline 9901 & String too large to assign & \multicolumn{3}{|l|}{|:="} \\
\hline 9902 & String stack is empty & \begin{tabular}{l}
PICK\$ \\
>\$CONST \\
RIGHT\$ \\
REV\$ \\
UCASE \(\$\)
\end{tabular} & \begin{tabular}{l}
DUP\$ \\
MID\$ \\
FINDC\$ \\
LTRIM\$ \\
LCASE\$
\end{tabular} & \begin{tabular}{l}
LEN\$ \\
LEFT\$ . \$ RTRIM\$ DROP\$
\end{tabular} \\
\hline 9903 & Need at least 2 strings on string stack & SWAP\$ FIND\$ & \[
\begin{aligned}
& \text { NIP\$ } \\
& \text { CMP\$ }
\end{aligned}
\] & \[
\begin{aligned}
& \text { OVER\$ } \\
& +\$
\end{aligned}
\] \\
\hline 9904 & String too large for string constant & \multicolumn{3}{|l|}{>\$CONST} \\
\hline 9905 & Illegal LEN value & MID\$ & LEFT\$ & RIGHT\$ \\
\hline 9906 & Need at least 3 strings on string stack & ROT\$ & -ROT\$ & REPLACE\$ \\
\hline 9907 & String is not a legal number & \multicolumn{3}{|l|}{VAL\$} \\
\hline 9999 & String stack not initialized & \multicolumn{3}{|l|}{any THROW if (\$sSize) =0} \\
\hline
\end{tabular}

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.

\subsection*{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.

\section*{15 TI Forth Block Utilities}

Words introduced in this chapter:

\section*{TIF2FBF TIFBLK TIFIDX TIFVU}

The TI Forth Block Utilities are not part of the resident dictionary so must be loaded from FBLOCKS (see current FBLOCKS file MENU for TI Forth Block Utilities). They are provided to make it easy to view TI Forth blocks (called "screens" in TI Forth), index lines of a range of blocks and copy a range of blocks to an fbForth \(\mathbf{2 . 0}\) blocks file. The utilities listed in the first three sections below perform these functions individually. The last section presents a browser/copier that is patterned after the fbForth \(\mathbf{2 . 0}\) block editors.
Note that "IS:" is short for "Inout Stream".

\subsection*{15.1 TIFBLK: Display TI Forth Block}

\section*{TIFBLK (IS:blk DSKn)}

TIFBLK displays block blk from disk DSKn. The display may be paused/resumed by tapping any key except <BREAK>, which will abort the display. The display is automatically paused if the block cannot be displayed all at once.
The following shows the first screen of a Text mode example displayed with the Forth code just before the screen shot:

TIFBLK 11 DSK2
```

01 (JDRAW continued...)
1: BASE->R HEX
F2! : JUP_CUR JPEN @ IF 5 0 SPRCDL ELSE
HAR ; JER_CUR F8900 H0C0 8000 0000 10 SPC
HAR ;:JDR_CUR F8F0 E0C0 8000 0000 10 SPC
HAR; MENUIF F 1 SPRPUT 1F F 2 SPRPUT F
5, MMENUF F 1 SPRPUT 1F F 2 SPRPUT F
G!IPEN:QEGIN KEY DUP CASE ( Toggle DMOD
DE!!
IF JER_CUR ELSE JDR_CU
R ENDIF ENDDF

```


The display was paused after twelve lines were displayed due to wrapping of 64-character lines on the 40 -character display. Tapping a key will continue the display of the remaining four lines.

The following is the same example in Text80 mode using the same Forth code as above:


\subsection*{15.2 TIFIDX: Display TI Forth Index Lines}

TIFIDX ( IS:strtBlk endBlk DSKn )
TIFIDX displays the index lines (first lines) of a range of TI Forth blocks (strtBlk to endBlk) from disk DSKn. The display may be paused/resumed by tapping any key except <BREAK>, which will abort the display. The display is automatically paused if the block cannot be displayed all at once.

The following shows the first screen of a Text mode example:
TIFIDX 1015 DSK2
```

    101 CR ." LDADing JDRFW---" CR ." Joy
    stick drawing program...",
121 (JDRAW continued...)
13: (JDRFW continued...)
14!
15!
...done ok:0

```

The index line, (line \#0) of each block from block \#10 - \#15 is listed above. Had more than 12 blocks ( 64 characters each index line) been selected, the display would have paused as for TIFBLK in the previous section.

The following is the same example in Text80 mode:
```

101 CR " LDADing JDRPW|---" CR ," Joystick drawing program..."
(JJRFIW continued...)
JDRRW}\mathrm{ continued...
14:
...done ok:0

```

\subsection*{15.3 TIF2FBF: Copy TI Forth Blocks to fbForth Blocks}

\section*{TIF2FBF ( IS:srcStrtBlk srcEndBlk DSKn dstStrtBlk dstFile)}

TIF2FBF functions in much the same way as CPYBLK. The format of the command is the same except that the source is \(D S K n\), not a filename. The \(n\) of \(D S K n\) is the disk number of the TI Forth disk. The destination dstFile must be the name of an existing blocks file (see MKBFL to create one). The following command will copy blocks \(4-7\) from TI Forth DSK2 to blocks \(10-13\) of DSK1.MYBLOCKS:

TIF2FBF 47 DSK2 10 DSK1.MYBLOCKS

\subsection*{15.4 TIFVU: TI Forth Browser/Copier}

TIFVU (IS:blk DSKn)
Browse TI Forth blocks and, optionally, copy a range of blocks to an fbForth blocks file. The browser is interactive with the following functions:
\begin{tabular}{ll} 
Key & Function \\
\hline\(<F C T N+4>\) & View next block. \\
<FCTN \(+6>\) & View previous block. \\
<FCTN + D> & View the next panel for Text mode-ignored in Text80 mode. \\
<FCTN + S> & View the previous panel for Text mode-ignored in Text80 mode. \\
<FCTN + T> & View a specific TI Forth block number. \\
<FCTN + F> & Specify a destination fbForth block number for next copy. \\
<CTRL + F> & Specify a destination fbForth blocks file, which must already exist. \\
<CTRL + S> & \begin{tabular}{l} 
Copy a range of blocks starting from the displayed TI Forth block to the \\
displayed destination fbForth block. You are prompted for the number of
\end{tabular} \\
& blocks to copy after selecting this command.
\end{tabular}

Following is an example of the browser/copier in Text mode, which shows three panels of the same block:

TIFVU 12 DSK2


Left panel showing columns \(0-33\).

Middle panel showing columns \(15-48\).


F4: +Block F6:-Block FI: +Panel FS: -Panel

Right panel showing columns 30-63.
 FT: TI\# FF: fb\# ^F:BlkFil ^S:TI>fb F9:xit

And, here is the same example in Text80 mode:


\section*{16 Speech Words}

Words introduced in this chapter:
SAY STREAM TALKING?
The words SAY, STREAM and TALKING? in this chapter were ported from TurboForth code courtesy of Mark Wills.
A TI-99/4A equipped with a TI Speech Synthesizer module can be made to talk by sending the Speech Synthesizer commands that include
- Word data addresses of words in the Speech Synthesizer's resident vocabulary or
- Raw speech data.

The fbForth 2.0 speech words require the system ISR to be active, which is the default. See Chapter 10 "Interrupt Service Routines (ISRs)" for how this works.

Consult § 22 of the Editor/Assembler Manual for a detailed discussion of speech processing.

\subsection*{16.1 Testing the State of the Speech Synthesizer}

Use TALKING? to test whether the Speech Synthesizer is busy before using SAY or STREAM in following sections.
TALKING? (--- flag)
TALKING? returns flag \(=0\) if the Speech Synthesizer is idle, otherwise, flag \(=1\).

\subsection*{16.2 Using the Speech Synthesizer's Resident Vocabulary}

To have the Speech Synthesizer speak words from its resident vocabulary (see table below), it is sufficient to use SAY , described here:
SAY ( \(a d d r n---\) )
SAY needs on the stack the address \(a d d r\) of a block of Speech Synthesizer ROM speech addresses and the number \(n\) of those addresses. This can be accomplished with DATA[ ... ]DATA .
This example uses addresses from the table below to say, "Do not be so negative.":
HEX DATA[ 2480 4AAB 1A42 6153 48DC ]DATA SAY ok:0
It is a good idea to use TALKING? before using SAY in word definitions to insure the Speech Synthesizer is not busy.

\subsection*{16.3 The Speech Synthesizer's Resident Vocabulary}

The following table of phrases and addresses from § 24.6 "Speech Synthesizer Resident Vocabulary" in the Editor/Assembler Manual is included here for your convenience:
\begin{tabular}{|c|c|c|c|c|c|}
\hline Phrase & Address & Phrase & Address & Phrase & Address \\
\hline - (negative) & 48DC & 2 & 145C & 6 & 15A8 \\
\hline + (positive) & 51B3 & 3 & 149A & 7 & 15E8 \\
\hline . (point) & 50EC & 4 & 14E7 & 8 & 1637 \\
\hline 0 & 13 C 3 & 5 & 1531 & 9 & 1664 \\
\hline 1 & 1409 & & & & \\
\hline A (ay) & 16E4 & ALL & 1807 & ANY & 1962 \\
\hline A1 (uh) & 1700 & AM & 1830 & ARE & 556E \\
\hline ABOUT & 1714 & AN & 1876 & AS & 19A7 \\
\hline AFTER & 1769 & AND & 18AC & ASSUME & 19E8 \\
\hline AGAIN & 17A5 & ANSWER & 1913 & AT & 1A25 \\
\hline B & 1A42 & BLACK & 1B47 & BUT & 1 C 20 \\
\hline BACK & 1A64 & BLUE & 1B8A & BUY & \(1 \mathrm{C48}\) \\
\hline BASE & 1A8F & BOTH & 1BB6 & BY & \(1 \mathrm{C48}\) \\
\hline BE & 1A42 & BOTTOM & 1BEA & BYE & 1C48 \\
\hline BETWEEN & 1ADE & & & & \\
\hline C & 1C86 & COLOR & 1E20 & COMPUTER & 2034 \\
\hline CAN & 1CD9 & COME & 1E54 & CONNECTED & 208B \\
\hline CASSETTE & 1D10 & COMES & 1E87 & CONSOLE & 20F3 \\
\hline CENTER & 1D47 & COMMA & 1EDE & CORRECT & 213 C \\
\hline CHECK & 1D82 & COMMAND & 1F1A & COURSE & 2182 \\
\hline CHOICE & 1DA2 & COMPLETE & 1F71 & CYAN & 21C0 \\
\hline CLEAR & 1DE6 & COMPLETED & 1FCD & & \\
\hline D & 2203 & DIFFERENT & 23 C 4 & DONE & 253E \\
\hline DATA & 223 C & DISKETTE & 242D & DOUBLE & 2599 \\
\hline DECIDE & 2294 & DO & 2480 & DOWN & 25D3 \\
\hline DEVICE & 22FD & DOES & 24B3 & DRAW & 2612 \\
\hline DID & 2366 & DOING & 24EA & DRAWING & 2668 \\
\hline E & 26CB & ELEVEN & 2579 & ENTER & 28AD \\
\hline EACH & 26F0 & ELSE & 27B6 & ERROR & 28EF \\
\hline EIGHT & 1637 & END & 27F5 & EXACTLY & 2937 \\
\hline EIGHTY & 2723 & ENDS & 2866 & EYE & 3793 \\
\hline
\end{tabular}
\begin{tabular}{|c|c|c|c|c|c|}
\hline Phrase & Address & Phrase & Address & Phrase & Address \\
\hline F & 299F & FINISH & 2B5B & FORTY & 2C3E \\
\hline FIFTEEN & 29 C 2 & FINISHED & 2B94 & FOUR & 14E7 \\
\hline FIFTY & 2A1D & FIRST & 2BD7 & FOURTEEN & 2C7F \\
\hline FIGURE & 2A60 & FIT & 2C14 & FOURTH & 2D19 \\
\hline FIND & 2AD7 & FIVE & 1531 & FROM & 2D74 \\
\hline FINE & 2B1E & FOR & 14E7 & FRONT & 2DBC \\
\hline G & 2DEB & GO & 2FFC & GOODBYE & 3148 \\
\hline GAMES & 2E28 & GOES & 3031 & GOT & 31A0 \\
\hline GET & 2E8C & GOING & 3079 & GRAY & 31D1 \\
\hline GETTING & 2EBA & GOOD & 30D6 & GREEN & 321D \\
\hline GIVE & 2F38 & GOOD WORK & 30FA & GUESS & 327E \\
\hline GIVES & 2F8D & & & & \\
\hline H & 32C0 & HEAD & 348C & HIT & 360A \\
\hline HAD & 32EF & HEAR & 34E5 & HOME & 363E \\
\hline HAND & 3339 & HELLO & 351A & HOW & 3689 \\
\hline HANDHELD UNIT & 337F & HELP & 3571 & HUNDRED & 36EF \\
\hline HAS & 3405 & HERE & 34E5 & HURRY & 3757 \\
\hline HAVE & 344A & HIGHER & 35AE & & \\
\hline I & 3793 & INCH & 38B5 & INSTRUCTIONS & 39BD \\
\hline I WIN & 37CF & INCHES & 38FA & IS & 3A32 \\
\hline IF & 3850 & INSTRUCTION & 394B & IT & 3A7A \\
\hline IN & 3872 & & & & \\
\hline J & 3AAE & JOYSTICK & 3AED & JUST & 3B4C \\
\hline K & 3B8A & KEYBOARD & 3BE9 & KNOW & 3C4F \\
\hline KEY & 3BB9 & & & & \\
\hline L & 3C8F & LEFT & 3E78 & LOAD & 404B \\
\hline LARGE & 3CD0 & LESS & 3EB2 & LONG & 40D3 \\
\hline LARGER & 3D19 & LET & 3F08 & LOOK & 413D \\
\hline LARGEST & 3 D 67 & LIKE & 3F2F & LOOKS & 4191 \\
\hline LAST & 3DDE & LIKES & 3F6A & LOWER & 41 E 7 \\
\hline
\end{tabular}
\begin{tabular}{|c|c|c|c|c|c|}
\hline Phrase & Address & Phrase & Address & Phrase & Address \\
\hline LEARN & 3E1E & LINE & 3FD5 & & \\
\hline M & 4233 & MEMORY & 4405 & MODULE & 45DF \\
\hline MADE & 4267 & MESSAGE & 446C & MORE & 4642 \\
\hline MAGENTA & 42AE & MESSAGES & 44D7 & MOST & 4693 \\
\hline MAKE & 432E & MIDDLE & 4551 & MOVE & 46DF \\
\hline ME & 437D & MIGHT & 4593 & MUST & 473D \\
\hline MEAN & 43BD & & & & \\
\hline N & 4789 & NEXT & 4959 & NO & 3C4F \\
\hline NAME & 47C0 & NICE TRY & 49A5 & NOT & 4AAB \\
\hline NEAR & 4833 & NINE & 1664 & NOW & 4ADA \\
\hline NEED & 4880 & NINETY & 4A4E & NUMBER & 4B20 \\
\hline NEGATIVE & 48DC & & & & \\
\hline O & 4B7D & ON & \(4 \mathrm{C41}\) & ORDER & 4D34 \\
\hline OF & 4BBA & ONE & 1409 & OTHER & 4D8A \\
\hline OFF & 4C13 & ONLY & 4C8B & OUT & 4DD4 \\
\hline OH & 4B7D & OR & 4CDC & OVER & 4E0A \\
\hline P & 4E66 & PLEASE & 5093 & PRINTER & 52AA \\
\hline PART & 4E9F & POINT & 50EC & PROBLEM & 52F9 \\
\hline PARTNER & 4EE0 & POSITION & 5148 & PROBLEMS & 5360 \\
\hline PARTS & 4F31 & POSITIVE & 51B3 & PROGRAM & 53EE \\
\hline PERIOD & 4F81 & PRESS & 5231 & PUT & 5477 \\
\hline PLAY & 4FE5 & PRINT & 526D & PUTTING & 54AA \\
\hline PLAYS & 502D & & & & \\
\hline Q & 5520 & & & & \\
\hline R & 556E & RECORDER & 5745 & RETURN & 58CF \\
\hline RANDOMLY & 55A0 & RED & \(57 \mathrm{C1}\) & REWIND & 593A \\
\hline READ (read) & 5652 & REFER & 5801 & RIGHT & 7 C 38 \\
\hline READ1 (red) & 57C1 & REMEMBER & 5861 & ROUND & 59 C 2 \\
\hline READY TO START & 56B3 & & & & \\
\hline S & 5A5A & SHAPES & 5DDE & SOME & 6197 \\
\hline
\end{tabular}
\begin{tabular}{|c|c|c|c|c|c|}
\hline Phrase & Address & Phrase & Address & Phrase & Address \\
\hline SAID & 5AA1 & SHIFT & 5E27 & SORRY & 61C6 \\
\hline SAVE & 5AEF & SHORT & 5E5C & SPACE & 6226 \\
\hline SAY & 5B65 & SHORTER & 5EA5 & SPACES & 625D \\
\hline SAYS & 3BA2 & SHOULD & 5F24 & SPELL & 62CC \\
\hline SCREEN & 5BFB & SIDE & 5F6D & SQUARE & 6333 \\
\hline SECOND & 5C5B & SIDES & 5FC8 & START & 637C \\
\hline SEE & 1 C 86 & SIX & 15A8 & STEP & 63 C 5 \\
\hline SEES & 5CBF & SIXTY & 601A & STOP & 63 F 7 \\
\hline SET & 5D1B & SMALL & 6070 & SUM & 6197 \\
\hline SEVEN & 15E8 & SMALLER & 60AE & SUPPOSED & 6423 \\
\hline SEVENTY & 5D50 & SMALLEST & 60F1 & SUPPOSED TO & 6489 \\
\hline SHAPE & 5DA5 & SO & 6153 & SURE & 64F4 \\
\hline T & 6551 & THERE & 6A72 & TIME & 6E69 \\
\hline TAKE & 658B & THESE & 6ADE & TO & 145C \\
\hline TEEN & 65BF & THEY & 6B47 & TOGETHER & 6EB0 \\
\hline TELL & 6603 & THING & 6BA0 & TONE & 6F1F \\
\hline TEN & 664E & THINGS & 6C0F & TOO & 145C \\
\hline TEXAS INSTRUMENTS & 6696 & THINK & 6 C 73 & TOP & 6F8D \\
\hline THAN & 675B & THIRD & 6CBC & TRY & 6FBB \\
\hline THAT & 67B6 & THIRTEEN & 6D11 & TRY AGAIN & 700F \\
\hline THAT IS INCORRECT & 6816 & THIRTY & 6DA2 & TURN & 7092 \\
\hline THAT IS RIGHT & 68FE & THIS & 6DDE & TWELVE & 70CE \\
\hline THE (thee) & 6974 & THREE & 149A & TWENTY & 7119 \\
\hline THE1 (the) & 69B6 & THREW & 6E26 & TWO & 145C \\
\hline THEIR & 6A72 & THROUGH & 6E26 & TYPE & 7170 \\
\hline THEN & 69E1 & & & & \\
\hline U & 71BE & UNDERSTAND & 729D & UPPER & \(73 \mathrm{C3}\) \\
\hline UHOH & 71F4 & UNTIL & 732F & USE & 7403 \\
\hline UNDER & 7245 & UP & 739F & & \\
\hline V & 7449 & VARY & 7487 & VERY & 74DA \\
\hline W & 7520 & WERE & 775D & WILL & 7A11 \\
\hline WAIT & 759D & WHAT & 77BC & WITH & 7A6B \\
\hline WANT & 75DF & WHAT WAS THAT & 77E9 & WON & 1409 \\
\hline
\end{tabular}
\begin{tabular}{|c|c|c|c|c|c|}
\hline Phrase & Address & Phrase & Address & Phrase & Address \\
\hline WANTS & 7621 & WHEN & 7875 & WORD & 7AAB \\
\hline WAY & 76B0 & WHERE & 78AB & WORDS & 7B0A \\
\hline WE & 767D & WHICH & 78F4 & WORK & 7B75 \\
\hline WEIGH & 76B0 & WHITE & 7924 & WORKING & 7BBC \\
\hline weight & 759D & wНо & 7969 & WRITE & 7 C 38 \\
\hline WELL & 7717 & WHY & 79B4 & & \\
\hline X & 7C8D & & & & \\
\hline Y & 7CB2 & YET & 7D99 & YOU WIN & 7DDB \\
\hline YELLOW & 7 CF 8 & YOU & 71BE & YOUR & 7E4D \\
\hline YES & 7D58 & & & & \\
\hline Z & 7E99 & ZERO & 13 C 3 & & \\
\hline
\end{tabular}

\subsection*{16.4 Streaming Raw Speech Data}

You can stream raw speech data to the Speech Synthesizer with the following word:
```

STREAM (addr n --- )

```

STREAM needs on the stack the address \(a d d r\) of a block of raw speech data to be spoken and the number of cells \(n\) in the buffer. This can be accomplished with DATA [ ... ]DATA . STREAM will feed the raw speech data to the Speech Synthesizer.

You should use TALKING? (see § 16.1 "Testing the State of the Speech Synthesizer" above) in word definitions to insure the Speech Synthesizer is not busy.

\section*{17 Sound Words}

Words introduced in this chapter:
PLAY PLAYING? SOUND
The words PLAY, SOUND and PLAYING? in this chapter were ported from TurboForth code courtesy of Mark Wills.
The TI-99/4A uses the TMS9919 Sound Generator Controller to generate sound. There are three tone generators and one noise generator available. For more detailed information about generating sound with the TMS9919, consult § 20 "Sound" in the Editor/Assembler Manual.

\subsection*{17.1 Generating Individual Sounds}

The three tone generators and the noise generator may be managed directly by the programmer with the SOUND word:

\section*{SOUND ( pitch vol ch\# --- )}

Pitch pitch, volume vol and channel ch\# are as described in the Editor/Assembler Manual in \(\S 20\). Pitch values range from \(0-1023,0\) representing the highest pitch. Volume values range from \(0-15,15\) representing silence. Channels \(0-2\) represent the corresponding tone generators and channel 3 is the noise generator.
SOUND uses the pitch value for setting the type of noise for the noise generator (channel 3). Shift rates are \(0-3\). Noise type can be white noise (0) or periodic noise (4). The pitch value to pass to SOUND is the sum of shift rate and noise type and ranges from \(0-7\).
Once a tone or noise generator is started, the sound/noise continues until silenced by executing SOUND with a volume of 15 . The pitch must be supplied, but is irrelevant. The following Forth code will silence channel 2 :
```

0 15 2 SOUND ok:0

```

\subsection*{17.2 Playing Sound Lists}

Playing sound lists involves setting up a sound table with one or more contiguous sound lists and providing a flag and the address of the sound table to PLAY (see description below).
fbForth \(\mathbf{2 . 0}\) does not use the console's keyscan routine for the interrupt-driven playing of sound lists as described in \(\S 20\) of the Editor/Assembler Manual, but rather processes them in its own ISR (see Chapter 10 "Interrupt Service Routines (ISRs)"). A sound table is, however, set up in the same manner as described in the Editor/Assembler Manual.

A second, presumably shorter, sound table can also be played while muting the first until the second table is finished. This should make it easy to periodically interrupt a game theme with short event-driven sounds such as crashes, beeps and warnings.

Each sound list consists of a list of sound commands starting with a byte count and ending with a duration count byte (sixtieths of a second) that is not included in the byte count. The last sound
list should silence all four sound generators (or, at least, the ones you used) and end with a duration of 0 . After setting up a sound table, it may be played with PLAY and monitored with PLAYING?:
PLAY ( addr flag --- )
Starts playing sound lists in the sound table set up at address addr depending on flag and continues until a sound list begins with a count of 0 or ends with a duration of 0 . The value of flag can be positive, negative or zero with the following effect:

\section*{Flag Action}

0 Do not play if either sound table is active.
1 Unconditionally play, killing all previous sound tables.
-1 Plays as sound table \#2, muting sound table \#1 for the duration of sound table \#2.

\section*{PLAYING? ( --- flag)}

PLAYING? checks both fbForth 2.0 sound status registers, ORs them and leaves that value on the stack as flag. If flag \(=0\), no sound table is active.
It should be noted that PLAYING? doe not work for sounds initiated with SOUND because SOUND communicates with the TMS9919 directly, bypassing the fbForth \(\mathbf{2 . 0}\) sound status registers.
A sound table may be prepared for PLAY with DATA [ ... ]DATA by dropping the cell count. Here is such a sound table set up as the word CHIME, which was taken from the chime sound example of § 20.4.2 of the Editor/Assembler Manual. For convenience, every other sound list in the CHIME sound table is shaded:
```

HEX
: CHIME ( -- addr )
DATA[
059F BFDF FFE3 0109 8E01 A402 C501 90B6 D306 0391 B7D4 0503
92B8 D504 05A7 0493 B0D6 0503 94B1 D706 0395 B2D8 0705 CA02
96B3 D006 0397 B4D1 0503 98B5 D204 0585 0390 B6D3 0503 91B7
D406 0392 B8D5 0705 A402 93B0 D606 0394 B1D7 0503 95B2 D804
05C5 0196 B3D0 0503 97B4 D106 0398 B5D2 0703 9FBF DF00
]DATA DROP ;

```

The CHIME sound table may now be played unconditionally with
CHIME 1 PLAY ok:0

\section*{18 Signed Integer Division}

When performing integer division, we usually think no further than the fact that there is a quotient and a remainder. This is often all there is. This is the case when both dividend and divisor have the same sign. But, what do we do when the signs differ? In what direction do we round the quotient? Toward zero (truncation)? Toward positive infinity (ceiling)? Negative infinity (floor)? What should be the sign of the remainder? The same as the dividend? The divisor? Always positive? Always negative? Whatever we choose must satisfy the equation,
\[
n=q d+r
\]
where \(n=\) numerator (dividend), \(q=\) quotient, \(d=\) denominator (divisor) and \(r=\) remainder.
There are good reasons for choosing one of several of these options, but we are only considering two, (1) so-called "symmetric integer division", which involves rounding toward zero or truncation and (2) "floored integer division", which involves rounding toward negative infinity. Probably the only reason (and, probably, not a good reason!) to choose signed integer division is that most of us expect the answers it gives and signed integer division is what many CPUs employ, but there are very sound reasons for choosing floored integer division, such as symmetry around zero. This is important in calculating position for electromechanical devices such as robots so that motions will be as nearly continuous as possible. It was for this reason that floored integer division was adopted for Forth-83. Because of the turmoil and consternation this caused many Forth programmers, Forth standards since then have made it optional by providing a way to do either signed integer division or floored integer division.

\subsection*{18.1 M/}

Signed integer division is the default for fbForth 2.0, but can easily be changed to floored integer division by changing the user variable \(\mathbf{S | F}\).
S|F ( --- addr ) (read "S or F" for "Symmetric or Floored")
User variable that determines whether M/ uses signed integer division or floored integer division. A value of zero (the default) specifies signed integer division and a non-zero value, floored integer division.

M/ acts like one of the ANS Forth words, SM/REM (symmetric M/ ) or FM/MOD (floored M/ ) depending on the value in S|F . M/ , SM/REM and FM/MOD all have the same stack signature:
( dn --- rem quot )

That is, they take a signed double number ( 32 bits) \(d\) as the dividend and a signed single number (16 bits) \(n\) as the divisor, leaving a signed single number remainder rem and a signed single number quotient quot. All of these words perform signed integer division. Where they differ is whether the division is symmetric or floored when the signs of divisor and dividend differ.

Currently, M/ uses signed integer division since fbForth is based on fig-Forth, which uses signed integer division. It will continue to be the default to support expectations of TI Forth programmers. However, \(\mathbf{S} \mid \mathbf{F}\) will make it easy for the user to change the behavior of \(\mathbf{M}\) / at will to accommodate floored integer division. Doing so will change all of the following words to use floored integer division because they are based on M/:
```

/MOD / MOD */MOD */

```

It should be noted that TurboForth uses floored integer division by default because it complies with the Forth-83 Standard, which, as noted in the Dr. Dobb's Journal article in § 18.3, was the first standard to make that move. The author would actually prefer to make floored integer division the fbForth default, but has chosen not to do so for the reasons in the last paragraph.

\subsection*{18.2 SM/REM and FM/MOD}

SM/REM performs symmetric integer division on a signed 32 -bit numerator by a signed 16 -bit denominator yielding a signed 16 -bit remainder and a signed 16 -bit quotient. The quotient is rounded toward zero, i.e., truncated, while the remainder takes the sign of the dividend. The code for SM/REM in fbForth \(\mathbf{2 . 0}\) is written in ALC, so here it is were it written in high-level Forth:
\begin{tabular}{cl} 
Word Definition & Comment \\
\hline : SM/REM & (dn --- rem quot ) <---stack signature \\
OVER >R >R & copy MSB of numerator to and move denominator to return stack \\
DABS R ABS & make numerator and denominator both positive \\
\(\mathbf{U /}\) & divide to get remainder and quotient \\
R> R XOR & get sign of quotient by XOR of numerator \& MSB of denominator \\
+- SWAP & give quotient proper sign \\
R> +- SWAP & give any remainder sign of numerator
\end{tabular}

FM/MOD performs floored integer division of a signed 32 -bit numerator by a signed 16 -bit denominator, yielding a signed 16 -bit remainder and a signed 16 -bit quotient. The quotient is rounded toward negative infinity, i.e., floored, while the remainder takes the sign of the divisor. The code for FM/MOD in fbForth 2.0 is written in ALC, so here it is were it written in high-level Forth:
```

Word Definition
: FM/MOD
OVER OVER XOR 0<
IF >R R SM/REM OVER

```

\section*{IF}
```

1- SWAP R> + SWAP
ELSE
R> DROP
THEN
ELSE
SM/REM
THEN
;

```

The code for FM/MOD is written in terms of SM/REM and is very similar to the Forth code for /MOD at the end of the article in the next section.

\subsection*{18.3 Dr. Dobb's Journal Article: "Signed Integer Division"}

This article originally appeared in Dr. Dobb's Journal, September, 1983. Permission to reproduce the full article has been sought.

\section*{Signed Integer Division}
by
Robert L. Smith

Not all methods of integer division produce a uniform result when the dividend and divisor have opposite signs. This may not be so important in the area of commerce where negative values are perhaps used less. When dealing with measurement and control, however, uniformity becomes more significant. The Forth-83 Standard adopts a method for signed integer division called "floored" division. While APL has used this method for the RES function for years and SmallTalk provides it as one of three methods for integer division, acceptance of the [Forth-83] Standard makes Forth the first "popular" language to embrace this method based on its theoretical merits. The problem is one of mathematical purity versus user expectation. This article will attempt to clarify some of the issues involved.

Integer division is a mathematical function of two integers (a dividend and a divisor) that yields an integer quotient and an integer remainder. That appears to be a fairly straightforward operation, but there is not universal agreement of the desired results when one or both arguments are negative. When an integer quotient is used in plotting or machine control, the desired function is usually not the quotient given by the majority of computers.

Most computers with a divide function produce a quotient that has a property of symmetry around zero when plotted as a function of the dividend, due to the fact that the quotient is rounded toward zero. Speaking mathematically, the property is actually one of anti-symmetry, where the sign of the quotient is reversed when the sign of the dividend (or numerator) is reversed. For integer division, this "symmetric" property leads to a sort of discontinuity around zero. In this case, the remainder is either zero or it takes the sign of the dividend. Figure 1(a) illustrates the quotient \(q\) as a function of a variable dividend, and a constant divisor 3.


We readily see the discontinuity near zero. This may be reasonably serious when this quotient function is used for plotting or moving robot arms. The integer quotient needs an associated remainder:
\[
r=n-q * d
\]
where \(n\) is the numerator or dividend, \(d\) is the denominator or divisor, \(q\) is the quotient, and \(r\) is the remainder. The remainder function for the constant divisor 3 is illustrated in Figure 1(b). If we look at the case of positive dividends and divisors, we observe the cyclic property that
\[
r(n+d)=r(n)
\]

In other words, the remainder usually has a repeating or cyclical property as the dividend changes. For the remainder shown in Figure 1(b), we see that this simple property is not maintained for dividends between \(-d\) and 0 .

If we require that the remainder be cyclical, then the quotient no longer has any unusual discontinuities. There are a number of possible choices here. One obvious choice is to make the remainder the same as the modulus or residue function [1]. In this case the quotient is rounded toward minus infinity. This rounding procedure is called the "floor" function. Figure 2 shows the floored quotient and its related modulus for the same arguments used in Figure 1. Notice the quotient behaves in a more nearly continuous fashion around zero. This is the form used in the Forth-83 Standard, as well as some of the older versions of Forth. The National 16032 microprocessor produces floored division in addition to the older "rounded toward zero" variety.

The modulus function is called MOD in Forth-83 and in the National 16032. It is called RES in APL.


The "floored" quotient shown in Figure 2 is not anti-symmetric around zero. However, for odd divisors one may easily obtain a symmetric result by adding a correction factor to the dividend prior to division. Although the quotient is generally not defined when the divisor is zero, the modulus is usually defined to take the value of the dividend for this case. If infinities are not allowed in computer representations, and the product of any number and zero is always zero, then this definition preserves the equation
\[
n=q * d+r
\]
for all values of \(d\), including zero.
Alternative remainder functions include a positive modulus and a remainder that takes the sign of the quotient [2]. Some other possibilities have the undesirable feature of negative remainders when the dividend and divisor are both positive.

Floored division is simply more useful in the majority of applications programs. The major objection is that the results are not what most people expect: -1 divided by 4 gives 0 in the rounded-toward-zero division case, but -1 for floored division. Both cases give the same results
when the dividend and divisor have the same sign. Timing efficiencies may play a small role in deciding which form of division to use, but generally the division process is sufficiently slow that additional tests for different forms of rounding take only a little extra time. Indeed, for some processors with built-in signed and unsigned divide functions, it may be faster in the common case of positive arguments to test signs and use the unsigned division than to just use the signed division function. If you have an older Forth system (such as Forth-79 Standard or fig-FORTH), the screen in Figure 3 shows a high-level conversion from the older form of /MOD to the newer version. For those unfamiliar with Forth, /MOD takes two arguments, the dividend and the divisor, and returns two results: the quotient and the modulus, or remainder. The quotient is returned as the most accessible element on the stack.

The appearance of floored division in some of the newer processor chips and languages indicates the increasing awareness of its utility. We might note in passing that even floating-point division will probably be different in the future than it was in the past due to the new Floating-Point Standard, which will require proper rounding of the quotient.
```

( Define 83-Standard /MOD in terms of old /MOD )
: /MOD ( num den -- mod quot )
OVER OVER XOR 0< ( test signs of arguments )
IF ( signs are different )
>R R@ /MOD OVER ( divide and examine remainder )
IF ( non-zero remainder )
1- SWAP R> + SWAP ( adjust results )
ELSE ( zero remainder )
R> DROP ( discard old den )
THEN
ELSE ( signs just the same )
/MOD ( just divide normally )
THEN ; ( end of definition )

```

Figure 3. High Level Forth Code to Convert to Floored Division

\section*{References}
1. Donald E. Knuth,The Art of Computer Programming: Volume I, Fundamental Algorithms, Second Edition, Menlo Park: Addison-Wesley, 1973, p. 127.
2. Robert Berkey, "Integer Division, Rounding and Remainders", 1982 FORML Conference Proceedings, San Jose, California: Forth Interest Group, 1983, pp. 13 23.

\section*{Appendix A ASCII Keycodes (Sequential Order)}
\begin{tabular}{|c|c|c|c|c|c|c|c|c|c|}
\hline \multicolumn{3}{|l|}{\multirow[b]{2}{*}{Character}} & \multicolumn{2}{|l|}{ASCII Code} & \multicolumn{3}{|c|}{\multirow[b]{2}{*}{Character}} & \multicolumn{2}{|l|}{ASCII Code} \\
\hline & & & hex & decimal & & & & hex & decimal \\
\hline NUL & <CTRL+,> & & 00h & 0 & SP & & & 20h & 32 \\
\hline SOH & <CTRL+A> & <FCTN+7> & 01h & 1 & ! & & & 21h & 33 \\
\hline STX & <CTRL+B> & <FCTN+4> & 02h & 2 & " & <FCTN+P> & & 22h & 34 \\
\hline ETX & <CTRL+C> & <FCTN+1> & 03h & 3 & \# & & & 23h & 35 \\
\hline EOT & <CTRL+D> & <FCTN+2> & 04h & 4 & \$ & & & 24h & 36 \\
\hline ENQ & <CTRL+E> & <FCTN+=> & 05h & 5 & \% & & & 25h & 37 \\
\hline ACK & <CTRL+F> & <FCTN+8> & 06h & 6 & \& & & & 26h & 38 \\
\hline BEL & <CTRL+G> & <FCTN+3> & 07h & 7 & & <FCTN+0> & & 27h & 39 \\
\hline BS & <CTRL+H> & <FCTN+S> & 08h & 8 & ( & & & 28h & 40 \\
\hline HT & <CTRL+1> & <FCTN+D> & 09h & 9 & ) & & & 29h & 41 \\
\hline LF & <CTRL+J> & <FCTN+ \(\times\) > & 0Ah & 10 & * & & & 2Ah & 42 \\
\hline VT & <CTRL+K> & <FCTN+E> & 0Bh & 11 & + & & & 2Bh & 43 \\
\hline FF & <CTRL+L> & <FCTN+6> & 0Ch & 12 & & & & 2Ch & 44 \\
\hline CR & <CTRL+M> & & 0Dh & 13 & - & & & 2Dh & 45 \\
\hline SO & <CTRL+N> & <FCTN+5> & 0Eh & 14 & & & & 2Eh & 46 \\
\hline SI & <CTRL+0> & <FCTN+9> & 0Fh & 15 & 1 & & & 2Fh & 47 \\
\hline DLE & <CTRL+P> & & 10h & 16 & 0 & <CTRL+0> & & 30h & 48 \\
\hline DC1 & <CTRL+Q> & & 11h & 17 & 1 & <CTRL+1> & & 31h & 49 \\
\hline DC2 & <CTRL+R> & & 12h & 18 & 2 & <CTRL+2> & & 32h & 50 \\
\hline DC3 & <CTRL+S> & & 13h & 19 & 3 & <CTRL+3> & & 33h & 51 \\
\hline DC4 & <CTRL+T> & & 14h & 20 & 4 & <CTRL+4> & & 34h & 52 \\
\hline NAK & <CTRL+U> & & 15h & 21 & 5 & <CTRL+5> & & 35h & 53 \\
\hline SYN & <CTRL+V> & & 16h & 22 & 6 & <CTRL+6> & & 36h & 54 \\
\hline ETB & <CTRL+W> & & 17h & 23 & 7 & <CTRL+7> & & 37h & 55 \\
\hline CAN & <CTRL+X> & & 18h & 24 & 8 & & & 38h & 56 \\
\hline EM & <CTRL+ >> & & 19h & 25 & 9 & <FCTN+Q> & <FCTN+,> & 39h & 57 \\
\hline SUB & <CTRL+Z> & & 1Ah & 26 & & <FCTN + /> & & 3Ah & 58 \\
\hline ESC & <CTRL+.> & & 1Bh & 27 & & <CTRL+/> & & 3Bh & 59 \\
\hline FS & <CTRL+;> & & 1Ch & 28 & < & <FCTN+0> & & 3Ch & 60 \\
\hline GS & <CTRL+=> & & 1Dh & 29 & \(=\) & <FCTN+;> & & 3Dh & 61 \\
\hline RS & <CTRL+8> & & 1Eh & 30 & \(>\) & <FCTN + B> & & 3Eh & 62 \\
\hline US & <CTRL+9> & & 1Fh & 31 & ? & < \(F\) CTN + H> & <FCTN+1> & 3Fh & 63 \\
\hline
\end{tabular}
...continued from previous page-
\begin{tabular}{|c|c|c|c|c|c|c|c|}
\hline \multicolumn{2}{|l|}{\multirow[b]{2}{*}{Character}} & \multicolumn{2}{|l|}{\multirow[t]{2}{*}{ASCII Code}} & \multicolumn{2}{|l|}{\multirow[b]{2}{*}{Character}} & \multicolumn{2}{|l|}{ASCII Code} \\
\hline & & & & & & hex & decimal \\
\hline @ & <FCTN+ \({ }^{\text {> }}\) & 40h & 64 & & < \(F C T N+C>\) & 60h & 96 \\
\hline A & <FCTN+K> & 41h & 65 & a & & 61h & 97 \\
\hline B & <FCTN+L> & 42h & 66 & b & & 62h & 98 \\
\hline C & <FCTN+M> & 43h & 67 & c & & 63h & 99 \\
\hline D & <FCTN+N> & 44h & 68 & d & & 64h & 100 \\
\hline E & & 45h & 69 & e & & 65h & 101 \\
\hline F & \(<F C T N+\gamma>\) & 46h & 70 & f & & 66h & 102 \\
\hline G & & 47h & 71 & g & & 67h & 103 \\
\hline H & & 48h & 72 & h & & 68h & 104 \\
\hline I & & 49h & 73 & i & & 69h & 105 \\
\hline J & & 4Ah & 74 & j & & 6Ah & 106 \\
\hline K & & 4Bh & 75 & k & & 6Bh & 107 \\
\hline L & & 4Ch & 76 & 1 & & 6Ch & 108 \\
\hline M & & 4Dh & 77 & m & & 6Dh & 109 \\
\hline N & & 4Eh & 78 & n & & 6Eh & 110 \\
\hline O & & 4Fh & 79 & o & & 6Fh & 111 \\
\hline P & & 50h & 80 & p & & 70h & 112 \\
\hline Q & & 51h & 81 & q & & 71h & 113 \\
\hline R & & 52h & 82 & r & & 72h & 114 \\
\hline S & & 53h & 83 & s & & 73h & 115 \\
\hline T & & 54h & 84 & t & & 74h & 116 \\
\hline U & & 55h & 85 & u & & 75h & 117 \\
\hline V & & 56h & 86 & v & & 76h & 118 \\
\hline W & & 57h & 87 & w & & 77h & 119 \\
\hline X & & 58h & 88 & x & & 78h & 120 \\
\hline Y & & 59h & 89 & y & & 79h & 121 \\
\hline Z & & 5Ah & 90 & z & & 7Ah & 122 \\
\hline [ & <FCTN+R> & 5Bh & 91 & \{ & <FCTN+F> & 7Bh & 123 \\
\hline \} & <FCTN+Z> & 5Ch & 92 & | & <FCTN+A> & 7Ch & 124 \\
\hline ] & <FCTN+>> & 5Dh & 93 & \} & <FCTN+G> & 7Dh & 125 \\
\hline \(\wedge\) & & 5Eh & 94 & \(\sim\) & <FCTN+W> & 7Eh & 126 \\
\hline & <FCTN+U> & 5Fh & 95 & DEL & \(<\) FCTN +1\(\rangle\) & 7Fh & 127 \\
\hline
\end{tabular}

\section*{Appendix B ASCII Keycodes (Keyboard Order)}
\begin{tabular}{|c|c|c|c|c|c|}
\hline \multirow[b]{2}{*}{Control Key} & \multicolumn{2}{|c|}{ASCII Code} & \multirow[b]{2}{*}{Function Key} & \multicolumn{2}{|r|}{ASCII Code} \\
\hline & hex & decimal & & hex & decimal \\
\hline <CTRL+1> & 31h & 49 & <FCTN+1> & 03h & 3 \\
\hline <CTRL+2> & 32h & 50 & <FCTN+2> & 04h & 4 \\
\hline <CTRL+3> & 33h & 51 & <FCTN+3> & 07h & 7 \\
\hline <CTRL+4> & 34h & 52 & <FCTN+4> & 02h & 2 \\
\hline <CTRL+5> & 35h & 53 & <FCTN+5> & 0Eh & 14 \\
\hline <CTRL+6> & 36h & 54 & <FCTN+6> & 0Ch & 12 \\
\hline <CTRL+7> & 37h & 55 & <FCTN+7> & 01h & 1 \\
\hline <CTRL+8> & 1Eh & 30 & <FCTN+8> & 06h & 6 \\
\hline <CTRL+9> & 1Fh & 31 & <FCTN+9> & 0Fh & 15 \\
\hline <CTRL+0> & 30h & 48 & <FCTN+0> & 3Ch & 60 \\
\hline <CTRL+=> & 1Dh & 29 & <FCTN+=> & 05h & 5 \\
\hline <CTRL+Q> & 11h & 11 & <FCTN+Q> & 39h & 57 \\
\hline <CTRL+W> & 17h & 23 & <FCTN+W> & 7Eh & 126 \\
\hline <CTRL+E> & 05h & 5 & <FCTN+E> & 0Bh & 11 \\
\hline <CTRL+R> & 12h & 18 & <FCTN+R> & 5Bh & 91 \\
\hline <CTRL+T> & 14h & 20 & <FCTN+ 7 > & 5Dh & 93 \\
\hline <CTRL+ \({ }^{\text {¢ }}\) > & 19h & 25 & <FCTN+ + > & 46h & 70 \\
\hline <CTRL+U> & 15h & 21 & <FCTN+U> & 5Fh & 95 \\
\hline <CTRL+|> & 09h & 9 & <FCTN+1> & 3Fh & 63 \\
\hline <CTRL+0> & 0Fh & 15 & <FCTN+0> & 27h & 39 \\
\hline <CTRL+P> & 10h & 16 & <FCTN+P> & 22h & 34 \\
\hline <CTRL+/> & 3Bh & 59 & <FCTN + /> & 3Ah & 58 \\
\hline
\end{tabular}
...continued from previous page-
\begin{tabular}{|c|c|c|c|c|c|}
\hline \multirow[b]{2}{*}{Control Key} & \multicolumn{2}{|c|}{ASCII Code} & \multirow[b]{2}{*}{Function Key} & \multicolumn{2}{|r|}{ASCII Code} \\
\hline & hex & decimal & & hex & decimal \\
\hline <CTRL+A> & 01h & 1 & <FCTN+A> & 7Ch & 124 \\
\hline <CTRL+S> & 13h & 19 & <FCTN+S> & 08h & 8 \\
\hline <CTRL+D> & 04h & 4 & <FCTN+D> & 09h & 9 \\
\hline <CTRL+F> & 06h & 6 & <FCTN+F> & 7Bh & 123 \\
\hline <CTRL+G> & 07h & 7 & <FCTN+G> & 7 Dh & 125 \\
\hline <CTRL+H> & 08h & 8 & <FCTN+H> & 3Fh & 63 \\
\hline <CTRL+J> & 0Ah & 10 & <FCTN+>> & 40h & 64 \\
\hline <CTRL+K> & 0Bh & 11 & <FCTN+K> & 41h & 65 \\
\hline <CTRL+L> & 0Ch & 12 & <FCTN+L> & 42h & 66 \\
\hline <CTRL;;> & 1-h & 28 & <FCTN+;> & 3Dh & 61 \\
\hline <CTRL+Z> & 1Ah & 26 & <FCTN+Z> & 5Ch & 92 \\
\hline <CTRL+X> & 18h & 24 & <FCTN + >> & 0Ah & 10 \\
\hline <CTRL+C> & 03h & 3 & <FCTN+C> & 60h & 96 \\
\hline <CTRL+V> & 16h & 22 & <FCTN+ \({ }^{\text {b }}\) > & 7Fh & 127 \\
\hline <CTRL+B> & 02h & 2 & <FCTN+B> & 3Eh & 62 \\
\hline <CTRL+N> & 0Eh & 14 & <FCTN+N> & 44h & 68 \\
\hline <CTRL+M> & 0Dh & 13 & <FCTN+M> & 43h & 67 \\
\hline <CTRL+,> & 00h & 0 & <FCTN + , > & 38h & 56 \\
\hline <CTRL+.> & 1Bh & 27 & <FCTN + .> & 39h & 57 \\
\hline
\end{tabular}

\title{
Appendix C How fbForth 2.0 differs from Starting FORTH (1st Ed.)
}
\begin{tabular}{|c|c|c|}
\hline Page & Word & Changes Required \\
\hline 10 & BACKSPACE & <FCTN + S> produces a backspace on the TI 99/4A. \\
\hline 10 & ok & fbForth 2.0 automatically prints a space before " ok: \(\boldsymbol{n}\) ". \\
\hline 16 & & The fbForth 2.0 dictionary can store names up to 31 characters in length. \\
\hline 18 & \(\wedge\) & Not a special character in fbForth 2.0. \\
\hline 18 & ." & Will execute inside or outside a colon definition in fbForth 2.0. \\
\hline 42 & /MOD & Uses signed numbers in fbForth 2.0. Remainder has sign of dividend. \\
\hline 42 & MOD & Uses signed numbers in fbForth 2.0. Remainder has sign of dividend. \\
\hline 50 & . S & The resident fbForth \(\mathbf{2 . 0}\) version prints a vertical bar ' \(\rho\) ' instead of ' 0 ' followed by the stack contents. The stack contents will be printed as unsigned numbers. The definition shown does not work in fbForth 2.0, even changing 'S to SP@ 2- 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. \\
\hline 52 & 2SWAP & This word is not in fbForth 2.0 but can be created with the following definition: \\
\hline & & : 2SWAP ROT >R ROT R> ; \\
\hline 52 & 2DUP & This word is not in fbForth 2.0 but can be created with the following definition: \\
\hline & & : 2DUP OVER OVER ; \\
\hline 52 & 20VER & This word is not in fbForth 2.0 but can be created with the following definition: \\
\hline & & 20VER SP@ 6 + @ SP@ 6 + @ ; \\
\hline 52 & 2DROP & This word is not in fbForth 2.0 but can be created with the following definition: \\
\hline & & : 2DROP DROP DROP ; \\
\hline 57 & & When you redefine a word that is already in the dictionary, fbForth 2.0 will issue a message saying " WORD isn't unique. ". In the example, a message saying " GREET isn't unique." would appear. \\
\hline 60 & & In fbForth 2.0, 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. \\
\hline
\end{tabular}
\begin{tabular}{|c|c|c|}
\hline Page & Word & Changes Required \\
\hline 63-82 & & The fbForth 2.0 Editor is different (much better) than the editor described in this section. Read the section of this fbForth \(\mathbf{2 . 0}\) Manual describing the Editor. \\
\hline 83 & DEPTH & DEPTH is defined in the resident fbForth 2.0 dictionary. \\
\hline 84 & COPY & fbForth 2.0 has CPYBLK for this purpose, q.v. \\
\hline 84-5 & & Ignore Editor words. \\
\hline 89ff & THEN & THEN is in the fbForth 2.0 vocabulary and is a synonym for the word ENDIF . Many people find ENDIF less confusing than THEN . \\
\hline 91 & NOT & This word is not in fbForth 2.0, but can be created with the following definition: \\
\hline & & : NOT 0= ; \\
\hline \multirow[t]{2}{*}{101} & \multirow[t]{2}{*}{? DUP} & This word is identical to -DUP in fbForth 2.0. Use the following definition if necessary: \\
\hline & & : ?DUP -DUP ; \\
\hline 101ff & ABORT" & As with the Forth-79 Standard, fibForth \(\mathbf{2 . 0}\) provides ABORT instead of ABORT" . \\
\hline 102 & ?STACK & In fbForth 2.0 this word automatically calls ABORT and prints the appropriate error message. \\
\hline \multirow[t]{2}{*}{107} & \multirow[t]{2}{*}{2*} & This word is not in fbForth 2.0, but can be created with the following definition: \\
\hline & & : 2* DUP + ; \\
\hline 107 & 2/ & This word is not in fbForth 2.0, but can be created with the following definition: \\
\hline & & : 2/ 1 SRA ; \\
\hline \multirow[t]{2}{*}{108} & \multirow[t]{2}{*}{NEGATE} & This word is not in fbForth 2.0, but can be created with the following definition: \\
\hline & & : NEGATE MINUS ; \\
\hline 110 & I & This word exists in fbForth 2.0 but also has a duplicate definition, R. I and \(\mathbf{R}\) are identical in function. They both get a copy of the return stack top. \\
\hline 110 & I' & This word is not in fbForth 2.0, but can be created with the following definition: (Note: \(\mathbf{R}\) is a synonym for \(\mathbf{I}\).) \\
\hline & & : I' R> R SWAP >R ; \\
\hline 112 & & If you will notice, there is a . (print) missing in the QUADRATIC definition. You must add a . after the last + to make QUADRATIC work correctly. \\
\hline 112 & & Ignore the last two paragraphs. They do not apply. \\
\hline
\end{tabular}

\section*{Page Word Changes Required}

131 Just a reminder! You must define 2DUP and 2DROP before the COMPOUND example may be used.

132 There is a mistake in the second definition of TABLE. It should look like this:
: TABLE CR 111 DO
111 DO I J * 5 U.R LOOP CR LOOP ;

DNEGATE This word is not in fbForth 2.0, but can be created with the following definition:

\section*{: DENEGATE DMINUS ; \\ ```
: DENEGATE DMINUS ;
```}

173 DMAX
UD.
D- \(\quad\) This word is not in fbForth 2.0, but can be created with the following definition:
: D- DMINUS D+ ;

This word is not in fbForth 2.0, but can be created with the following definition:
```

: DMAX 2OVER 20VER D- SWAP DROP 0<
IF 2SWAP ENDIF
2DROP ;

```
\begin{tabular}{|c|c|c|}
\hline Page & Word & Changes Required \\
\hline \multirow[t]{3}{*}{173} & \multirow[t]{3}{*}{DMIN} & This word is not in fbForth 2.0, but can be created with the following definition: \\
\hline & & : DMIN 2OVER 20VER 2SWAP D- SWAP DROP 0< IF 2SWAP ENDIF \\
\hline & & 2DROP ; \\
\hline \multirow[t]{2}{*}{173} & \multirow[t]{2}{*}{\(D=\)} & This word is not in fbForth 2.0, but can be created with the following definition: \\
\hline & & : D= D- 0= SWAP 0= AND ; \\
\hline \multirow[t]{2}{*}{173} & \multirow[t]{2}{*}{D0=} & This word is not in fbForth 2.0, but can be created with the following definition: \\
\hline & & : D0= 0. D= ; \\
\hline 173 & D< & This word is not in fbForth 2.0, but can be created with the following definition: \\
\hline & & : D< D- SWAP DROP 0<; \\
\hline \multirow[t]{8}{*}{173} & \multirow[t]{8}{*}{DU<} & This word is not in fbForth 2.0, but can be created with the following definition: \\
\hline & & : DU< ROT SWAP OVER OVER U< \\
\hline & & IF (determined less using high order halves) DROP DROP DROP DROP 1 \\
\hline & & ELSE (test if high halves equal)
\[
=
\] \\
\hline & & IF (equal so just test low halves) U< \\
\hline & & ELSE (test fails) DROP DROP 0 \\
\hline & & ENDIF \\
\hline & & ENDIF ; \\
\hline
\end{tabular}

174 M+ This word is not in fbForth 2.0, but can be created with the following definition:
: M+ 0 D+ ;
This word is different in fbForth 2.0 and can be changed with the following definition:

\section*{: M/ M/ SWAP DROP ;}

174 M*/ Not available in fbForth 2.0 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 fbForth 2.0.

\section*{Page Word Changes Required}

183ff Variables in fbForth 2.0 are required to be initialized at creation, thus the word VARIABLE takes the top item on the stack and places it into the variable as its initial value. For example, 12 VARIABLE DATE both creates the variable DATE and initializes it to 12. If desired, the advanced user can use the words <BUILDS and DOES> to create a new defining word, VARIABLE, which has exactly the behavior of VARIABLE as used in this section. The code to do this is:
: VARIABLE <BUILDS 0 , DOES> ;
193 2VARIABLE This word is not in fbForth 2.0, but can be created with the following definition:
```

: 2VARIABLE <BUILDS 0. , , DOES> ;

```

This definition does not require a number to be on the stack when it is executed.

193 2! This word is not in fbForth 2.0, but can be created with the following definition:

> : 2! >R R ! R> 2+ ! ;

193 2@ This word is not in fbForth 2.0, but can be created with the following definition:
```

: 2@ >R R 2+ @ R> @ ;

```

193 2CONSTANT This word is not in fbForth 2.0, but can be created with the following definition:
: 2CONSTANT <BUILDS , , DOES> 2@ ;
This definition does not require a number on the stack.
You must place a 0 on the stack before executing VARIABLE COUNTS 10 ALLOT . This, however, initializes only the first element of the array COUNTS to 0 . You must execute either the FILL or ERASE instruction at the bottom of the page to properly initialize the array.
204 DUMP fbForth 2.0 already has a dump instruction which must be loaded from the disk. Dumps are always printed in hexadecimal. See Appendix D for location of DUMP .

The CREATE word of fbForth \(\mathbf{2 . 0}\) behaves somewhat differently. Hackers should consult fig-Forth documentation.
216 EXECUTE Because this word operates a little differently in fbForth 2.0, it must be preceded by the word CFA. The example should read:

\section*{' GREET CFA EXECUTE}

The example illustrating indirect execution must be modified to work in fbForth 2.0:
' GREET CFA POINTER ! POINTER @ EXECUTE
\begin{tabular}{|c|c|c|}
\hline Page & Word & Changes Required \\
\hline 218 & ['] & In fbForth 2.0, this word is unnecessary as the word ' will take the following word of a definition when used in a definition. \\
\hline 219 & NUMBER & In fbForth 2.0, NUMBER is always able to convert double precision numbers. \\
\hline 219 & ' NUMBER & fbForth 2.0 does not use ' NUMBER to locate the NUMBER routine. \\
\hline 220 & & In fbForth 2.0, the name field is variable length and contains up to 31 characters. Also, the link field precedes the name field in fbForth 2.0. \\
\hline 225 & EXIT & \begin{tabular}{l}
This word is ; S in fbForth 2.0. ; \(\mathbf{S}\) is the word compiled by ; so to create EXIT we might use: \\
: EXIT [COMPILE] ;S ; IMMEDIATE
\end{tabular} \\
\hline 225 & I & In fbForth 2.0, the interpreter pointer is called IP, not I \\
\hline 232 & & See Chapter 1 in this fbForth 2.0 Instruction Manual for instructions for loading elective blocks. \\
\hline 232 & RELOAD & This instruction is not available in fbForth 2.0. \\
\hline 233 & H & This word is DP ( dictionary pointer ) in fibForth 2.0. \\
\hline 235 & 'S & In fbForth 2.0, SP@ is used instead of ' \(\mathbf{S}\) \\
\hline 240 & & See Appendix F in this fbForth 2.0 Instruction Manual for a complete list of user variables. \\
\hline 240 & > IN & This word is IN in fbForth 2.0. \\
\hline 245 & LOCATE & fbForth 2.0 does not support LOCATE \\
\hline 256 & COPY & In fbForth 2.0, use the word CPYBLK. CPYBLK is disk resident. See Appendix D for location and usage. \\
\hline 259 & ['] & Change the ['] to ' in the bottom example. In fbForth 2.0, ' will compile the address of the next word in the colon definition. \\
\hline 261 & >TYPE & Unnecessary in non-multiprogramming systems. Not present in fbForth 2.0. \\
\hline 265 & RND & fbForth 2.0 has two random number generators: RND and RNDW. See Appendix D for descriptions. See also definitions for SEED and RANDOMIZE . \\
\hline 266 & MOVE & \begin{tabular}{l}
In fbForth 2.0, MOVE moves \(u\) words in memory, not \(u\) bytes. Use CMOVE instead. If you must conform to Starting FORTH (1 \(1^{s t} E d\).), MOVE can be redefined: \\
: MOVE 2/ MOVE ;
\end{tabular} \\
\hline 266 & <CMOVE & Not present in fbForth 2.0. 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. \\
\hline
\end{tabular}
\begin{tabular}{|c|c|c|}
\hline Page & Word & Changes Required \\
\hline 270 & WORD & In fbForth 2.0, the word WORD does not leave an address on the stack. \\
\hline 270 & TEXT & \begin{tabular}{l}
This word's name conflicts with fbForth 2.0's Graphics mode word of the same name. The definition that follows will work, but has a different name to avoid conflict. It does not check for a 72 -character limit: \\
: TXT PAD 72 BLANKS PAD HERE - 1- \\
DUP ALLOT MINUS SWAP WORD ALLOT ; \\
If you want the count to also be stored at PAD, remove the 1- from the definition. See also, TOKEN and S"
\end{tabular} \\
\hline 277 & >BINARY & This is named (NUMBER) in fbForth 2.0. \\
\hline 277 & & \begin{tabular}{l}
Because WORD does not leave an address on the stack, it is necessary to redefine PLUS as follows: \\
: PLUS 32 WORD DROP NUMBER + ." = " . ;
\end{tabular} \\
\hline 279 & NUMBER & This definition of NUMBER is not compatible with fbForth 2.0. \\
\hline 281 & -TEXT & Not in fbForth 2.0. Use the definition on page 282. \\
\hline 292 & & fbForth 2.0 uses the word pair <BUILDS ... DOES> to define a new defining word. <BUILDS calls CREATE as part of its function. \\
\hline 297 & & To create a byte ARRAY in fbForth 2.0:
\[
\begin{aligned}
& : ~ A R R A Y ~<B U I L D S ~ O V E R ~, ~ * ~ A L L O T ~ \\
& \text { DOES> DUP @ ROT } *++2+;
\end{aligned}
\] \\
\hline 298 & & Just a reminder! Don't forget to define 2* before trying the example at the bottom of the page. Also, replace the word CREATE with <BUILDS . \\
\hline 301 & (DO) & This is the runtime behavior of \(\mathbf{D O}\) just as listed. \(\mathbf{2 > R}\) is not used, however. \\
\hline 301 & DO & The given definition of DO is not compatible with fbForth 2.0. fbForth 2.0's definition of DO is much more complex because of compile-time error checking. \\
\hline 303 & (LITERAL) & The fbForth 2.0 name for this word is LIT \\
\hline 306 & & fbForth 2.0 remains in compilation mode until a is typed. \\
\hline
\end{tabular}

\section*{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 \(\mathbf{2 . 0}\) 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 \(\mathbf{2 . 0}\) 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, the String Stack effects will be shown similarly as the return stack effects with "( SS: before --- after )". See Chapter 14 "The Stack-based String Library" for details.

A few words expect information from the input stream following the Forth word, which will be shown after stack effects with "( IS: input )".

\section*{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".
\begin{tabular}{ll} 
Term/Abbreviation & Meaning \\
\hline\(\$ C a d d r\) & string constant address \\
addr, \(a d d r_{1}, \ldots\) & memory address \\
\(b\) & byte \\
col & column position \\
\(\mathbf{c c c c}, \mathbf{n n n n}, \mathbf{x x x x}\) & string representations \\
\(c f a\) & code field address
\end{tabular}

ASCII Collating Sequence: ! " \# \$ \% \& ( ) * + , . / digits : ; < = > ? @ ALPHA [ \ ] ^ _` alpha \{ | \} ~
\begin{tabular}{|c|c|}
\hline Term/Abbreviation & Meaning \\
\hline char & ASCII character code \\
\hline count & count ( length ) \\
\hline d, \(d_{1}, d_{2}, \ldots\) & signed double-precision numbers (2 cells each) \\
\hline dotcol, dotcol \(_{1}\), dotcol \(_{2}, \ldots\) & dot column position \\
\hline dotrow, dotrow \({ }_{1}\), dotrow \({ }_{2}\), ... & dot row position \\
\hline flag & Boolean flag \\
\hline false & Boolean false flag (value \(=0\) ) \\
\hline \(f, f_{1}, f_{2}, \ldots\) & floating point numbers (4 cells each) \\
\hline lfa & link field address \\
\hline \(n, n_{1}, n_{2}, \ldots\) & signed single-precision numbers \\
\hline nfa & name field address \\
\hline \(p f a\) & parameter field address \\
\hline row & row position \\
\hline rem & remainder \\
\hline blk & block number \\
\hline spr & sprite number \\
\hline str & string address \\
\hline true & Boolean true flag (value \(\neq 0\) ) \\
\hline tol & tolerance limit \\
\hline \(u\) & unsigned single-precision number \\
\hline \(u d\) & unsigned double-precision number (2 cells) \\
\hline vaddr & VDP address \\
\hline
\end{tabular}

\section*{D. 2 Naming Conventions for Forth Words}

This section is an effort to aid you in navigating this glossary, as well as to assist you in contriving names for your own Forth words.

A few, core Forth words are very short and cryptic because they are used so often:
\begin{tabular}{cl} 
Word & Function \\
\hline\(:\) & Begin definition. \\
; & End definition. \\
@ & Fetch. \\
! & Store. \\
, & Compile. \\
' & Look up. \\
: & Print.
\end{tabular}

Words that begin or end with the above word symbols are usually expected to have a similar function. Here are a few examples:
\begin{tabular}{ll} 
Word & Function \\
\hline C@ & Character fetch. \\
C! & Character store. \\
C, & Character compile. \\
.BASE & Print value in BASE. \\
.S & Print stack. \\
D. & Double-number print. \\
ASM: & Assembly Language Code, begin definition. \\
; ASM & End definition, Assembly Language Code.
\end{tabular}
' ( ) ' and ' < > ' surround runtime versions of similarly named words. Here are a few examples:
\begin{tabular}{ll} 
High-level Word & Runtime Version \\
\hline DO & (DO) \\
LOOP & (LOOP) \\
+LOOP & (+LOOP) \\
-FIND & (FIND) \\
NUMBER & (NUMBER) \\
USEBFL & (UB) \\
CLOAD & <CLOAD> \\
." & (.")
\end{tabular}
' > ', may also mean "greater?", "to" (sometimes preceded by ' - ' in words inherited from TI Forth) or "this". Here are some examples with implied locations in brackets:
\begin{tabular}{|c|c|}
\hline Word & Function \\
\hline >R & [Parameter stack] to return stack \\
\hline R> & Return stack to [parameter stack] \\
\hline F->S & Floating point (FP) number to parameter stack \\
\hline DOES> & Does this (high-level Forth code that follows this word) \\
\hline > & Greater? ( \(n_{1}>n_{2}\) ? \()\) \\
\hline >F & To FP number (converts FP number text in input stream and pushes it to the stack) \\
\hline >DEG & [Parameter stack value (radians)] to degrees \\
\hline
\end{tabular}

ASCII Collating Sequence: ! " \# \$ \% \& ( ) * + , - / digits : ; < = > ? @ ALPHA [ \ ] ^ _` alpha \{ | \}~
\begin{tabular}{ll} 
Word & Function \\
\hline ASM \(>\) CODE & Assembly language code to hexadecimal machine code \\
BASE->R & Value in BASE to return stack \\
R->BASE & Return stack value to BASE \\
F> & FP greater? \(\left(f_{1}>f_{2} ?\right)\) \\
F>R & FP number on parameter stack to return stack
\end{tabular}
' <', may also mean "less?" or "that". Here are some examples with implied locations in brackets:
\begin{tabular}{ll} 
Word & Function \\
\hline\(<\) & Less? \(\left(n_{1}<n_{2} ?\right)\) \\
<BUILDS & That (the word pointed to) builds new words \\
\(0<\) & 0 less? \((n<0 ?)\)
\end{tabular}

At the beginning of a word, ' ? ' usually means "query" and may or may not leave a flag on the parameter stack. Most of the words below are querying for error conditions and will abort with an error message when such an error condition exists. At the end of a word, you may think of it as making the word a question.
\begin{tabular}{ll} 
Word & Function \\
\hline ? & Query address on stack and print contents. \\
?COMP & \begin{tabular}{l} 
Query STATE for compilation. Abort with error message if not. \\
?CSP
\end{tabular} \\
\begin{tabular}{l} 
Query stack position for same level as start of definition. Abort \\
with error message if not.
\end{tabular} \\
?ERROR & \begin{tabular}{l} 
Query flag on stack and issue error number \(n\) if false.
\end{tabular} \\
?EXEC & \begin{tabular}{l} 
Query STATE for execution. Abort with error message if not.
\end{tabular} \\
?FLERR & \begin{tabular}{l} 
Query for FP calculation error. Abort with error message if so \\
?KEY
\end{tabular} \\
Query keyboard for any key. Leave 7-bit ASCII value of key or 0 \\
if none.
\end{tabular}

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

Word Function
?TERMINAL Query keyboard for break key. Leave true if so; false if not.
PLAYING? Are we playing a soundlist? Leave a flag on the stack to indicate whether a soundlist is active.
TALKING? Is the Speech Synthesizer talking? Leave a flag on the stack to indicate whether speech is active.

A very good reference for explaining and recommending naming conventions is THINKING FORTH: A Language and Philosophy for Solving Problems by Leo Brodie (1984, 1994, 2004), available free online from SourceForge at http://thinking-forth.sourceforge.net/. Of particular note are:
- Chapter 5 "Implementation: Elements of Forth Style", p.135-especially, the sections on
- "Choosing Names: The Art", p. 163, and
- "Naming Standards: The Science", p. 167
- Appendix E "Summary of Style Conventions", p. 283

\section*{D. 3 fbForth 2.0 Word Descriptions}
( \(n\) addr --- )
Stores 16 bit-number \(n\) at address. Pronounced "store".
! CSP
( --- )
Saves the stack position in user variable CSP . Used as part of compiler security.
\#
\(\left(d_{1}--d_{2}\right)\)
Converts the rightmost digit of a double number \(d_{1}\) to an ASCII character, which is placed in a pictured numeric output string built downward from PAD to HERE. The digit to convert is the remainder from division of \(d_{1}\) by the current radix contained in BASE . The quotient \(d_{2}\) is maintained for further processing. Used between <\# and \#>. See \#S , <\# and \#>. The details of pictured numeric output are shown at <\# .
\#>
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 \(<\#\).

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

Sets sprite numbers 0 to \(n-1\) in automotion.
( \(d_{1}--d_{2}\) )
Generates pictured numeric output as ASCII text at PAD from \(d_{1}\) by executing \# until a zero double number \(d_{2}\) results. Used between <\# and \#>, q.v. The details of pictured numeric output are shown at <\# .
( --- ) (SS: --- str ) (IS: string" ) "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,

\section*{\$" 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 (\$").
\$.
( \(n\)--- )
Display the top number on the stack as an unsigned hexadecimal number.
( --- ) (SS: --- )
The word \(\$ . \mathbf{S}\) 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

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

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 ) (IS: word)
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: comment) )
( 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)
( \(n\)--- )
The runtime procedure compiled by +LOOP, which adds \(n\) to the loop index and then tests for loop completion. See +LOOP .
( --- )
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:, DOES \(>\) CODE: and ;CODE (execution mode, for TI Forth compatibility), that rewrites the code field of the most recently defined word to point to the machine code sequence following DOES>ASM: , DOES \(>\) CODE: or ;CODE. See DOES>ASM:, DOES>CODE: and ;CODE for more details.

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

\section*{(ABORT)}

Resident
( --- )
Executes after an error when WARNING \(<0\). Normally, WARNING \(=1\). (ABORT) normally executes ABORT, but may be redirected (with care!) to execute a user's alternative procedure. It is defined as
: (ABORT) ABORT ;
If you wished to have (ABORT) execute your error procedure, say MY_ERROR_PROC , you would need to replace the \(c f a\) of ABORT in the definition of (ABORT) with the \(c f a\) of MY_ERROR_PROC . Fortunately, this is easy to do! The cfa of ABORT sits in the parameter field of (ABORT) , the address \(p f a\) of which is what ticking (ABORT) gives you. You can verify this with the following code:
```

HEX ok:0

```
' (ABORT) @ U. 6AAC ok:0
' ABORT CFA U. 6AAC ok:0
The second line above ticks (ABORT), fetches the resulting pfa's contents and prints what should be the \(c f a\) of ABORT. The third line above ticks ABORT, gets its \(c f a\) and prints it. As you can see, they are, indeed, the same address.
Now, to install your error procedure, simply get its \(c f a\) and stash it in the parameter field of (ABORT) as follows:
```

' MY_ERROR_PROC CFA ok:1
' (ABORT) ! ok:0

```

To get your error procedure to run at the next error, set WARNING to a negative number as below:

\section*{-1 WARNING ! ok:0}

To re-instate normal fbForth \(\mathbf{2 . 0}\) error handling, you only need to store a positive number in WARNING. You can restore the default action of (ABORT) with the following Forth code:
```

' ABORT CFA ok:1
' (ABORT) ! ok:0

```
( --- )
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

ASCII Collating Sequence: ! " \# \$ \% \& ( ) * + , . / digits : ; < = > ? @ HLPHA [ \] ^ _` alpha \{ | \}~
the text at \(a d d r\). The addresses, \(a d d r\) and \(n f a\), both point to the length byte of packed character strings (see footnote 4 on page 22). Returns the parameter field address \(p f a\), 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.]

Resident

\section*{( \(n\) blk --- addr count )}

Converts the line number \(n\) and the Forth block number blk to the disk buffer address \(a d d r\) 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.
( --- )
The runtime procedure compiled by LOOP , which increments the loop index and tests for loop completion. See LOOP .
(NUMBER)
Resident
( \(d_{1}\) addr \(r_{1}--d_{2} a d d r_{2}\) )
The double number \(d_{1}\) should be 0 , i.e., the stack should contain two 16 -bit zeroes. The address \(a d d r_{1}\) 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 \(a d d r_{1}+1\) with respect to the current radix in BASE. The new value is accumulated with double number \(d_{1}=0\) as the initial value. If a decimal point is encountered in the string, DPL is updated with the number of digits to the right of the decimal point. The address of the first unconvertible digit is \(a d d r_{2}\). (NUMBER) is used by NUMBER .
( --- )
The run time procedure compiled by \(\mathbf{0 F}\).
( addr --- )
Runtime routine compiled or executed by USEBFL that changes the current blocks file to the filename as a packed character string (see footnote 4 on page 22) pointed to by \(a d d r\).
*
\(\left(\begin{array}{lll}n_{1} & n_{2} & ---n_{3}\end{array}\right)\)
Leaves the signed product of two signed numbers.
( \(n_{1} n_{2} n_{3}--\) quot )
Leaves the quotient quot of \(\left(n_{1} * n_{2}\right) / n_{3}\), where all are signed numbers. Retention of an intermediate signed 32 -bit product permits greater accuracy than would be available with the sequence :

ASCII Collating Sequence: ! " \# \$ \% \& ( ) * + , . / digits : ; < = > ? @ ALPHA [ \ ] ^ _ alpha \{ 1 \}~
\[
n_{1} n_{2} * n_{3} /
\]
*/ is based on \(\mathbf{M} /\), which uses user variable \(\mathbf{S | F}\), q.v., to determine whether symmetric (the default) or floored division is used. See Chapter 18 "Signed Integer Division" for more details.

Resident
( \(n_{1} n_{2} n_{3}--\) rem quot )
Leaves the quotient quot and remainder rem of the operation \(\left(n_{1} * n_{2}\right) / n_{3}\). An intermediate signed 32 -bit product is used just as for \(* /\). In fact, \(* /\) MOD is used by */. */MOD is based on M/, which uses user variable S|F, q.v., to determine whether symmetric (the default) or floored division is used. See Chapter 18 "Signed Integer Division" for more details.
\(\left(n_{1} n_{2}---n_{3}\right)\)
Leaves the sum of \(n_{1}+n_{2}\) as \(n_{3}\).
( \(n\) addr --- )
Adds \(n\) to the value at the address. Pronounced "plus store".
( --- ) (SS: str \(\left.r_{1} s r_{2}-s t r_{1} \& ~ s t r_{2}\right) \quad\) "concatenate strings"
The word \(\boldsymbol{+}\) 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
\(\left(n_{1} n_{2}---n_{3}\right)\)
Apply the sign of \(n_{2}\) to \(n_{1}\), which is left as \(n_{3}\).
( addr \(_{1}---a d d r_{2}\) flag )
Advance the disk buffer address \(a d d r_{1}\) to the address of the next buffer \(a d d r_{2}\). Boolean flag is false when \(a d d r_{2}\) is the buffer presently pointed to by user variable PREV .

Used in a colon definition in the form:
```

DO ... n +LOOP

```

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

Compile time: (addr 3 --- )
+LOOP compiles the runtime word (+LOOP) and the branch offset computed from HERE to the address \(a d d r\) left on the stack by DO. The value 3 is used for compiletime 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.
( \(n\)--- )
Store \(n\) into the next available dictionary memory cell, advancing the dictionary pointer. Pronounced "comma".
\(\left(n_{1} n_{2}---n_{3}\right)\)
Leave the difference \(n_{3}\) of \(n_{1}-n_{2}\).
( --- )
Continues interpretation with the next Forth block in the current blocks file. --> can only be used while loading blocks. Pronounced "next block".
\(\left(n_{1}--n_{1} \mid n_{1} n_{1}\right)\)
Duplicate \(n_{1}\) only if it is non-zero. This is usually used to copy a value just before IF , to eliminate the need for an ELSE clause to drop a DUPed 0.
( --- false |pfa len true ) (IS: word)
Accepts the next text word (delimited by blanks) in the input stream to HERE as a packed character string (see footnote 4 on page 22), searches the CONTEXT and then CURRENT vocabularies for a matching entry. If found, the dictionary entry's parameter field address \(p f a\), its length byte len and true are left. Otherwise, only false is left. [Note: See Chapter 12 about the length byte.]
( \(n_{1} n_{2} n_{3}---n_{3} n_{1} n_{2}\) )
Rotate right the top three numbers on the stack, resulting in the top number on the bottom.
( --- ) ( SS: str str \(_{2}\) str \({ }_{3}--\) str \(_{3}\) str \(r_{1}\) str \(r_{2}\) ) "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. See Chapter 14 for implementation details regarding stack space limitations.

\section*{-TRAILING}

Resident
( addr \(n_{1}---\) addr \(n_{2}\) )
Adjusts the character count \(n_{1}\) of a character string at \(a d d r\) to suppress the output of trailing blanks by TYPE, i.e., the characters at \(a d d r+n_{2}\) to \(a d d r+n_{1}\) are blanks. If the character string is a packed character string (see footnote on page ), addr points to the first character after the length byte. -TRAILING 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 COUNT are suitable input parameters for -TRAILING .

Resident
( \(n\)--- )
Prints a number from a signed 16 -bit two's complement value \(n\), converted according to the numeric base stored in BASE . A trailing blank follows. Pronounced "dot".
[immediate word]
Resident
(---) (IS: string")
Used in the form:
```

." cccc"

```

Compiles an in-line string cccc (delimited by the trailing ") with an execution procedure to transmit the text to the selected output device. If executed outside a definition, " will immediately print the text until the final ". See (." ).

Stack-based String Library [42]
( ---) (SS: str --- ) "display string"
The word . \(\$\) pops the topmost string from the string stack and displays it. For example,

\section*{\$" Hello, World!" . \$ Hello, World! oK:0}
. \(\$\) CONST
Stack-based String Library [42]
( \(\$\) 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

```

ASCII Collating Sequence: ! " \# \$ \% \& ( ) * + , . / digits : ; < = > ? @ HLPHA [ \ ] ^ _` alpha \{ | \}~
( --- \(n\) )
Print the decimal value \(n\) of the current radix (number base).
( \(n\) blk --- )
Print on the terminal device a line of text from the current blocks file corresponding to the line number \(n\) of block number blk. Trailing blanks are suppressed.

Resident
( \(n_{1} n_{2}---\) )
Prints the number \(n_{1}\) right aligned in a field whose width is \(n_{2}\). No following blank is printed.
.S
( --- )
Prints the entire contents of the parameter stack as unsigned numbers in the current BASE. The bottom of the stack is shown by an initial ' \(\mid\) '.
( \(n_{1} n_{2}---n_{3}\) )
Leaves the quotient \(n_{3}\) of \(n_{1} / n_{2}\). / is based on \(\mathbf{M} /\), which uses user variable \(\mathbf{S} \mid \mathbf{F}\), \(q . v\), to determine whether symmetric (the default) or floored division is used. See Chapter 18 "Signed Integer Division" for more details.
( \(n_{1} n_{2}\)--- rem quot )
Leaves the remainder rem and signed quotient quot of \(n_{1} / n_{2}\). The remainder has the sign of the dividend. /MOD is based on M/, which uses user variable \(\mathbf{S} \mid \mathbf{F}\), q.v., to determine whether symmetric (the default) or floored division is used. See Chapter 18 "Signed Integer Division" for more details.
0123
Resident
(---n)
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.
\(0<\)
Resident
( \(n\)--- flag )
Leaves a true flag if the number \(n\) is less than zero (negative). Otherwise, \(0<\) leaves a false flag.
( \(n\)--- flag )
Leaves a true flag if the number is equal to zero. Otherwise, \(0=\) leaves a false flag.
0>
Resident
( \(n\)--- flag )
Leaves a true flag if the number is greater than zero (positive). Otherwise, \(0>\) leaves a false flag.
OBRANCH
(flag --- )
The runtime procedure to conditionally branch. If flag is false (zero), the following in-line parameter is added to the interpretive pointer to branch ahead or back. Compiled by IF , UNTIL, END and WHILE .
1+ Resident
( \(n_{1}---n_{2}\) )
Increments \(n_{1}\) by 1 .
1-
( \(n_{1}---n_{2}\) )
Decrements \(n_{1}\) by 1 .
2+
Resident
( \(n_{1}---n_{2}\) )
Leaves \(n_{1}\) incremented by 2 as \(n_{2}\).
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
(---) (IS: <new name> <Forth code> ; )
Used in the form, called a colon definition:
```

: cccc ... ;

```

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

Creates a dictionary entry defining \(\mathbf{C C C C}\) as equivalent to the sequence of Forth word definitions in '...' until the next ; , DOES>ASM: or DOES>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 FORGETting it.

ASM: <new word> <assembly mnemonics> ; ASM
; ASM puts 045Fh at HERE and advances HERE. This machine code for ALC, B *NEXT or B *R15, branches to the inner interpreter to fetch the next word to be executed. See Chapter 9 "The fbForth 2.0 TMS9900 Assembler" for more information. See also ASM: .
; CODE
[immediate word] Resident
( [ ] | C0DEh --- )
Only if compiling: ( IS: [ \(\langle a l c\rangle \mid\langle m c\rangle\), [ \(\langle m c\rangle\), ] ...] NEXT, ) to maintain compatibility with TI Forth.

ASCII Collating Sequence: ! " \# \$ \% \& ( ) * + , . / digits : ; < = > ? @ HLPHA [ \ ] ^ _` alpha \{ | \}~
;CODE is the terminator for CODE: and for DOES>CODE: . CODE: defines a new word cccc with machine code contents that do not use , to compile them. The following ALC quadruples the value on the stack by double addition:
```

ASM: QUAD ( n -- 4n )
*SP *SP A,
*SP *SP A,
; ASM

```

The above code can be re-stated in machine code without requiring the TMS9900 Assembler:

HEX CODE: QUAD A659 A659 ; CODE
With a very long definition, using CODE: cccc ... ; CODE is significantly faster and is much clearer to read regardless of the code's length.
The same situation obtains for DOES>CODE: , q.v. for more details. The following code
: cccc <BUILDS ... DOES>CODE: <MC only> ;CODE
is the machine-code equivalent of
: cccc <BUILDS ... DOES>ASM: <ALC only> ;ASM
( --- )
Stops interpretation of a Forth block. ; \(\mathbf{S}\) is also the runtime word compiled at the end of a colon definition, which returns execution to the calling procedure.
( \(n_{1} n_{2}--\) flag )
Leaves a true flag if \(n_{1}\) is less than \(n_{2}\). Otherwise, < leaves a false flag.
Resident
( --- )
Sets up for pictured numeric output formatting using the words, <\#, \#, HOLD, \#S , SIGN and \#> . <\# initializes HLD with PAD. HLD is decremented by \# via HOLD 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.

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

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:

\section*{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]
( \(n_{1} n_{2}---\) flag )
Result flag is true (1) if \(n_{1} \neq n_{2}\) and false (0) otherwise.

It is used within a colon-definition to build a new defining word:
```

cccc <BUILDS ... DOES> ... ; or
cccc <BUILDS ... DOES>ASM: ... ;ASM or
cccc <BUILDS ... DOES>CODE: ... ;CODE

```

Each time cccc is executed, <BUILDS defines a new word with a high-level (DOES> ) or machine-code (DOES>ASM: or DOES>CODE: ) 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 parameter field address \(p f a\) on the stack and executes the words after DOES> in cccc . For the definition with DOES>ASM: or DOES>CODE: , when nnnn is later executed, it only executes the words after DOES>ASM: or DOES>CODE: in CCCC, but without the pfa of nnnn on the stack. <BUILDS allows runtime procedures to be written in high-level code with DOES> , in assembler with DOES>ASM: or in machine code with DOES>CODE: .
<BUILDS is simply defined as
: <BUILDS CREATE SMUDGE ;
<CLOAD>
Resident
( --- )
The runtime procedure compiled by CLOAD .

\section*{=}
( \(n_{1} n_{2}--\) flag )
Leaves a true flag if \(n_{1}=n_{2}\). Otherwise, it leaves a false flag.
( \(a d d r_{1}--a d d r_{1} \mid a d d r_{2}\) )
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.
>
( \(n_{1} n_{2}\)--- flag )
Leaves a true flag if \(n_{1}>n_{2}\). Otherwise, it leaves a false flag.
( \$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
```

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

```
WELCOME >$ ok:0
```

```

Stack-based String Library [42]
( \(\$\) 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,
```

```
$ $CONST COLOR ok:0
```

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

```

Stack-based String Library [42]
string using . \$CONST as follows:

Resident
( \(f_{1}---f_{2}\) )
Converts an 8 -byte floating point number \(f_{1}\) from radians to \(f_{2}\) degrees.

\section*{[immediate word] \\ Resident}
( ---f) ( IS: <fp number string> )
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.

ASCII Collating Sequence: ! " \# \$ \% \& ( ) * + , - / digits : ; < = > ? @ HLPHA [ \ ] ^ _` alpha \{ | \}~
(bank addr ---)
This word is ported from TurboForth code courtesy of Mark Wills.
If a SAMS card is present, >MAP maps memory bank bank to address \(a d d r\).
Address \(a d d r\) should be a valid address on a 4 KiB boundary, viz., 2000h, 3000h, A000h, B000h, C000h, D000h, E000h or F000h. Bank bank should be a number between 0 and FFh.
S0\&TIB!, q.v., should be used to change S0 and TIB both to EFA0h or DFA0h (exactly 4 KiB or 8 KiB [1 or 2 SAMS page(s)] lower than the default FFA0h ), thus allowing the use of E000h and/or F000h with impunity!
When a SAMS memory expansion card is installed, the 32 KiB of CPU RAM is actually taken from the SAMS memory. At startup, fbForth 2.0 reserves the following banks of SAMS memory for the "standard" 32 KiB RAM:
\begin{tabular}{cc} 
Bank & \begin{tabular}{c}
4 KiB \\
Boundary
\end{tabular} \\
\hline F8h & 2000 h \\
F9h & 3000 h \\
FAh & A000h \\
FBh & B000h \\
FCh & C000h \\
FDh & D000h \\
FEh & E000h \\
FFh & F000h
\end{tabular}

As can be seen from the above table, fbForth assumes a 1024 KiB SAMS memory card; so, fbForth \(\mathbf{2 . 0}\) is not compatible with 256 KiB AMS cards.
Lower RAM 2000h - 3FFFh is reserved by fbForth \(\mathbf{2 . 0}\) for four block buffers, lowlevel support, system variables and the return stack; therefore, extreme care should be taken when paging banks F8h and F9h out of 2000h and 3000h, respectively. The same care should be taken with upper RAM when paging banks FAh and FFh out of A000h (start of User Dictionary) and F000h (TIB and base of parameter stack), respectively.
Because the RAM portion of the dictionary grows up from A030h and the parameter stack grows down from FFA0h, extreme care must be taken mapping SAMS memory if not using S0\&TIB!. It is probably advisable to limit SAMS mapping to one or two 4 KiB window(s) at D000h and/or E000h. If E000h is used, the stack is limited to 2000 cells, which is probably sufficient for most programming.

ASCII Collating Sequence: ! " \# \$ \% \& ( ) * + , . / digits : ; < = > ? @ ALPHA [ \ ] ^ _`alpha \{ | \}~
( \(n---\) ) ( R: --- \(n\) )
Removes a number from the parameter stack and place as the most accessible number on the return stack. Use should be balanced with \(\mathbf{R}>\) in the same definition.
\(>\) RAD
\(\left(f_{1}---f_{2}\right)\)
Converts an 8 -byte floating point number \(f_{1}\) from degrees to \(f_{2}\) radians.
?
( --- )
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 ?COMP 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 IN and BLK and executes QUIT , e.g.,

\section*{90 DO I . LOOP DO ? compilation only}

Though LOOP is also a compile-only word, DO is the first one encountered and the one that triggers the above error.
( --- )
This word is used in the definitions of ; , DOES>ASM: and DOES>CODE: to insure that the stack position at the end of the definition is at the same height as when it was started with : , which stores the stack pointer in CSP. 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.,

\section*{: XXXX IF ; i ? definition not finished}
(flag \(n\)--- )
Issues an error message corresponding to error number \(n\) if the Boolean flag is true. ? ERROR is the word that all the error-checking words in fbForth 2.0 execute to actually check for an error and to display the error message. It is defined as
: ?ERROR SWAP IF ERROR ELSE DROP THEN ;
?EXEC
Resident
( --- )
This word is used in the definitions of : , CODE , ASM: and most of the words in the ASSEMBLER vocabulary to insure those words are executing and not being used in a

ASCII Collating Sequence: ! " \# \$ \% \& ( ) * + , . / digits : ; < = > ? @ HLPHA [ \]^_`alpha \{ | \} ~
definition. ?EXEC issues the error message, "execution only", as in
: XXXX : ... ; \(\underset{\underline{?}}{\underline{?}}\) execution only
?FLERR
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 8354h. ?FLERR issues the error message, "floating point error", upon finding an error. The nature of the floating-point error may be ascertained by executing FLERR, q.v., to get the FP error number and crossreferencing the code in the error table in \(\S 7.14\) "Floating Point Error Codes".
?KEY
( --- char )
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.
?KEY8
(--- \(n\) )
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.
?LOADING
Resident
( --- )
This word is used in the definition of --> to insure that fbForth \(\mathbf{2 . 0}\) is loading from the current blocks file rather than executing on the command line. ?LOADING issues error message, "use only when loading", if not loading as in

\section*{--> --> ? use only when loading}
?PAIRS
( \(n_{1} n_{2}---\) )
Issue the error message, "conditionals not paired", if \(n_{1}\) does not equal \(n_{2}\). The message indicates that compiled conditionals do not match, such as when a DO has been left without a LOOP , an IF has no corresponding ENDIF or THEN , etc.
?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 ;

ASCII Collating Sequence: ! " \# \$ \% \& ( ) * + , . / digits : ; < = > ? @ HLPHA [ \]^_`alpha \{ | \}~
?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.
@
( \(a d d r---n\) )
Leave the 16-bit contents \(n\) of \(a d d r\).
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 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.
( \(n_{1}---n_{2}\) )
Leaves the absolute value of \(n_{1}\) as \(n_{2}\).
[immediate word]
Resident
Used in a colon definition in the form:

\section*{BEGIN ... AGAIN}

Compile time: ( \(a d d r 1---\) )
AGAIN compiles BRANCH with an offset from HERE to \(a d d r\), which it copies to the space reserved for it at \(a d d r\). 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.
ALIGN
Resident
( --- )
ALIGN insures that HERE is on an even address boundary. Use of \(\mathbf{C}\), is one way HERE can land on an odd address boundary. CREATE uses ALIGN before installing the header for a new word definition. Align is very similar to =CELLS except that it neither expects nor leaves anything on the stack.

ASCII Collating Sequence: ! " \# \$ \% \& ( ) * + , . / digits : ; < = > ? @ ALPHA [ \ ] ^ _`alpha \{ | \}~
( \(n\)--- )
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
\(\left(n_{1} n_{2}---n_{3}\right)\)
Leave the bitwise logical AND of \(n_{1}\) and \(n_{2}\) as \(n_{3}\).
APPND
( --- )
Assigns the APPEND attribute to the file whose PAB (Peripheral Access Block) is pointed to by PAB-ADDR .
( --- ascii) (IS:token)
Leaves on the stack the ASCII value of the first character of the next token in the input stream:
\[
\text { ASCII G . } \underline{71} \text { ok:0 }
\]

ASM:
Resident
( --- ) (IS: <new word> <alc> ; ASM)
ASM: opens an ASM: cccc ... ; ASM word definition that allows the programmer to write the body of the word in Assembly Language. To do so, requires the fbForth 2.0 TMS9900 Assembler first be loaded from FBLOCKS. Typing MENU will reveal the block number to load the Assembler. It is used as follows:

ASM: <new word> <assembly mnemonics> ; ASM
See Chapter 9 "The fbForth 2.0 TMS9900 Assembler" for more information. See also ;ASM .

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

\section*{ASM>CODE}

ASM>CODE -- Code Output Utility
( --- ) ( IS:word DSKn.file)
ASM>CODE appends to DSKn.file the hexadecimal machine code of a Forth word written in ALC (Assembly Language Code) in CODE: newword ... ; CODE format, where '...' represents the machine code in text. This is useful for loading words defined in ALC without the need for loading the fbForth TMS9900 Assembler from FBLOCKS. Please note that ASM>CODE should not be used for words in the resident dictionary because word entries in the resident dictionary are in an unconventional, non-contiguous format.
ASM>CODE first checks to insure that word is a word defined in ALC. If it is not or it does not exist, ASM>CODE quits with an error message to that effect. If it is an ALC word, ASM>CODE attempts to open the file DSKn.file in Append mode. Failing that, DSKn.file is created and opened in Output mode.
As an example you might assemble the word LDCR , the ALC for which is listed in Appendix H "Assembly Source for CODEd Words", and then run the following code:

\section*{ASM>CODE LDCR DSK1.CRUWORDS}

Examining the contents of DSK1.CRUWORDS would reveal the same code as shown in Block \#5 of FBLOCKS (17JUN2016 and later).
If you are using the TI-99/4A emulator, Classic99 (www.HarmlessLion.com), in Microsoft Windows, you can use the Windows clipboard as the file CLIP as follows:

\section*{ASM>CODE LDCR CLIP}

See Chapter 9 "The fbForth 2.0 TMS9900 Assembler" in the manual for additional information.

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 .
( \(f_{1}---f_{2}\) )
Calculates the arctangent in radians of \(f_{1}\) leaving the floating point result \(f_{2}\) on the stack.

B/BUF
Resident
( --- 1024 )
This constant leaves the number of bytes \(n\) per disk buffer (always 1024 in fbForth 2.0), the byte count read from the current blocks file by BLOCK . It is included for backward compatibility with TI Forth

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

BACK
( addr --- )
Calculates the backward branch offset from HERE to \(a d d r\) and compile into the next available dictionary memory address. Used by LOOP , +LOOP , UNTIL and AGAIN to calculate the distance back to the beginning of the loop.
( addr \(n_{1}---n_{2}\) )
Returns on the stack the contents \(n_{2}\) of the cell at address \(a d d r\) 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.
BANKC@
Resident
( \(a d d r n---b\) )
Returns on the stack the contents \(b\) of the byte at address addr in bank \(n\). See BANK@ .
BASE
( --- addr )
A user variable containing the current radix or number base used for input and output conversion.
BASE->R
Resident
( --- )
Places the current radix on the return stack. Caution must be exercised when using BASE->R and R->BASE with CLOAD as these will cause the return stack to be polluted if a LOAD is aborted and the BASE->R is not balanced by a \(\mathbf{R}->\) BASE at execution time. See R->BASE .
BEEP
( --- )
Produces the sound associated with correct input or prompting.
BEGIN
[immediate word]
Resident

Occurs in a colon-definition in the form:
```

BEGIN ... UNTIL or BEGIN ... END
BEGIN ... AGAIN
BEGIN ... WHILE ... REPEAT

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

Compile time: ( --- addr 1 )
BEGIN leaves its return address addr for branching calculation and storage by UNTIL, END , AGAIN and REPEAT and a 1 for compiler error checking.
Runtime: ( --- )
BEGIN marks the start of a sequence that may be repetitively executed. It serves as a return point from the corresponding UNTIL, AGAIN or REPEAT. When executing UNTIL, a return to BEGIN will occur if the top of the stack is false; for AGAIN and REPEAT a return to BEGIN always occurs.
BFLNAM
Resident
( flag ---- [ ] | addr )
Helper routine that gets a blocks filename from the input stream into PAD or HERE and passes a name pointer (addr) if flag is true (used on command line), but passes nothing if flag is false ( \(a d d r\) is compiled by SLIT in a colon definition).
BL
( --- char )
A constant that leaves the ASCII value 32 (20h) for "blank".
BLANKS
Resident
( addr count --- )
Fills an area of memory beginning at \(a d d r\) with count blanks.
BLK
Resident
( --- addr )
A user variable containing the block number being interpreted. If zero, input is being taken from the terminal input buffer.
BLKRW
Resident
( [ bfnaddr | \#blks bfnaddr | bufaddr blk\# ] opcode --- flag )
Blocks I/O utility routine called by DO_BRW. Addresses passed point to blocks file name (bfnaddr) and block RAM buffer (bufaddr). The number of items required on the stack depends on the opcode (passed by the corresponding command) as follows:
\[
\begin{aligned}
(\text { bfnaddr }-14-- \text { - flag ) } & \text { passed by USEBFL } \\
\text { (\#blks bfnaddr }-16 \text {--- flag ) } & \text { passed by MKBFL } \\
\text { (bufaddr blk\#-18 --- flag ) } & \text { passed by RBLK } \\
\text { (bufaddr blk\#-20 --- flag ) } & \text { passed by WBLK }
\end{aligned}
\]

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.

ASCII Collating Sequence: ! " \# \$ \% \& ( ) * + , . / digits : ; < = > ? @ HLPHA [ \] ^ _` alpha \{ | \}~
( \(n\)--- 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 .
( \(n \mid[]---\) )
This word's functionality has been changed from the original TI Forth functionality, which essentially was a continuation of COLD. It now simply restarts the system as though the user had just chosen the second or third option on the cartridge menu screen. It expects the default text mode \(n\) on the stack. The value \(n\) is forced to 0 or 1 for TEXT80 or TEXT, respectively. BOOT may be executed with nothing on the stack, in which case TEXT is used.

A key may be held down to select the boot disk number or <ENTER> may be held down to prevent loading of FBLOCKS.
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.
( --- 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 - -- blk \({ }_{2}\) )
Places a binary image (starting at \(b l k_{1}\) and going as far as necessary) of all dictionary contents between \(a d d r\) and HERE. The next available Forth block number \(b l k_{2}\) 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

ASCII Collating Sequence: ! " \# \$ \% \& ( ) * + , - / digits : ; < = > ? @ ALPHA [ \]^_`alpha \{ | \} ~
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.
( b addr --- )
Stores the low-order byte ( 8 bits) of \(b\) (16-bit number on the stack) at \(a d d r\).
C,
( \(b\)--- )
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, wordoriented 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 .
(---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).
( --- addr )
A user variable whose value is the number of characters per line. See C/L.
C@
( \(a d d r\)--- \(b\) )
Leaves the 8 -bit contents \(b\) of memory address \(a d d r\) on the stack.
[immediate word]
Resident
Used in a colon definition to initiate the construct:
CASE
\[
n_{1} \text { OF ... ENDOF }
\]
\(n_{2}\) OF ... ENDOF
..
ELSEOF ... ENDOF <==This clause is optional. See below.
ENDCASE
Compile time: ( --- csp 4 )
CASE gets the value \(c s p\) of CSP to the stack for later restoration at the end of

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

ENDCASE 's compile-time activity. It stores the current stack position in CSP to help ENDCASE track how many OF clause 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 \(\mathbf{0 F}\), ELSEOF or ENDCASE to consume. If \(n=\boldsymbol{n}_{1}\), 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 \(\mathbf{O F}\), 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 .

Use of the optional ELSEOF obviates the necessity of putting any difficult-to-design default action between the last ENDOF and ENDCASE .

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
( \(f_{1}---f_{2}\) )
Finds the least integer \(f_{2}\) (in floating point format) not less than the floating point number \(f_{1}\).

CELLS
More Useful Stack Words etc. [41]
( \(n---2 n\) )
Replace \(n\) (a number of cells) with \(2 n\) (the number of bytes in \(n\) cells).
CF?
Compact Flash Utilities [69]
( --- flag )
Checks for the magic number, AA03h, at VRAM address, 3FF8h, where the nanoPEB or CF7+ DSR places it. It leaves a true flag (1) if found and a false flag (0) if not.
CFA
Resident
( \(p f a\)--- cfa )
Converts the parameter field address \(p f a\) of a definition to its code field address \(c f a\).
CFMOUNT
Compact Flash Utilities [69]
( vol\# dsk\# --- )
Mounts the volume vol\# in the virtual disk dsk\#. The following entry will mount volume \#234 in DSK2:

2342 CFMOUNT

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

The volumes mounted by CFMOUNT will persist only through the current session of fbForth 2.0. This includes cycling through COLD and BOOT. A reset to the TI99/4A title screen will mount the three volumes stored in the CF card before fbForth 2.0 was started. As you can see, you will need to use some other means to write the volume mounts to the CF card. The following permanent mounting methods are available:
- CALL MOUNT(vol\# dsk\#)-TI Basic command, available when a nanoPEB or CF7+ is attached to the TI-99/4A;
- CFMGR—TI-99/4A program supplied with the nanoPEB or CF7+;
- CF2K-TI-99/4A program by Fred Kaal (www.ti99-geek.nl);
- TI99Dir.exe—PC program by Fred Kaal (www.ti99-geek.nl).

\section*{Compact Flash Utilities [69]}
( --- volDSK1 volDSK2 volDSK3 )
Leaves on the stack the volume numbers associated with DSK1, DSK2 and DSK3.
Resident
( \(n_{1} n_{2} n_{3} n_{4}\) char --- )
Defines character \# char to have the pattern specified by the 4 numbers \(\left(n_{1}, n_{2}, n_{3}, n_{4}\right)\) on the stack. The definition for character \(\# 0\) by default resides at 800 h . Each character definition is 8 bytes long with each number on the stack representing two bytes.
(char --- \(n_{1} n_{2} n_{3} n_{4}\) )
Places the 4-cell (8-byte) pattern of a specified character char on the stack. By default, the definition for character \(\# 0\) resides at 800 h .
( blk --- )
Gets a block buffer for block\# blk, fills it with blanks and marks it as updated.
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 \(a d d r\) of the line to be written in memory, the number of characters count

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

\section*{blk CLOAD WWWW}

CLOAD will load Forth block blk only if the word WWWW is not in the CONTEXT vocabulary. WWWW should be the last word loaded when the series of blocks beginning with \(b l k\) is loaded. A block number of \(0(b l k=0)\) will suppress loading of the current Forth block if the specified word has already been compiled.
```

CLR_BLKS
( $b l k_{1} b l k_{2}---$ )
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
( --- )
Clears the display screen by filling the screen image table with blanks. The screen image table runs from SCRN_START to SCRN_END .
( --- )
Closes the file whose PAB (Peripheral Access Block) is pointed to by PAB-ADDR .
CMOVE
( addr $_{1}$ addr $r_{2}$ count --- )
Moves count number of bytes from $a d d r_{1}$ to $a d d r_{2}$. The contents of $a d d r_{1}$ is moved first, proceeding toward high memory. This is not overlap safe for $a d d r_{1}<a d d r_{2}$.
CMP\$
( --- -1|0|+1 ) ( SS: str $r_{1}$ str $r_{2}--$ 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 $s t r_{1}<s t r_{2}, 0$ if $s t r_{1}=s t r_{2}$ and +1 if $s t r_{1}>s t r_{2}$. The strings are retained. For example,

```
$" hello" $" HELLO" CMP$ . \underline{1 ok:0}
$" hello" $" hello" CMP$ . \underline{0}\mathrm{ ok:0}
$" hell" $" hello" CMP$ . \underline{-1 ok:0}
```

A case insensitive comparison can easily be built as follows:
ASCII Collating Sequence: ! " \# \$ \% \& ( ) * + , - / digits : ; < = > ? @ ALPHA [ \ ] ^ _` alpha \{ | \}~

```
: CMPCI$ ( --- flag ) ( SS: str 
OVER$ OVER$ UCASE$ SWAP$
UCASE$ CMP$ DROP$ DROP$ ;
```

The above code creates copies of $\operatorname{str}_{1}$ and str $_{2}$ (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 $s t r_{1}$ and $s t r_{2}$ are removed from the string stack; thus, $s t r_{1}$ and $s t r_{2}$ are retained, unchanged.
CODE (Deprecated TI Forth word) Resident
( --- )
CODE has been maintained for TI Forth compatibility. It has been deprecated in favor of ASM: and CODE: , q.v. See Chapter 9 "The fbForth 2.0 TMS9900 Assembler" for details.
[immediate word]
Resident
( --- ) ( IS: <newword $>$ [<mc> ...]; CODE )
CODE: opens a CODE: CCCC ... ; CODE word definition that converts numbers in the IS and compiles them before the interpreter sees them, obviating the necessity of using the , required by CODE in TI Forth. It is also faster. $\mathbf{N}>\mathbf{S}, q . v$, has been provided to push numbers from the IS to the stack for necessary calculations. See Chapter 9 "The fbForth 2.0 TMS9900 Assembler" for more information.
COINC
Resident
( $\operatorname{spr}_{1}$ spr $_{2}$ 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.

## COINCXY <br> Resident

( dotcol dotrow spr tol --- flag )
Detects a coincidence between a specified sprite and a given point (dotcol,dotrow) within a given tolerance of tol dot positions. A true flag indicates a coincidence.
COLD
Resident
( --- )
COLD is the cold-start procedure that may be called from the terminal to remove application programs and to restart fbForth 2.0. It is also the last routine executed by the fbForth $\mathbf{2 . 0}$ startup code. Formerly, it was a high-level Forth word that called another high-level Forth word ( BOOT ) at its conclusion. They have both been combined into a single ALC routine that (re-)sets the Forth environment to the default startup conditions.

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

In restarting fbForth 2.0, COLD resets user variables to their startup values, including the dictionary pointer (to point to just after the resident dictionary), resets the current blocks file to the default DSK $n$.FBLOCKS ( $n$ is the boot disk number), loads block \#1 and executes ABORT, q.v.
COLD may be called from the terminal to remove application programs and to restart fbForth 2.0.

See § 1.5.5 "Changes to COLD" for more detail.
COLOR
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 $n_{2}$.
COLTAB
Resident
( --- vaddr )
A constant whose value is the beginning VDP address of the color table. The default value is $\mathbf{3 8 0 h}$. This constant can only be changed via user variable number 22 h .
COMPILE
Resident
( --- ) (IS: <word>)
COMPILE 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.e., an immediate word. This effectively defers compilation of the word following COMPILE until the word containing them is executed within the definition of yet another word.
When the word containing COMPILE executes during the compilation of a new word, the execution address $c f a$ of the word following COMPILE is copied (compiled) into the dictionary entry for the new word's definition. For example,

```
WORD1 ... COMPILE WORD0 ... ; IMMEDIATE
```

: WORD2 WORD1 ... ;

When WORD2 is compiled, WORD1 executes, which executes COMPILE to place the $c f a$ of WORD0 into the definition of WORD2 .
CONSTANT
( $n$--- ) (IS: <new name>)
A defining word used in the form:
$n$ CONSTANT cccc
to create word $\mathbf{~ c c c c}$, with its parameter field containing $n$. When $\mathbf{c c c c}$ is later executed, it will invoke CONSTANT 's execution procedure to push the value of $n$ to the stack.

## CONTEXT

Resident
( --- addr )
A user variable containing a pointer to the vocabulary within which dictionary searches will first begin.

Resident
( $f_{1}---f_{2}$ )
Calculates the cosine of $f_{1}$ radians and leaves the floating point result $f_{2}$ on the stack.
( $a d d r_{1}---a d d r_{2} b$ )
Leave the byte address $a d d r_{2}$ and byte count $b$ of the packed character string (see footnote 4 on page 22) beginning at $a d d r_{1}$. It is presumed that the first byte at $a d d r_{1}$ contains the character count $b$ and that the actual text starts with the second byte. Typically, COUNT is followed by TYPE .

## CPYBLK

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:

```
CPYBLK src_start src_end src-file dst_start dst-file,
```

where src_start and src_end are source start and end block numbers, src-file is the source blocks file, dst_start is the destination start block number and dstfile is the destination blocks file.
Example:
CPYBLK 410 DSK1.FBLOCKS 25 DSK2.MYBLOCKS $\underline{4} \underline{6} \underline{\mathbf{7}} \underline{8} \underline{1} \underline{10}$ ok: 0
will copy blocks $4-10$ from DSK1.FBLOCKS to DSK2.MYBLOCKS, starting at block 25.
CR Resident
( --- )
Transmit a carriage return and a line feed to the current output device.

## CREATE

( --- )
A defining word used in the form:

## CREATE cccc

by such words as : , <BUILDS, ASM: and CODE: to create a dictionary header for a Forth definition. The code field contains the address of the word's parameter field.

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

Space for the parameter field is not reserved by CREATE. The new word is created in the CURRENT vocabulary. CREATE limits new word names to 31 characters in length in fbForth 2.0 by ANDing the count byte with 31 . The reason for this is that the three leftmost bits of the count byte are control bits (see Chapter 12 " fbForth 2.0 Dictionary Entry Structure"), which leaves only 5 bits for the character count.

Resident
( --- addr )
A user variable temporarily storing the stack pointer position for compilation error checking.
CURPOS
Resident
( --- addr )
A user variable that stores the current VDP (Visual Display Processor) screen cursor position.
CURRENT
Resident
( --- addr )
A user variable pointing to the vocabulary into which new definitions will be compiled. DEFINITIONS will store the contents of CONTEXT into CURRENT. At system startup, CURRENT points to the FORTH vocabulary.
D+ Resident
$\left(d_{1} d_{2}--d_{3}\right)$
Leave the double number sum of two double numbers $\left(d_{3}=d_{1}+d_{2}\right)$.
D+-
( $d_{1} n--d_{2}$ )
Negate double number $d_{1}$ if the sign of $n$ is negative, leaving the result as $d_{2}$.
D.

Resident
( $d$--- )
Print a signed double number from a 32 -bit two's complement value $d$. The highorder 16 bits are most accessible on the stack. Conversion is performed according to the current radix in BASE . A blank follows. Pronounced "d dot".
D.R

Resident
( $d n---$ )
Print a signed double number $d$ right-aligned in a field $n$ characters wide.
DABS
$\left(d_{1}--d_{2}\right)$
Leave the absolute value $d_{2}$ of a double number $d_{1}$.
DATA[
[immediate word]
Resident
( --- addr $n$ ) (IS: $n_{1} \ldots n_{\mathrm{n}}$ )
DATA [ opens a DATA [ ... ]DATA construct that compiles numbers and leaves their
beginning address $a d d r$ and cell count $n$ on the stack. If compiling within another definition, DATA [ compiles DATA [] and cell count $n$ in front of the array.
DATA[]
Resident
( --- addr n)
Runtime routine compiled by DATA[ to push to the stack the address $a d d r$ and number of cells $n$ of the number array that follows it in a word definition.

DCHAR
Resident
( addr cnt chr --- )
DCHAR is similar to "CALL CHAR" in TI Extended Basic, but is not limited to 4 characters. It is similar to CHAR, but uses an array of numbers instead of the stack for pattern definition. It is used to define one or more characters starting at the pattern address of character chr. DCHAR moves cnt cells from address addr to the pattern address of character $c h r$ in VRAM.
DCOLOR
Resident
( --- addr )
A variable which contains the dot-color information used by DOT. 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.
DCT
Resident

## ( --- addr )

A constant that pushes to the stack the address $a d d r$ of the Default Colors Table for all VDP modes. It also gives the user access to the default text mode because it immediately follows the table.

| VDP Mode | Table <br> Offset <br> (bytes) | Screen/ <br> Text <br> Colors | Color <br> Table <br> Colors |
| :--- | :---: | :---: | :---: |
| TEXT80 | 0 | 4 Fh | 00 h |
| TEXT | 2 | 4 Fh | 00 h |
| GRAPHICS | 4 | F4h | F4h |
| MULTI | 6 | 11h | F4h |
| GRAPHICS2 | 8 | FEh | 10h |
| SPLIT | 10 | FEh | F4h |
| SPLIT2 | 12 | FEh | F4h |
|  | Table <br> Offset <br> (bytes) |  |  |
| Default Text |  |  |  |
| Mode | 14 | 0001h |  |

All changes to the above values will survive execution of COLD .

[^1]
## DECIMAL

( --- )
Set the radix in BASE for decimal input/output.
DEFBF
Resident
( --- addr )
Gets the address $a d d r$ of the default blocks filename (DSK1.FBLOCKS) in low RAM to the stack. This address points to the string-length byte and can be displayed by

## COUNT TYPE

If the boot disk is other than DSK1, that will be reflected in the name displayed by the above Forth code.
DEFINITIONS
Resident
( --- )
Sets the CURRENT vocabulary to the CONTEXT vocabulary by copying the contents of CONTEXT to CURRENT. Executing a vocabulary name makes it the CONTEXT vocabulary and executing DEFINITIONS makes both specify the same vocabulary. The following example will make both CONTEXT and CURRENT point to the FORTH vocabulary, which is the system default:

```
FORTH DEFINITIONS ok:0
```

DEG/RAD
Resident
(---f)
Constant in floating point format representing degrees/radian $=57.295779513082$.
DELALL
Resident
( --- )
Delete all sprites. DELALL stops sprite motion, fills the sprite motion table with zeroes and stores D0h in the $\boldsymbol{y}$ position of all 32 sprites to leave them in an undefined state. DELALL does nothing to the sprite descriptor table. See § 6.6.2 for details.

DELALL must be used to initialize sprites after changing to the desired VDP mode.
DELSPR
Resident
( spr --- )
Delete the specified sprite by positioning it off-screen at $\boldsymbol{x}=1, \boldsymbol{y}=192$; setting it to sprite pattern $\# 0$; and clearing its motion table entries.
DEPTH
( --- $n$ )
Return the number of cells on the parameter stack. This word is used by the new command-line (ok: $\boldsymbol{n}$ ) response, where $\boldsymbol{n}$ indicates stack depth.
DEPTH\$
Stack-based String Library [42]
(---n) (SS: --- )
Returns the current depth $n$ of the string stack, with 0 meaning the string stack is empty.

[^2]( char $n_{1}$--- false $\mid n_{2}$ true )
Convert the ASCII character char (using number base $n_{1}$ ) to its binary equivalent $n_{2}$, accompanied by a true flag. If the conversion is invalid, leave only a false flag. For example, "DECIMAL 5310 DIGIT" will leave" $5 \mathbf{1}$ " on the stack because 53 is the ASCII code for ' 5 ' and is a legitimate digit in base 10. On the other hand, "DECIMAL 7416 DIGIT" will leave only " 0 " on the stack because 74 is the ASCII code for ' $J$ ' and is not a legitimate digit in base 16. However, "DECIMAL 7420 DIGIT" will leave " $19 \mathbf{1}$ " on the stack because ' J ' is a legitimate digit in base 20.

DIR
DIR--Disk Catalog Utility [36]
( --- )
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". DIR will not load if CAT is loaded.

Usage: DIR DSK1.
DISK_BUF
Resident
( --- addr )
A user variable that points to the first byte in VDP RAM of the 128-byte fbForth 2.0 record buffer.

DKB+
Resident
( $n$--- )
Defining word used to create words that calculate addresses from user variables containing offsets from fbForth 2.0'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 $n$ passed to DKB+ .
Usage: userVarOffset DKB+ new_word
DLITERAL [immediate word] Resident
Compile time: ( $d$--- ) Runtime: ( $--d$ ) Interpreting: ( --- )
Same behavior as LITERAL, q.v., except for a double number $d$
DLT
Resident
( --- )
The file I/O routine that deletes the file whose PAB (Peripheral Access Block) is pointed to by PAB-ADDR .
DMINUS
Resident
( $d_{1}---d_{2}$ )
Convert $d_{1}$ to its double number two's complement $d_{2}$, i.e., $d_{2}=-d_{1}$.

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


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.

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 $a d d r$ 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>
[immediate word] Resident ( --- )
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:
[immediate word]
Resident
( --- ) (IS: [<alc>];ASM)
DOES $>$ ASM: has the same function as DOES> , q.v., for defining the runtime action within a high-level defining word, except that its runtime action is defined using Assembly Language Code (ALC) rather than high-level Forth code. DOES>ASM: must be paired with ; ASM to enclose the ALC:

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

See Chapter 9 "The fbForth 2.0 TMS9900 Assembler" for more information.

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

This is the machine-code (MC) version of DOES>ASM: , q.v. It must be paired with ; CODE to enclose the MC:

```
: cccc <BUILDS ... DOES>CODE: ... ;CODE
```

DOES $>$ CODE: compiles any numbers it finds in the IS by jumping into CODE: , q.v., to compile the MC. Just as with CODE: , use $\mathbf{N}>\mathbf{S}, q . v$., to push numbers to the stack for necessary calculations. See Chapter 9 "The fbForth 2.0 TMS9900 Assembler" for more information.
(dotcol dotrow --- )
In bitmap graphics, plots a dot at (dotcol,dotrow) in whatever mode is selected by DMODE and in whatever color is selected by DCOLOR .
DO_BRW
Resident
( [ bfnaddr | \#blks bfnaddr | bufaddr blk\#] opcode --- )
Helper routine that executes BLKRW and processes returned flag. See BLKRW for items required on stack for each opcode and for an explanation of the stack effects abbreviations.
( --- addr )
A user variable, the dictionary pointer, which contains the address of the next free memory above the dictionary. The value may be read by HERE and altered by, and ALLOT , among other words.
( --- addr )
A user variable containing the number of digits to the right of the decimal point on double integer input. It may also be used to hold output column location of a decimal point in user-generated formatting. The default value on single number input is -1 for no decimal point. DPL is updated for every double number input.

DRAW
Resident
( --- )
Sets DMODE equal to 0 . This means that dots are plotted in the 'on' state.
DROP
Resident
( $n$--- )
Drop the top number from the stack.
DROP\$
Stack-based String Library [42]
( --- ) (SS: str --- ) "drop string"
The word DROP\$ removes the topmost string item from the string stack. For

ASCII Collating Sequence: ! " \# \$ \% \& ( ) * + , . / digits : ; < = > ? @ HLPHA [ \ ] ^ _` alpha \{ | \}~
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.
DSPLY
( --- )
Assigns the attribute DISPLAY to the file pointed to by PAB-ADDR .
DSRLNK
Resident
( --- )
Links an fbForth 2.0 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 8356 h . See the Editor/Assembler Manual 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 .
DTOG
( --- )
Sets DMODE equal to 2. This means that each dot plotted takes on the opposite state as the dot currently at that location.
( addr $n---$ )
Print the contents of $n$ memory locations beginning at addr. Both addresses and contents are shown in hexadecimal notation. DUMP is 80 -column-text-mode aware if your computer is so equipped. See PAUSE .
( $n---n n$ )
Duplicates the value on top of the stack.
DUP\$
Stack-based String Library [42]
( --- ) ( SS: str $r_{1}--s t r_{1} s t r_{1}$ ) "duplicate string"
The word DUP\$ duplicates the top item on the string stack. For example,

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

DXY
( dotcol $_{1}$ dotrow $_{1}$ dotcol $_{2}$ dotrow $_{2}--n_{1} n_{2}$ )
Places on the stack the square of the $\boldsymbol{x}$ distance $n_{1}$ and the square of the $\boldsymbol{y}$ distance $n_{2}$ between the points (dotcol ${ }_{1}$, dotrow $_{1}$ ) and (dotcol ${ }_{2}$, dotrow $_{2}$ ).

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

## ECOUNT

Resident
( --- addr )
A user variable that contains an error count. This is used to prevent error recursion.

```
ED@
```

Resident
( --- )
Brings you back into the 40/80-column editor on the last fbForth 2.0 block you edited. This block is pointed to by SCR. 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.
[immediate word]
Resident
Occurs within a colon-definition in the form:

> IF ... ELSE ... ENDIF

Compile time: ( $a d d r_{1} 2$--- $a d d r_{2} 2$ )
ELSE emplaces BRANCH, reserving a branch offset and leaves the address $a d d r_{2}$ 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 $a d d r_{1}$ to HERE and storing it at $a d d r_{1}$.
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.
ELSEOF
[immediate word]
Resident
ELSEOF is the start of the catchall default ELSEOF ... ENDOF clause that occurs inside a colon definition as the optional default clause within the CASE ... ENDCASE construct, just before ENDCASE . If execution reaches ELSEOF , the words between ELSEOF and ENDOF will always be executed. There should be no value preceding ELSEOF because the runtime stack value will be duplicated in its place to force a match by the compiled ( $\mathbf{O F}$ ).

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

Use of the ELSEOF clause guarantees that ENDCASE will never execute. It is a lot easier to use an ELSEOF clause instead of trying to contrive a default action ahead of ENDCASE . Compare with a description of just such a default action at CASE .
Compile time: ( 4 --- addr 5 )
Checks for the value 4 on the stack left there by CASE or a previous ENDOF, compiles DUP to force runtime comparison of the value on the stack with itself (guaranteeing a match), compiles ( $\mathbf{O F}$ ), leaves its address addr for branching resolution by ENDOF and leaves a 5 for its matching ENDOF to check.
Runtime: ( $n$ n --- )
Duplicates the value $n$, which was on top of the stack when CASE 's runtime action occurred. Comparison of the two identical numbers forces execution of the words between ELSEOF and ENDOF . See CASE and ENDOF .
( 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
( --- )
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
( $a d d r_{1}$ char --- addr $r_{1} n_{1} n_{2} \quad n_{3}$ )
The text scanning primitive used by WORD. From the text address $a d d r_{1}$ and an ASCII-delimiting character char, is determined the byte offset $n_{1}$ to the first nondelimiter character, the offset $n_{2}$ to the delimiter after the text and the offset $n_{3}$ 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 $n_{3}$ and calculate the parsed word's length as $n_{2}-n_{1}$ for use in constructing the packed character string (see footnote 4 on page 22) 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:

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

1) $n_{1} n_{3}\{0\} n_{2}$
2) $n_{1}\{\operatorname{chr}\} \ldots\{\operatorname{chr}\} n_{2} n_{3}\{0\}$
3) $n_{1}\{\operatorname{chr}\} \ldots\{\operatorname{chr}\} n_{2}\{\operatorname{delim}\} n_{3}\{\operatorname{chr} \mid 0\} \ldots$

The offsets, $n_{1}, n_{2}$ and $n_{3}$ 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 $\operatorname{mind}$ is that $n_{3}$ 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.
[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 ${ }_{1} \ldots$ addr $_{n} 4$--- )
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 clauses for which it must calculate branch distances from the addresses $\left(a d d r_{1} \ldots a d d r_{n}\right)$ that each ENDOF left on the stack.
Runtime: ( $n$--- )
If all $\mathbf{O F}$ 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 $\mathbf{O F}$ clause.
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 .
A better default action is to use an ELSEOF clause (with no preceding value) as the last clause before ENDCASE. See ELSEOF for more information.
[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 $a d d r$. 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

ASCII Collating Sequence: ! " \# \$ \% \& ( ) * + , . / digits : ; < = > ? @ HLPHA [ \ ] ^ _` alpha \{ | \}~
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: ( $a d d r_{1} 5---a d d r_{2} 4$ )
ENDOF checks for a 5 on the stack. It then compiles BRANCH, leaves its address addr $r_{2}$ 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 $a d d r_{1}$ to HERE for its matching $\mathbf{O F}$ and stores the value at the spot reserved for it at $a d d r_{1}$.

Runtime: ( --- )
ENDOF causes execution to proceed after ENDCASE. See OF .
ERASE
Resident
( addr $n$--- )
Clear $n$ bytes of memory to zero starting at $a d d r$.
ERROR
Resident
( $\left.n_{1}--n_{2} n_{3} \mid[]\right)$
ERROR processes error notification and restarts the interpreter. WARNING is first examined. If WARNING $<1$, (ABORT) , q.v., is executed. The sole action of (ABORT) is to execute ABORT. This allows the user to (cautiously!) modify this behavior by replacing the $c f a$ of (ABORT) with the $c f a$ of the user's error procedure. ABORT clears the stacks and executes QUIT, which stops compilation and restarts the interpreter.
If WARNING $\geq 0$, ERROR clears ECOUNT and the parameter stack. Then, if the input stream is coming from the loading of blocks and not the terminal, ERROR leaves the contents of IN $n_{2}$ and BLK $n_{3}$ on the stack to assist in determining the location of the error. Execution of WHERE, at this point, will open the offending block in the editor and place the cursor at the text immediately following the token that caused the error.
If WARNING $>0$, ERROR prints the error text of system message number $n_{1}$. If WARNING $=0$, ERROR prints $n_{1}$ as an error number (This was used in TI Forth in a non-disk installation; but, this is unnecessary in fbForth 2.0 because the system messages are always present in cartridge ROM). The last thing ERROR does is to execute QUIT , which, as above, stops compilation and restarts the interpreter.
EXECUTE
Resident
( cfa --- )
Execute the definition whose code field address is on the stack. The code field address is also called the compilation address.
( --- )
EXIT is a synonym for ; $\mathbf{S}$, which stops interpretation of a Forth block or ends the current word's execution and returns to the calling procedure.
EXP
Resident
( $f_{1}---f_{2}$ )
Raises $\boldsymbol{e}$ to the power specified by the floating point number $f_{1}$ on the stack and leaves the result $f_{2}$ on the stack.
( addr count --- )
Transfer characters from the terminal to addr until <ENTER> or count characters have been received. The character count is not stored with the string. One or more nulls are added at the end of the text.
( faddr --- )
Stores a floating point number $f$ into the 4 words (cells) beginning with the specified address.
F* Resident
$\left(f_{1} f_{2}--f_{3}\right)$
Multiplies the top two floating point numbers on the stack and leaves the result on the stack. $f_{1} * f_{2}=f_{3}$.
F+ Resident
$\left(f_{1} f_{2}---f_{3}\right)$
Adds the top two floating point numbers on the stack and places the result on the stack. $f_{1} * f_{2}=f_{3}$.
F-
$\left(f_{1} f_{2}--f_{3}\right)$
Subtracts $f_{2}$ from $f_{1}$ and places the result on the stack $\left(f_{1}-f_{2}=f_{3}\right)$.
F->S
Resident
( $f---n$ )
Converts a floating point number $f$ on the parameter stack into a single precision number $n$.
F-D" [immediate word] Resident
(---) (IS: filename" )
Expects a file descriptor ending with a " to follow. This instruction places the file descriptor in the PAB (Peripheral Access Block) pointed to by PAB-ADDR .

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

Resident
( $f$--- )
Prints a floating point number $f$ in TI Basic format to the output device.
F/
Resident
$\left(f_{1} f_{2}--f_{3}\right)$
Divides $f_{1}$ by $f_{2}$ and leaves the floating point quotient $f_{3}$ on the stack. $f_{1} / f_{2}=f_{3}$.
F0<
Resident
( $f$--- flag )
Compares the floating point number $f$ on the stack to 0 . If it is less than 0 , a true flag is left on the stack, else a false flag is left.
F0=
Resident
( $f$--- flag )
Compares the floating point number $f$ on the stack to 0 . If it is equal to 0 , a true flag is left on the stack, else a false flag is left.
$\mathbf{F}<\quad$ Resident
( $f_{1} f_{2}---$ flag $)$
Leaves a true flag if $f_{1}<f_{2}$, else leaves a false flag.
$\mathrm{F}=\mathrm{Resident}$
( $f_{1} f_{2}---$ flag $)$
Leaves a true flag if $f_{1}=f_{2}$, else leaves a false flag.
F> Resident
( $f_{1} f_{2}---$ flag $)$
Leaves a flag if $f_{1}>f_{2}$, else leaves a false flag.
$F>R$
( $f$---) ( R: --- $f$ )
Moves the 8-byte floating point number $f$ from the parameter stack to the return stack. See R>F .
( $a d d r---f$ )
Retrieves the floating point contents $f$ of the given address ( 4 words) and places it on the stack.

FABS
Resident
( $f_{1}---f_{2}$ )
Converts the floating point number $f_{1}$ to its absolute value $f_{2}$.

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

( -- -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.
( $f$--- )
Drops the top floating point number $f$ from the stack.
( $f---f f$ )
Duplicates the top floating point number $f$ on the stack.
FENCE
Resident
( --- addr )
A user variable containing an address (usually the nfa of a Forth word) below which FORGETting 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 UNFORGETABLE [sic] will report false; however, FORGET will still trap that error.
FFMT.
Resident
( $f$ [intLen fracLen] optMask --- )
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 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:

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

- 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. . 0000000000000 ok:16
210 9 FFMT. 1.234567890E-** ok:12
212 25 FFMT. 1.23456789012E-102 ok:8
214 31 FFMT. +1.2345678901230E-102 ok:4
2 15 9 FFMT. field too big! ok:0
```

Resident
( vaddr $_{1}$ addr vaddr $_{2}---$ ) ( IS: <file word> )
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 ( addr vaddr r FILE cccc
```

When cccc executes, PAB-ADDR, PAB-BUF and PAB-VBUF are set to $v a d d r_{1}, a d d r$ and $v a d d r_{2}$, respectively.
( $n$--- )
Change the number of files fibForth $\mathbf{2 . 0}$ 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.
( addr count b--- )
Fill memory beginning at $a d d r$ with count bytes of byte $b$.
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" $" green" 0 FIND$ . \underline{3 ok:0}
```

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

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\$ . $\underline{8}$ 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 .
( --- addr )
A constant that leaves the address of the first (lowest) block buffer.
( --- addr )
A user variable which contains the first byte of the disk buffer area.
(---n)
Returns on the stack the contents $n$, reported by the floating point library in its floating point error variable.

Resident
( $f_{1}--f_{2}$ )
Finds the greatest integer $f_{2}$ (in floating point format) not less than the floating point number $f_{1}$.
FLUSH
( --- )
Writes to disk all disk buffers that have been marked as updated.
FM/MOD
( $d n$--- rem quot )
A mixed magnitude math operator that performs floored division to leave the signed remainder rem and signed quotient quot from a double-number dividend $d$ and singlenumber divisor $n$. The quotient is rounded toward negative infinity and the remainder given the sign of the divisor. See Chapter 18 "Signed Integer Division" for more details.
FMINUS
Resident
( $f_{1}---f_{2}$ )
Negates $f_{1}$ by taking the two's complement of the cell on top of the stack, i.e., $f_{2}=-f_{1}$.

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

The top cell on the stack is the most significant cell (2 bytes) of a floating point number.

FNT
Resident
( --- )
FNT loads either the default font file (can be changed by user with USEFFL, q.v.) or the console font into the Pattern Descriptor Table (PDT) depending on the value of the user variable SCRFNT. The default font is loaded from DSK1.FBFONT by FNT (or from DSK $n$.FBFONT if key $n$ is held down) at fbForth 2.0 startup because SCRFNT $=-1$ at startup. The fbForth 2.0 system default font contains the patterns for ASCII character codes $0-127$. The font pattern for each character is 8 bytes, which means that 1 KiB of pattern code is loaded into the PDT. This font contains true lowercase characters with true descenders.

Executing COLD will maintain the currently selected font as the default. Restarting the system with BOOT , MON or a power cycle will restore loading of the system font from DSK1.FBFONT.

See Chapter 13 "Screen Fonts and the Font Editor" for more detail.
FONTED
Resident
( --- )
Typing FONTED 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.
( --- )
Executed in the form:
FORGET cccc
Deletes the definition named cccc from the dictionary along with all dictionary entries physically following it.
FORGET first checks the lfa of cccc to see if it is lower than the address in FENCE . If it is not, FORGET 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, FORGET updates HERE to the lfa of cccc, effectively deleting the desired part of the dictionary. Otherwise, an appropriate error message is displayed.
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 FORGETting it.
FORTH
[immediate word]
Resident
( --- )
The name of the primary vocabulary. Execution makes FORTH the CONTEXT vocabulary. Until additional user vocabularies are defined, new user definitions become a part of FORTH because it is at that point also the CURRENT vocabulary.

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

Because FORTH is immediate, it will execute during the creation of a colon definition to select this vocabulary at compile time.
FOVER
$\left(f_{1} f_{2}---f_{1} f_{2} f_{1}\right)$
Copies the second floating point number on the stack to the top of the stack.
FP1
(---f)
Pushes a floating point $1 f$ to the stack.
(---f)
Pushes a floating point $10 f$ to the stack.
( --- vaddr )
Pushes VRAM address vaddr of user screen font file PAB to stack.
FRAC
( $f_{1}---f_{2}$ )
Truncates $f_{1}$, leaving the fractional portion $f_{2}$ on the stack.
FRND
( ---f)
Generates a pseudo-random floating point number $f$ greater than or equal to 0 and less than 1.
FROT
$\left(f_{1} f_{2} f_{3}---f_{2} f_{3} f_{1}\right)$
Moves the third floating point number $f_{1}$ down from the top of the stack to the top of the stack.
FSWAP
$\left(f_{1} f_{2}---f_{2} f_{1}\right)$
Swaps the top two floating point numbers on the stack.
FVARIABLE
Resident
( --- addr ) (IS: )
A defining word used in the form:

## $f$ FVARIABLE cccc

to create word cccc, with its parameter field containing $\boldsymbol{f}$. When cccc is later executed, it will invoke FVARIABLE 's execution procedure to push cccc 's address $a d d r$ to the stack.

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

FXD

## ( --- )

Assigns the attribute FIXED to the file whose PAB (Peripheral Access Block) is pointed to by PAB-ADDR .
( col row --- char )
Returns on the stack the ASCII code char of the character currently at (col,row). Note: Rows and columns are numbered from 0 .
( col row --- )
Places the cursor at the designated column col and row row position. Note: Rows and columns are numbered from 0 .
GPLLNK Resident
( addr --- )
Links a Forth program to the Graphics Programming Language (GPL) routine located at the given address.
GRAPHICS Resident
( --- )
Converts from present display screen mode into standard Graphics mode configurations.
GRAPHICS2 Resident
( --- )
Converts from present display screen mode into standard Graphics2 (Bitmap) mode configuration.
HCHAR
Resident
( col row count char ---)
Prints a horizontal stream of a specified character char beginning at (col,row) and having a length count. Note: Rows and columns are numbered from 0 . HCHAR does not check to see whether (col,row) is within the screen buffer or whether count will overrun VRAM after the screen buffer. This is the same behavior as in TI Forth. This behavior will be changed in the next build of fbForth 2.0 to conform to how TI Basic and TI Extended Basic implement this function, i.e., in the next build, HCHAR will throw an error if it would start outside the screen buffer and it will wrap to the start of the screen buffer upon reaching the end of the screen buffer.
HERE
Resident
( --- addr )
Leave the address of the next available dictionary location.

[^3]( --- )
Set the numeric conversion base to sixteen (hexadecimal).
Resident
( --- addr )
A user variable that holds the address of the latest character of text during numeric output conversion.
( char --- )
Used between <\# and \#> to insert an ASCII character into a pictured numeric output string, e.g., 2E HOLD will place a decimal point.
HONK
Resident
( --- )
Produces the sound associated with incorrect input.
I
Resident
(---n)
Used within a DO loop to copy the loop index to the stack. $\mathbf{I}$ is a synonym for $\mathbf{R}$.
ID.
Resident
( $n f a---$ )
Print a definition's name from its name field address $n f a$.
IF [immediate word] Resident
Occurs in a colon definition in form:

```
IF (true part) ... THEN
IF (true part) ... ENDIF
IF (true part) ... ELSE (false part) ... THEN
IF (true part) ... ELSE (false part) ... ENDIF
```

Compile time: ( --- addr 2 )
IF compiles 0BRANCH and reserves space for an offset at $a d d r$; $a d d r$ and 2 are used later for resolution of the offset and error testing.
Runtime: (flag --- )
IF selects execution based on a Boolean flag. If flag is true (non-zero), execution continues ahead through the true part. If flag is false (zero), execution skips to just after ELSE to execute the false part when an ELSE clause is present. After either part, execution resumes after THEN (or ENDIF ). ELSE and its false part are optional. With no ELSE clause, false execution skips to just after THEN (or ENDIF).
IMMEDIATE
Resident
( --- )
Mark the most recently made definition so that when encountered at compile time, it will be executed rather than being compiled. i.e., the precedence bit in its header is

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

set. This method allows definitions to handle unusual compiling situations rather than build them into the fundamental compiler. The user may force compilation of an immediate definition by preceding it with [COMPILE] .
IN
Resident
( --- addr )
A user variable containing the byte offset within the current input text buffer (terminal or disk) from which the next text will be accepted. WORD uses and moves the value of IN .
INDEX
( $n_{1} n_{2}---$ )
Prints to the terminal a list of the line \#0 comments from Forth block $n_{1}$ through Forth block $n_{2}$. See PAUSE .
INIT\$
Stack-based String Library [42]
( size --- ) ( SS: --- ) "initialize string stack"
The string stack must be initialized to some convenient size by executing INIT\$ once the library is LOADed:

## 512 INIT\$ ok:0

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.
( --- )
Assigns the attribute INPUT to the file whose PAB is pointed to by PAB-ADDR .
INT
( $f_{1}--f_{2}$ )
Leaves the integer portion of a floating point number on the stack.
INTERPRET
Resident
( --- )
The outer text interpreter, which sequentially executes or compiles text from the input stream (terminal or disk) depending on STATE. If the word name cannot be found after a search of CONTEXT and then CURRENT, INTERPRET attempts to convert it into a number according to the current radix in BASE. That also failing, an error message echoing the name with a "?" will be given. Text input will be taken according to the convention for WORD . If a decimal point is found as part of a number, a double number value will be left. The decimal point has no other purpose than to force this action. See NUMBER .

## INTLNK

( --- addr )
A user variable which is a pointer to the Interrupt Service linkage.

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

INTRNL
Resident
( --- )
Assigns the attribute INTERNAL to the file whose PAB is pointed to by PAB-ADDR .
( --- addr )
A user variable that initially contains 0 to indicate that no user Interrupt Service Routine (ISR) has been installed. The user must modify ISR to contain the $c f a$ of the Forth routine to be executed each $1 / 60$ second. Next, the contents of the console ISR hook, 83C4h, must contain the address of the fbForth 2.0 ISR, which it does at startup. Note that the interrupt service linkage code address is always available in INTLNK.

The console ISR hook, 83C4h, should be zeroed before changing ISR and restored with the value in INTLNK after changing it.
See Chapter 10"Interrupt Service Routines (ISRs)" for much more detail.

J

JMODE
( $--n$ )
Used within an inner DO loop to copy the loop index of the next outer DO loop to the stack.
$\left(n_{1}--n_{2}\right)$
Executed by JOYST when JMODE $\neq 0$, JCRU allows input from joystick \#1 ( $n_{1}=1$ ) or \#2 $\left(n_{1}=2\right)$. 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, $2=\mathrm{W}, 4=\mathrm{E}, 8=\mathrm{S}$ and $16=\mathrm{N}$. Two-bit directional combinations are $18=\mathrm{NW}(\mathrm{N}+\mathrm{W}$ or $16+2)$, $20=$ NE, $10=$ SW and $12=$ SE. See $\S 6.8$ "Using Joysticks" for more information.
( $n_{1}--$ char $n_{2} n_{3}$ )
Executed by JOYST when JMODE = 0, JKBD allows input from joystick \#1 and the left side of the keyboard $\left(n_{1}=1\right)$ or from joystick $\# 2$ and the right side of the keyboard $\left(n_{1}=2\right)$. Values returned are the character code char of the key pressed, the $\boldsymbol{x}$ status $n_{2}$ and the $\boldsymbol{y}$ status $n_{3}$. See $\S 6.8$ "Using Joysticks" for more information.

## 

( --- addr )
A user variable that uses offset 26 h 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 .
$\left(\begin{array}{ll}\left.\left.n_{1}---\left[\begin{array}{lll}\text { char } & n_{2} & n_{3}\end{array}\right] \right\rvert\, n_{2}\right)\end{array}\right.$
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 ( $n_{1}=2$ ). Return values depend on the value in JMODE . If JMODE $=0$ (default), JOYST executes JKBD, which returns the character code char of the key pressed, the $\boldsymbol{x}$ status $n_{2}$ and the $\boldsymbol{y}$ status $n_{3}$. If JMODE $\neq 0$, JOYST executes JCRU, which reads only the joysticks and returns a single value with 0 or more of the 5 least significant bits set. See JCRU and § 6.8 "Using Joysticks" for their meaning.
( --- char )
Wait for the next terminal keystroke. Leave its ASCII (7-bit) value on the stack.
KEY8
( --- char )
Wait for the next terminal keystroke. Leave its full 8 -bit value on the stack.
(--- $n$ )
Returns on the stack the number of lines per Forth block.
LATEST
Resident
( --- nfa )
Leave the name field address $n f a$ of the most recently defined word in the CURRENT vocabulary. At compile time, this "latest" word will be the most recently compiled word.
LCASE\$
Stack-based String Library [42]
( --- ) (SS: str $r_{1}--$ 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
LD
Resident
( $n$--- )
The file I/0 process to load a program file from a disk into VDP RAM. The parameter $n$ specifies the maximum number of bytes to be loaded and is usually the size of the file on disk. The file's PAB must be set up and be the current PAB, to which PAB-ADDR points, before executing this word.
LDCR
CRU Words [5]
( $n_{1} n_{2}$ addr --- )
Performs a TMS9900 LDCR instruction. The CRU base address $a d d r$ will be shifted

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

left one bit and stored in workspace register R12 prior to executing the TMS 9900 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 $n_{2}$ 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. See $\S 11.3$ and CRU documentation in the Editor/Assembler Manual for more information.

Resident
( --- )
Force termination of a DO loop at the next opportunity by setting the loop limit equal to the current value of the index. The index itself remains unchanged, and the execution proceeds normally until LOOP or $\mathbf{+ L O O P}$ is encountered.

Stack-based String Library [42]
(len --- ) (SS: str $r_{1}--$ str $r_{1} s t r_{2}$ ) "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.
(--- len ) (SS: --- ) "length of string"
The word LEN\$ returns the length of the topmost string on the string stack. For example,

```
$" Hello world!" len$ . \underline{12 ok:0}
```

( $p f a$--- lfa )
Convert the parameter field address pfa of a dictionary definition to its link field address lfa.
LIMIT
( --- addr )
A constant which leaves the address $a d d r$ just above the highest memory available for a disk buffer.
( --- 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
( dotcol $_{1}$ dotrow $_{1}$ dotcol $_{2}$ dotrow $_{2}$--- )
The high resolution graphics routine which plots a line from (dotcol ${ }_{1}$,dotrow ${ }_{1}$ ) to

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

(dotcol ${ }_{2}$, dotrow $_{2}$ ). DCOLOR and DMODE must be set before this instruction is used.
( 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
( $n$--- )
Begin interpretation of Forth block $n$. Loading will terminate at the end of the Forth block or at ;S. See ;S and -->.
LOG
Resident
( $f_{1}---f_{2} \mid f_{1}$ )
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
$\left(f_{1}--f_{2} \mid f_{1}\right)$
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.

ASCII Collating Sequence: ! " \# \$ \% \& ( ) * + , . / digits : ; < = > ? @ HLPHA [ \ ] ^ _`alpha \{ | \}~
[immediate word]
Resident
Occurs in a colon definition in the form:
DO ... LOOP
Compile time: (addr 3 --- )
LOOP compiles (LOOP) and uses $a d d r$ 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 $---s t r_{2}$ ) "trim left of string"
The word LTRIM\$ removes leading spaces from the topmost string. For example,
\$" hello!" LTRIM\$ .\$ hello! ok:0
M*
Resident
( $n_{1} n_{2}---d$ )
A mixed magnitude math operation that leaves the double-number signed product $d$ of two signed numbers, $n_{1}$ and $n_{2}$.
(d n--- rem quot )
A mixed magnitude math operator that leaves the signed remainder rem and signed quotient quot from a double-number dividend $d$ and single-number divisor $n$. M/ uses user variable S|F, q.v., to determine whether to use SM/REM, q.v., for symmetric division (the default) or FM/MOD , q.v., for floored division. See Chapter 18 "Signed Integer Division" for more details.

M/MOD
Resident
( ud u --- urem udquot)
An unsigned mixed-magnitude math operation that leaves an unsigned doublenumber quotient udquot and an unsigned single-number remainder urem from an unsigned double-number dividend $u d$ and an unsigned single-number divisor $u$.

MAGNIFY
Resident
( $n_{1}---$ )
Alters the sprite magnification factor to be $n_{1}$. The value of $n_{1}$ must be $0,1,2$ or 3 .
MAX
Resident
$\left(\begin{array}{ll}n_{1} & \left.n_{2}---n_{3}\right)\end{array}\right.$
Leave the greater $n_{3}$ of the two numbers, $n_{1}$ and $n_{2}$.

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

MAXLEN\$
Stack-based String Library [42]
( \$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
```

Resident
( $n$ col row --- )
Places a square of color $n$ at (col,row ). Used in multicolor mode.
( --- )
Displays the available Load Options.
MESSAGE
Resident
( $n$--- )
Print on the selected output device the text of system error number $n$. If WARNING $=$ 0 , the message will simply be printed as a number ( $\mathbf{m s g} \boldsymbol{\# n}$ ). When WARNING $=0$ 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 $\mathbf{2 . 0}$ now reside in cartridge ROM. The word . LINE , q.v., can be used for this purpose.
( start end --- ) ( SS: $\operatorname{str}_{1}---\operatorname{str}_{1} \operatorname{str}_{2}$ ) "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" 37 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 $\left(s t r_{1}\right)$ is retained. Note also that the first character in the string (the leftmost character) is character number 0 .
$\left(\begin{array}{lll}n_{1} & n_{2}---n_{3}\end{array}\right)$
Leave the smaller $n_{3}$ of the two numbers ( $n_{1}$ and $n_{2}$ ).

Initializes the monitor screen for use with MCHAR .
$\left(n_{1}---n_{2}\right)$
Leaves the two's complement $n_{2}$ of a number $n_{1}$, i.e., negates $n_{1}$.
MKBFL
Resident
( ---) (IS: DSKn.<blocks filename>n)
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
( $n_{1} n_{2}$--- rem )
Leave the remainder rem of $n_{1} / n_{2}$, with the same sign as $n_{1}$. MOD is based on M/, which uses user variable S|F , q.v., to determine whether symmetric (the default) or floored division is used. See Chapter 18 "Signed Integer Division" for more details.
( --- )
Exit to the TI 99/4A color bar display screen and the system monitor program.
( $n_{1} n_{2}$ spr --- )
Assigns a horizontal $n_{1}$ and vertical $n_{2}$ velocity to the specified sprite $s p r$.
MOVE
( $a_{d d r_{1}} a d d r_{2} n---$ )
Moves the contents of $n$ cells (16-bit contents) beginning at $a d d r_{1}$ into $n$ cells beginning at $a d d r_{2}$. The contents of $a d d r_{1}$ is moved first, proceeding toward high memory. This is not overlap safe for $a d d r_{1}<a d d r_{2}$.
MULTI
( --- )
Converts from present display screen mode into standard Multicolor mode configuration.
MYSELF
[immediate word]
Resident
( --- )
Used in a colon definition. Places the code field address (cfa) of a word into its own definition. This permits recursion.

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

$\mathrm{N}>\mathrm{S}$
( --- n ) (IS: token ) "number to stack"
$\mathrm{N}>\mathbf{S}$ attempts to convert the next blank-delimited token in the input stream to a number in the current radix (number base). If successful, $\mathbf{N}>\boldsymbol{S}$ pushes the number to the stack. Otherwise, an error message is issued and the stack is cleared:
$\begin{array}{lll}\text {. BASE } & 10 \quad \text { ok: } 0 \\ N>S & 123 & \underline{o k: 1} \\ N>S & 12 X & 12 \mathrm{X} ?\end{array}$
$\mathrm{N}>\mathbf{S}$ is the only way to get numbers from the input stream to the stack while using CODE: and DOES>CODE: because those words compile numbers into the dictionary instead of pushing them to the stack as the interpreter does.
NEXT,
( --- )
NEXT, is one of only two words in the Assembler vocabulary that are part of the resident dictionary. The other is ; ASM. NEXT, puts 045Fh at HERE and advances HERE. This machine code for ALC, B *NEXT or B *R15, branches to the inner interpreter to fetch the next word to be executed. See ASM: , ; ASM and Chapter 9 "The fbForth 2.0 TMS9900 Assembler" for more information.
( $p f a---n f a$ )
Convert the parameter field address $p f a$ of a definition to its name field address $n f a$.
NIP
( $n_{1} n_{2}---n_{2}$ )
Remove from the stack the number that is under the top number.
( --- ) (SS: str str $_{2}$--- str $r_{2}$ ) "nip string"
The word NIP\$ removes the string underneath the topmost string from the string stack. For example,

$$
\begin{aligned}
& \text { \$" red" ok:0 } \\
& \text { \$" blue" ok:0 }
\end{aligned}
$$

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.
( --- )
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 $\mathbf{2 . 0}$ with the name, ' NULL '. The name field for NULL contains an ASCII 0 . Every fbForth $\mathbf{2 . 0}$ 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
( $a d d r$--- $d$ )
Convert a packed character string (see footnote 4 on page 22) 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.

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 $a d d r$ 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 $\mathbf{O F}$ and ENDOF will be executed. Otherwise, $n$ is put back on the stack for execution to continue after ENDOF . See CASE and ENDOF .
( --- )
Opens the file whose PAB is pointed to by PAB-ADDR .
OR Resident
$\left(n_{1} n_{2}---n_{3}\right)$
Leave the bit-wise logical OR $n_{3}$ of two 16-bit values, $n_{1}$ and $n_{2}$.

ASCII Collating Sequence: ! " \# \$ \% \& ( ) * + , . / digits : ; < = > ? @ HLPHA [ \ ] ^ _`alpha \{ | \}~
( --- addr )
A user variable that contains a value incremented by EMIT and EMIT8. The user may alter and examine OUT to control display formatting.
OUTPT
( --- )
Assigns the attribute OUTPUT to the file whose PAB is pointed to by PAB-ADDR .
OVER
$\left(\begin{array}{llll}n_{1} & n_{2}---n_{1} & n_{2} & n_{1}\end{array}\right)$
Copy the second stack value $n_{1}$ to the top of the stack.
OVER\$
Stack-based String Library [42]
( --- ) ( SS: str $r_{1}$ str $r_{2}--$ str $_{1}$ str $r_{2} s t r_{1}$ ) "over string"
The word OVER\$ pushes a copy of the string $s t r_{1}$ 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"
PAB-ADDR
Resident
( --- addr)
A variable containing the VDP address of the first byte of the current PAB (Peripheral Access Block).
PAB-BUF
( --- addr )
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.

PAB-VBUF
Resident
( --- addr )
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 PAB-VBUF is also stored in the file's PAB.

PABS
( --- addr )
A user variable which points to a region in VDP RAM, which has been set aside for creating PABs.

ASCII Collating Sequence: ! " \# \$ \% \& ( ) * + , . / digits : ; < = > ? @ HLPHA [ \ ] ^ _` alpha \{ | \}~
( --- addr )
Leave the address of the text output buffer, which is a fixed offset ( 68 bytes in fbForth 2.0) above HERE . Every time HERE changes, PAD is updated.
( --- )
Clears the display screen and places the cursor at the top, left corner. It is a shortcut for

```
CLS
0 0 GOTOXY
```

( $x y w h$--- )
Sets up a panel within the video display for SCROLL to scroll in any orthogonal direction with or without wrapping, depending on the value of WRAP . The panel will be $w$ characters wide, $h$ characters high with its upper, left corner at column $x$ and row $y$.
PAUSE
( --- flag )
Checks for a keystroke and issues false if none, true if <BREAK> (<CLEAR> or $<$ FCTN $+4>$ ) or idles until a second keystroke before issuing false (or true if second keystroke is <BREAK>). The words LIST, INDEX, DUMP and VLIST all call the word PAUSE. These routines exit when flag = true. PAUSE 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 <BREAK>.
( --- vaddr )
A constant which contains the VDP address of the Pattern Descriptor Table. Default value is $\mathbf{8 0 0 h}$. This constant can only be changed via user variable number $\mathbf{2 8 h}$.
( $n f a$--- pfa )
Convert the name field address nfa of a compiled definition to its parameter field address $p f a$.
( ---f)
A floating point approximation of $\pi$ to 13 significant figures. (3.141592653590)
PICK
More Useful Stack Words etc. [41]
( $+n$--- $[n]$ )
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.

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

0 PICK is equivalent to DUP .
1 PICK is equivalent to OVER .
PICK\$
Stack-based String Library [42]
(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.

## PLAY

Resident
( addr flag --- )
This word is ported from TurboForth code courtesy of Mark Wills.
PLAY starts the table of sound lists at address $a d d r$, depending on flag:

## Flag Action

0 Do not play if either sound table is active.
1 Unconditionally play, killing all previous sound tables.
Plays as sound table \#2, muting sound table \#1 for the duration of sound table
-1 \#2.
Sound lists consist of a list of sound commands starting with a byte count and ending with a duration count byte (sixtieths of a second) that is not included in the byte count. The last sound list should silence all four sound generators and end with a duration of 0 . See § 20 of the Editor/Assembler Manual for details on sound lists.
A sound table may be prepared for PLAY with DATA[ ... ]DATA by dropping the cell count:

DATA[ <sound list> ]DATA
DROP 1 PLAY
PLAYING?
Resident
( --- flag )
This word is ported from TurboForth code courtesy of Mark Wills.
PLAYING? checks both fbForth $\mathbf{2 . 0}$ sound status registers, ORs them and leaves that value on the stack as flag. If flag $=0$, no sound table is active.
PLAYING? is intended for use with PLAY, not SOUND. SOUND does not use the fbForth $\mathbf{2 . 0}$ sound status registers.
( --- 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: $\boldsymbol{n}$.
R Resident
( --- $n$ ) (R: $n---n$ )
Copy the top of the return stack to the parameter stack.
R\#
( --- addr )
A user variable which may contain the location of an editing cursor or other filerelated function.
R->BASE Resident
( --- ) (R: $n$--- )
Restore the current base from the return stack. See BASE->R .
R/W
Resident
( addr $n_{1}$ 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 $a d d r, n_{1}$ 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!.
R>
(---n) (R: $n---$ )
Remove the top value from the return stack and leave it on the parameter stack. See $>\mathbf{R}$ and $\mathbf{R}$.

ASCII Collating Sequence: ! " \# \$ \% \& ( ) * + , . / digits : ; < = > ? @ HLPHA [ \ ] ^ _`alpha \{ | \}~
$R>F$
Resident
(---f) ( $\mathrm{R}: f---$ )
Moves the 8 -byte floating point number $f$ from the return stack to the parameter stack. See F>R .

RAD/DEG
Resident
( ---f)
Constant in floating point format representing degrees/radian $=0.01745329251994$.
RANDOMIZE Resident
( --- )
Creates an unpredictable seed for the random number generator.
RBLK
Resident
( addr blk --- )
Read a block from the current blocks file.
RD
Resident
( --- count )
The file I/O instruction that reads from the current PAB. This instruction uses PAB-BUF and PAB-VBUF .
REC-LEN Resident ( $b$--- )
Stores the length $b$ of the record for the upcoming write into the appropriate byte in the current PAB.
REC-NO
Resident
( $n$--- )
Writes a zero-based record number $n$ into the appropriate location in the current PAB.
REPEAT
[immediate word]
Resident
Used within a colon-definition in the form:

```
BEGIN ... WHILE ... REPEAT
```

Compile time: ( $a d d r_{1} 1$ addr $r_{2} 4$--- )
At compile-time, REPEAT processes the 0BRANCH offset $a d d r_{2}$ and the offset from HERE to the loop-back address $a d d r_{1}$, which it stores at the space reserved for it at $a d d r_{1}$ by BEGIN , q.v. The values 1 and 4 are used for error testing.
Runtime: ( --- )
At runtime, REPEAT forces an unconditional branch back to just after the corresponding BEGIN . See WHILE and BEGIN .

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

## REPLACE\$

Stack-based String Library [42]
(--- pos|-1) ( SS: str str $_{2}$ str $r_{3}--s t r_{4} \mid\left[\right.$ str $r_{1}$ str $\left.r_{2}\right]$ "replace string"
The word REPLACE\$ searches string str $_{2}$ for the first occurrence of string $s t r_{3}$. If it is found, it is replaced with the string $s t r_{1}$, the position of $s t r_{3}$ within $s t r_{2}$ is pushed to the data stack, $s t r_{1}$ and $s t r_{3}$ are removed from the string stack and the new string $s t r_{4}$ is left on the string stack. For example,


If the search string $s t r_{3}$ is not found, -1 is pushed to the data stack, $s t r_{1}$ and $s t r_{2}$ 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 $r_{1}--s t r_{2}$ ) "reverse string"
The word REV\$ replaces the topmost string on the string stack with its reversed equivalent. For example,
\$" green" REV\$ .\$ neerg ok:0
RIGHT\$
Stack-based String Library [42]
(len --- ) ( SS: str ${ }_{1}--$ str $_{1} s t r_{2}$ ) "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.
RLTV
Resident
( --- )
Assigns the attribute RELATIVE to the file whose PAB is pointed to by PAB-ADDR .

RND
$\left(n_{1}--n_{2}\right)$
Generates a positive random integer $n_{2}$ greater than or equal to 0 and less than $n_{1}$.
(--- $n$ )
Generates a random word. The value of the word may be positive or negative depending on whether the sign bit is set.
([n] ... [0] $+n---[n-1] \ldots[0][n])$
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 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.
Resident
$\left(\begin{array}{llllll}n_{1} & n_{2} & n_{3} & -- & n_{2} & n_{3}\end{array} n_{1}\right)$
Rotate the top three values on the stack, bringing the third $n_{1}$ to the top.
( --- ) ( SS: $s t r_{1} \operatorname{str}_{2}$ str $_{3}---\operatorname{str}_{2}$ 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. See Chapter 14 for implementation details regarding stack space limitations.
( --- )
A procedure to initialize the return stack pointer from user variable R0.
( --- addr )
Returns the address $a d d r$ of the current top of the return stack.
( $n$--- )
Restores the file whose PAB is pointed to by the current PAB to the specified record number $n$.

RTRIM\$
Stack-based String Library [42]
( --- ) ( SS: str ${ }_{1}--s t r_{2}$ ) "trim right of string"
The word RTRIM\$ removes leading spaces from the topmost string. For example,

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

ASCII Collating Sequence: ! " \# \$ \% \& ( ) * + , . / digits : ; < = > ? @ HLPHA [ \ ] ^ _`alpha \{ | \} ~
( --- addr|[]) (IS: string")
Accepts a string from the input stream (IS) until '"' is encountered. When executing, the packed string is stored at PAD and the address $a d d r$ of the length byte is left on the stack.

When compiling a word definition, SLIT is first compiled into the definition, then the packed string. Later, when the word is executed, SLIT will push the address of the string's length byte to the stack and skip over the string to the word following it in the definition.
S->D Resident
( $n--d$ )
Sign-extend a single number $n$ to form a double number $d$.
( $n---f$ )
Converts a single-precision number $n$ on the stack to a floating point number $f$.
( --- addr )
User variable that points to the base of the parameter stack. Pronounced "s zero". See SP! .

S0\&TIB!
Resident
( $a d d r_{1}---a d d r_{2}$ )
This word is primarily for use in a 1024 KiB SAMS environment, where it is or may be necessary to move the stack base and TIB buffer, both of which start up at the same address, viz., FFA0h. S0\&TIB! forces $a d d r_{1}$ to AFA0h, BFA0h, CFA0h, DFA0h, EFA0h or FFA0h; copies it to the user variables, S0 and TIB, in the table of default values so the settings will survive COLD ; and leaves the new address on the stack as $a d d r_{2}$. The lower limit is forced above HERE so as not to destroy the user's dictionary.

Resident
( --- )
This word is ported from TurboForth code courtesy of Mark Wills.
This calls the SAMS initialization in the startup code in bank 1 to restore SAMS mapping to initial conditions.
SAMS?
Resident
( --- flag )
This word is ported from TurboForth code courtesy of Mark Wills.
Leaves a copy of the SAMS flag from startup as flag.

ASCII Collating Sequence: ! " \# \$ \% \& ( ) * + , . / digits : ; < = > ? @ HLPHA [ \ ] ^ _` alpha \{ | \}~
( --- vaddr )
A constant whose value vaddr is the VDP address of the Sprite Attribute List. Default value is $\mathbf{3 0 0 h}$. This constant can only be changed via user variable number 24h.
(addr n --- )
This word is ported from TurboForth code courtesy of Mark Wills.
SAY needs on the stack the address $a d d r$ of a block of Speech Synthesizer ROM speech addresses and the number $n$ of those addresses. This can be accomplished with DATA[ ...]DATA . Consult Section 22 of the Editor/Assembler Manual for details.
( addr --- )
This word expects to find on the stack the CRU address $a d d r$ 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, SB0 0, to effect setting the bit. See § 11.3 and CRU documentation in the Editor/Assembler Manual for more information.
( addr --- )
This word expects to find on the stack the CRU address $a d d r$ 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. See § 11.3 and CRU documentation in the Editor/Assembler Manual for more information.

CPYBLK -- Block Copying Utility [4]
( str $_{1}$ str $r_{2}---1|0|+1$ )
Compares two strings with leading byte counts pointed to by $s t r_{1}$ and $s t r_{2}$ and leaves the result on the stack: -1 , if $s t r_{1}<s t r_{2} ; 0$, if $s t r_{1}=s t r_{2} ;+1$, if $s t r_{1}>s t r_{2}$.
( --- addr )
A user variable containing the Forth block number most recently referenced by LIST or EDIT .

SCRFNT
( --- addr )
A user variable containing a flag indicating whether FNT should load the current default font $($ flag $\neq 0)$ or the console font (flag $=0$ ). Changing the value in SCRFNT does not take effect until the next time FNT is executed.
See Chapter 13 "Screen Fonts and the Font Editor" for more detail.

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

( $n$--- )
Changes the display screen color to the color specified $n$. The foreground (FG) and background (BG) screen colors must be placed in the low-order byte of $n$, with FG the high-order 4 bits and BG the low-order 4 bits, e.g., $n=27$ (1Bh) for black on light yellow. The FG color is only necessary in the text modes.

## SCRN_END

Resident
( --- addr )
A user variable containing the address $a d d r$ of the byte immediately following the last byte of the display screen image table to be used as the logical display screen.

## SCRN_START

Resident
( --- addr )
A user variable containing the address $a d d r$ of the first byte of the display screen image table to be used as the logical display screen.
SCRN_WIDTH Resident
( --- addr )
A user variable which contains the number of characters ( 32 or 40 ) that will fit across the display screen. Used by the display screen scroller.
SCROLL
Resident
( dir --- )
Scrolls the display screen panel set up by PANEL in direction dir. PANEL must be executed at least once before SCROLL because its parameters are indeterminate after powerup. Acceptable values for dir are

| Direction | Value |
| :--- | :---: |
| left | 0 |
| right | 2 |
| up | 4 |
| down | 6 |

SEED
Resident
( $n$--- )
Places a new seed $n$ into the random number generator.

## SET-PAB

Resident
( --- )
This instruction assumes that PAB-ADDR is set. It then zeroes out the PAB (Peripheral Access Block) pointed to by PAB-ADDR and places the contents of PABVBUF into the appropriate word of the PAB. This initializes the PAB.

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

$$
(n---1|0|+1)
$$

Returns the sign of $n$ or 0 .
( $n d---d$ )
Stores a minus sign (ASCII 45 or 2 Dh ) 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 ... \#> .
$\left(f_{1}---f_{2}\right)$
Finds the sine $f_{2}$ of the floating point number $f_{1}$ on the stack and leaves the result $f_{2}$ on the stack.
SLA
Resident
( $n_{1}$ count --- $n_{2}$ )
Arithmetically shifts the number $n_{1}$ on the stack count bits to the left, leaving the result $n_{2}$ on the stack. Shifting by count will be modulo 16 except when count $=0$, which causes 16 bits to be shifted. To create a word which does not perform a 16-bit shift when count is zero, use the following definition for the same stack contents:

```
: SLA0 -DUP IF SLA ENDIF ;
```

SLIT
Resident

## ( --- addr )

SLIT is similar to LIT but acts on strings instead of numbers. SLIT places the address $a d d r$ of the string following it on the stack. It modifies the top of the return stack to point to just after the string.
( $d n$--- rem quot )
A mixed magnitude math operator that performs symmetric division to leave the signed remainder rem and signed quotient quot from a double-number dividend $d$ and single-number divisor $n$. The quotient is rounded toward zero and the remainder given the sign of the dividend. See Chapter 18 "Signed Integer Division" for more details.
( addr $_{1}$ count $_{1} n$--- addr $r_{2}$ vaddr count ${ }_{2}$ )
The assembly code routine that formats a line of tiny characters. It expects the address $a d d r_{1}$ of the line in memory, the number count $_{1}$ of characters per line, and the line number $n$ to which it is to be written. It returns on the stack the line buffer address addr $_{2}$, a VDP address vaddr, and a byte count count ${ }_{2}$. See CLIST and CLINE .
( --- vaddr )
A constant whose value is the VDP address of the Sprite Motion Table. Default value is 780 h . This constant can only be changed via user variable number 26 h .
SMUDGE
( --- )
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. SMUDGE is simply defined as

HEX : SMUDGE LATEST 20 TOGGLE ;
SOUND
Resident
( pitch vol ch\# --- )
This word is ported from TurboForth code courtesy of Mark Wills.
Pitch pitch, volume vol and channel ch\# are as described in the Editor/Assembler Manual in Section 20. Pitch values range from $0-1023,0$ representing the highest pitch. Volume values range from $0-15,15$ representing silence. Channels $0-2$ represent the corresponding tone generators and channel 3 is the noise generator.
SOUND uses the pitch value for setting the type of noise for the noise generator (channel 3). Shift rates are $0-3$. Noise type can be white noise (0) or periodic noise (4). The pitch value to pass to SOUND is the sum of shift rate and noise type and ranges from $0-7$.
Once a tone or noise generator is started, the sound/noise continues until silenced by executing SOUND with a volume of 15 . The pitch must be supplied, but is irrelevant. The following Forth code will silence channel 2 :

## 0152 SOUND

SP!
Resident
( --- )
A procedure to initialize the parameter stack pointer from S0, the user variable that points to the base of the parameter stack.

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

SP@
Resident
( --- addr )
This word returns the address of the top of the stack as it was before SP@ was executed, e.g.,

12 SP@ @ . . . $\underline{2} \underline{2} \underline{1}$ ok:0
SPACE
( --- )
Transmit a blank character (ASCII 32|20h) to the output device.
SPACES
Resident
( $n---$ )
Transmit $n$ blank characters (ASCII 32|20h) to the output device.
SPCHAR

( $n_{1} n_{2} n_{3} n_{4}$ char --- )
Defines a character char in the Sprite Descriptor Table to have the pattern composed of the 4 words (cells) on the stack.

SPDCHAR
Resident
( addr cnt chr --- )
Same as DCHAR, but for sprite pattern definitions because SPDTAB does not always start at the same VRAM address as PDT.

SPDTAB
Resident
( --- vaddr )
A constant whose value is the VDP address of the Sprite Descriptor Table. Default value is $\mathbf{8 0 0 h}$. Notice that this coincides with the Pattern Descriptor Table. This constant can only be changed via user variable number 42h.
SPLIT
( --- )
Converts from present display screen mode into standard Split mode configuration.
SPLIT2
Resident
( --- )
Converts from present display screen mode into standard Split2 mode configuration.
SPRCOL
Resident
( $n \operatorname{spr}---$ )
Changes color of the given sprite number $s p r$ to the color $n$ specified.
SPRDIST
Resident
( $s p r_{1} s p r_{2}--n$ )
Returns on the stack the square of the distance $n$ between two specified sprites, $s p r_{1}$

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

and $s p r_{2}$. Distance is measured in pixels and the maximum distance that can be detected accurately is 181 pixels.

## SPRDISTXY

Resident
( dotcol dotrow spr --- n )
Places on the stack $n$, the square of the distance between the point (dotcol,dotrow) and a given sprite spr. Distance is measured in pixels and the maximum distance that can be detected accurately is 181 pixels.
( spr --- dotcol dotrow)
Returns the dot column dotcol and dot row dotrow position of sprite spr.

## SPRITE

Resident
(dotcol dotrow n char spr --- )
Defines sprite number spr to have the specified location (dotcol,dotrow), color $n$, and character pattern char. The size of the sprite will depend on the magnification factor.
SPRPAT
Resident
( char spr --- )
Changes the character pattern of a given sprite $s p r$ to char.
SPRPUT
Resident
( dotcol dotrow spr --- )
Places a given sprite spr at location (dotcol,dotrow).
SQNTL
Resident
( --- )
Assigns the attribute SEQUENTIAL to the file whose PAB is pointed to by PABADDR .

Resident
( $f_{1}---f_{2}$ )
Finds the square root of a floating point number $f_{1}$ and leaves the result $f_{2}$ on the stack.

SRA
Resident
( $n_{1}$ count --- $n_{2}$ )
Arithmetically shifts $n_{1}$ count bits to the right and leaves the result $n_{2}$ on the stack. Shifting by count will be modulo 16 except when count $=0$, which causes 16 bits to be shifted. To create a word which does not perform a 16 -bit shift when count is zero, use the following definition for the same stack contents:

```
: SRA0 -DUP IF SRA ENDIF ;
```

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

SRC
( $n_{1}$ count --- $n_{2}$ )
Performs a circular right shift of count bits on $n_{1}$ leaving the result $n_{2}$ on the stack. If count is 0,16 bits are shifted. To create a word which does not perform a 16-bit shift when count is zero, use the following definition for the same stack contents:

```
: SRC0 -DUP IF SRC ENDIF ;
```

( $n_{1}$ count $---n_{2}$ )
Performs a logical right shift of count bits on $n_{1}$ and leaves the result $n_{2}$ on the stack. If count is 0,16 bits are shifted. To create a word which does not perform a 16 -bit shift when count is zero, use the following definition for the same stack contents:

## : SRL0 -DUP IF SRL ENDIF ;

( vaddr --- )
No longer initializes sprites! Use DELALL, q.v., instead. SSDT places the Sprite Descriptor Table at the specified VDP address vaddr and initializes all sprite tables. The address given must be on an even 2 K boundary. See § 6.6.2 "Sprite Initialization" for details.
(---b)
Reads the status of the current PAB and returns the status byte $b$ to the stack. See the table in § 8.5 following the explanation of STAT for the meaning of each bit of the status byte.
( --- addr )
A user variable containing the compilation state. Zero indicates execution and a nonzero value indicates compilation. The compilation-state value for fbForth 2.0 (inherited from TI Forth) is C 0 h . The reason for this value is that the length byte of a found word, which is also immediate, has the high-order two bits set (see Chapter 12, "fbForth 2.0 Dictionary Entry Structure" for details). INTERPRET compares the value of STATE with the length byte to decide whether to execute a word during compilation.
STCR
CRU Words [5]
( $n_{1}$ addr --- $n_{2}$ )
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 0 Fh. If

[^4]$n_{1}=0,16$ 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 $\S 11.3$ and CRU documentation in the Editor/Assembler Manual for more information.
STREAM
Resident
( addr $n$--- )
This word is ported from TurboForth code courtesy of Mark Wills.
STREAM needs on the stack the address $a d d r$ of a block of raw speech data to be spoken and the number of cells $n$ in the buffer. This can be accomplished with DATA [ ... ]DATA . STREAM will feed the raw speech data to the Speech Synthesizer.
( count --- )
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
$\left(n_{1} n_{2}--n_{2} n_{1}\right)$
Exchange the top two values on the stack.
SWAP\$
( --- ) ( SS: str ${ }_{1}$ str $r_{2}--s t r_{2}$ 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.
( --- )
A special purpose word which permits EMIT to output characters to an RS232 device rather than to the screen. See UNSWCH .
$\left(n_{1}--n_{2}\right)$
Reverses the order of the two bytes in $n_{1}$ and leaves the new number as $n_{2}$.
SYS\$
Resident
( --- addr )
A user variable that contains the address of the system support entry point.

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

SYSTEM
( $n$--- )
Calls the system synonyms. You must specify an offset $n$ into a jump table for the routine you wish to call. The offset $n$ must be one of the predefined even numbers. See system Forth block 33 for offsets $0-26$.
( --- addr ) "s or f"
User variable that determines whether M/ uses symmetric or floored integer division. A value of zero (the default) specifies symmetric integer division and a non-zero value, floored integer division. See Chapter 18 "Signed Integer Division" for more details.
TALKING?
Resident
( --- flag )
This word is ported from TurboForth code courtesy of Mark Wills.
TALKING? returns flag $=0$ if the Speech Synthesizer is idle, otherwise, flag $=1$.
It is a good idea to use TALKING? to insure the Speech Synthesizer is not busy before executing SAY or STREAM .
$\left(f_{1}---f_{2}\right)$
Finds the tangent of the floating point number ( $f_{1}=$ angle in radians) on the stack and leaves the result $f_{2}$.
TASK
Resident
( --- )
A no-operation word or null definition, TASK is the last word defined in the resident Forth vocabulary of fbForth $\mathbf{2 . 0}$ 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 (see Chapter 11): ' TASK 21 BSAVE (Be sure to back up the original disk before trying this!). By redefining TASK at the beginning of an application, you can mark the boundary between applications. By FORGETting TASK and re-compiling, an application can be discarded in its entirety. You will be able to FORGET each instance of the definition of TASK except the first one described above.
тв
CRU Words [5]
( addr --- flag )
TB performs the TMS9900 TB instruction. The bit at CRU address $a d d r$ is tested by this instruction. Its value (flag $=1 \mid 0)$ is returned to the stack. The CRU base address $a d d r$ 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. See § 11.3 and CRU documentation in the Editor/Assembler Manual for more information.
( --- addr )
Points to the array that holds the tiny character definitions for the 64 -column editor. See CLIST .
TEXT
( --- )
Converts from present display screen mode into standard Text mode configuration.
TEXT80
( --- )
Converts from present display screen mode into Text80 mode configuration if your computer has that facility.

THEN
[immediate word]
Resident
( --- )
An alias for ENDIF .
( --- addr )
A user variable containing the address of the terminal input buffer.
TIF2FBF
TI Forth Block Utilities
( --- ) ( IS: srcStrtBlk srcEndBlk DSKn dstStrtBlk dstFile )
Copies the range of blocks (screens) srcStrtBlk - srcEndBlk from TI Forth disk DSKn to fbForth blocks file dstFile, starting at block dstStrtBlk.
TIFBLK
TI Forth Block Utilities
( ---) (IS: blk DSKn)
Lists block (screen) blk of TI Forth disk DSKn to the display. The display will pause for user intervention in Text mode due to wrapping 64-byte lines on a 40 -column display.
TIFIDX
( --- ) (IS: strtBlk endBlk DSKn )
Lists the index (line \#0) lines of a range of blocks (screens) strtBlk - endBlk of TI Forth disk $D S K n$ to the display. The display will pause for user intervention if the list requires scrolling.
TIFVU
TI Forth Block Utilities
( IS: blk DSKn )
Starts the TI Forth disk browser/copier at block (screen) blk of TI Forth disk DSKn. The browser is patterned after the fbForth block editors, allowing scrolling left and right by panels and blocks. The user may also copy a range of TI Forth blocks to an fbForth blocks file, which must have been created prior to entering the browser/copier.

ASCII Collating Sequence: ! " \# \$ \% \& ( ) * + , . / digits : ; < = > ? @ HLPHA [ \ ] ^ _` alpha \{ | \}~
( addr b--- )
Complement (XOR) the contents of the byte at $a d d r$ by the bit pattern of byte $b$.
( delim --- addr | [ ] ) ( IS: string<delim> )
TOKEN gets a string ending with delim from the input stream (IS) into PAD as a packed string and passes the address addr of the string's length byte on the stack if interpreting (command line or loading), but compiles the packed string to HERE , with nothing to the stack, if compiling.
TOKEN is used by several words in the resident dictionary, including MKBFL, USEBFL, $\mathrm{S}^{\prime \prime}$, ." , WLITERAL and USEFFL .

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 TRACE option, it may be executed with or without a trace. To implement a trace, type TRON before execution. To execute without a trace, type TROFF . 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. See TRON , TROFF , UNTRACE and § 5.4 .
TRAVERSE
Resident
( $a d d r_{1} n---a d d r_{2}$ )
Traverse the name field of a fig-Forth variable-length name field. The starting point $a d d r_{1}$ is the address of either the length byte or the last letter. If $n=1$, the direction is toward high memory; if $n=-1$, the direction is toward low memory. The resulting address $a d d r_{2}$ points to the other end of the name.

TRIAD
Printing Routines [19]
(blk --- )
Display on the RS232 device the three Forth blocks that include block number blk, 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/figForth extension".
TRIADS
Printing Routines [19]
( $b l k_{1} b l k_{2}---$ )
May be thought of as a multiple TRIAD , q.v. You must specify a Forth block range. TRIADS will execute TRIAD as many times as necessary to cover that range.
TRIM\$
Stack-based String Library [42]
( --- ) ( SS: str - --- str $r_{2}$ ) "trim string"
The word TRIM\$ removes both leading and trailing spaces from the topmost string. For example,

[^5]```
$" hello! " TRIM$ .$ hello! ok:0
```

TROFF
TRACE -- Colon Definition Tracing [18]
( --- )
Turn off tracing of words compiled with the TRACE option. See TRON, TRACE, UNTRACE and § 5.4 .
( --- )
Turn on tracing of words compiled with the TRACE option. See TROFF, TRACE, UNTRACE and § 5.4 .

TRUNC
TRACE -- Colon Definition Tracing [18]
[ex$\left(f_{1}---f_{2}\right)$

Truncates $f_{1}$, leaving the integer portion $f_{2}$ on the stack.
$\left(n_{1} n_{2}---n_{2} n_{1} n_{2}\right)$
Put a copy of the top number under the top two numbers on the stack.
TYPE
Resident
( addr count --- )
Transmit count characters from addr to the selected output device.
U
Resident
(---n)
Places the contents $n$ of workspace register UP (R8) on the stack. Register U contains the base address of the user variable area. This is quicker than executing U0 @ , which accomplishes the same thing.
U* $\left(u_{1} u_{2}---u d\right) \quad$ Resident ( $u_{1} u_{2}---u d$ )
Leave the unsigned double number product $u d$ of two unsigned numbers, $u_{1}$ and $u_{2}$.
U.
( $u$--- )
Prints an unsigned number $u$ to the output device.
U.R

Resident
( $u n---$ )
Prints an unsigned number $u$ right justified in a field of width $n$.
U/
Resident
( ud u--- urem uquot)
Leave the unsigned remainder urem and unsigned quotient uquot from the unsigned double dividend $u d$ and unsigned divisor $u$.

ASCII Collating Sequence: ! " \# \$ \% \& ( ) * + , . / digits : ; < = > ? @ ALPHA [ \ ] ^ _`alpha \{ | \}~
( --- addr )
A user variable that points to the base of the user variable area.
$\mathrm{U}<$
Resident
( $u_{1} u_{2}--$ - flag )
Leaves a true flag if $u_{1}$ is less than $u_{2}$, else leaves a false flag.
UCASE\$
Stack-based String Library [42]
( --- ) (SS: str $r_{1}--$ str $_{2}$ ) "convert to upper case"
The word UCASE\$ converts all lower case characters in the topmost string to upper case. For example,
( --- addr )
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 COLD start.

UD.
Resident
( ud --- )
Prints an unsigned double number $u d$ to the output device.
UD.R
Resident
( ud n--- )
Prints an unsigned double number $u d$ right justified in a field of length $n$.
UM/MOD
( ud u--- urem uquot)
See U/ . This word is not in fbForth 2.0, but is identical to $\mathbf{U} /$ and is referenced here because of the inclusion of ANS Forth words SM/REM and FM/MOD , q.v.
UNDRAW
Resident
( --- )
Sets DMODE to 1. This means that dots are plotted in the off mode.
UNFORGETABLE [sic] Resident
( addr --- flag )
Decides whether or not a word can be forgotten. A true flag is returned if the address is not located between FENCE and HERE. Otherwise, a false flag is left. See FORGET. 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 (TASK ) in the core dictionary such that UNFORGETABLE [sic] will report false; however, FORGET will still trap that error.

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

( --- )
Causes the computer to send output to the display screen instead of an RS232 device. See SWCH

UNTIL
[immediate word]
Resident
Occurs within a colon-definition in the form:
BEGIN ... UNTIL
Compile time: (addr 1 --- )
UNTIL compiles (0BRANCH) and an offset from HERE to $a d d r$, which it stores at the space reserved for it at $a d d r$ 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
( --- )
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
[immediate word]
Resident
( --- ) (IS: DSKn.<blocks file> )
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

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

USEFFL
[immediate word]
Resident
( --- ) (IS: DSKn.<font file>)
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
( $n$--- )
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. g900 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
```

ASCII Collating Sequence: ! " \# \$ \% \& ( ) * + , . / digits : ; < = > ? @ ALPHA [ \ ] ^ _`alpha \{ | \} ~
( 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
( \(n\)--- ) (IS: <new name> )
A defining word used in the form:

\section*{\(n\) 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.

\section*{VCHAR}

Resident
( col row count char --- )
Prints on the display screen a vertical stream of length count of the specified character char. The first character of the stream is located at (col,row). Rows and columns are numbered from 0 beginning at the upper left of the display screen. VCHAR does not check to see whether (col,row) is within the screen buffer. Upon reaching the end of the screen buffer, it wraps to the top of the same column. This is different from TI Forth, which wraps to the next column and then to \((0,0)\), filling the screen buffer if count is high enough. This behavior will be changed in the next build of fbForth 2.0 to conform to how TI Basic and TI Extended Basic implement this function, i.e., in the next build, VCHAR will throw an error if it would start outside the screen buffer and it will wrap to \((0,0)\) upon reaching the end of the screen buffer, as it does now.
VDPMDE
Resident
( --- addr )
A user variable used by the mode changing words TEXT80, TEXT, GRAPHICS, MULTI, GRAPHICS2, SPLIT and SPLIT2 to hold \(0-6\), respectively. VMODE, q.v., also changes VDPMDE .
VFILL Resident
( vaddr count b---)
Fills count locations beginning at the given VDP address vaddr with the specified byte \(b\).
VLIST
Resident
( --- )
Prints the names of all words defined in the CONTEXT vocabulary. Note that VLIST will display the names of even ill-defined words in the dictionary that cannot be found with ', -FIND or (FIND), q.v., because their smudge bits are set. See SMUDGE and PAUSE .
( vaddr addr count --- )
Reads count bytes beginning at the given VDP address vaddr and places them at addr.
( addr vaddr count --- )
Writes count bytes from addr into VDP beginning at the given VDP address vaddr.
( \(n\)--- )
Changes the VDP mode to mode \(n\), corresponding to the values shown in the entry for VDPMDE above.

VMOVE
Resident
( vaddr \(_{1}\) vaddr \(_{2} n---\) )
Move a block of \(n\) bytes of VRAM from \(v a d d r_{1}\) to \(v a d d r_{2}\), all in VRAM, proceeding toward high memory. This is not overlap safe for vaddr \(_{1}<\) vaddr \(_{2}\).
VOC-LINK
Resident
( --- addr )
A user variable containing the address of a field in the definition of the most recently created vocabulary. All vocabulary names are linked by these fields to allow control for forgetting with FORGET through multiple vocabularies.
VOCABULARY
Resident
( --- )
A defining word used in the form:

\section*{VOCABULARY cccc}
to create a vocabulary definition cccc. Subsequent use of cccc will make it the CONTEXT vocabulary which is searched first by INTERPRET. The sequence cccc DEFINITIONS will also make cccc the CURRENT vocabulary into which new definitions are placed.
cccc will be so chained as to include all definitions of the vocabulary in which cccc is itself defined. All vocabularies ultimately chain to Forth. By convention, vocabulary names are to be declared IMMEDIATE . See VOC-LINK .
(b vaddr --- )
Performs a logical OR on the byte at the specified VDP address and the given byte \(b\). The result byte is stored back into the VDP address.
VRBL
( --- )
Assigns the attribute VARIABLE to the file whose PAB is pointed to by PAB-ADDR .
(vaddr --- b)
Reads a single byte from the given VDP address vaddr and places its value \(b\) on the stack.
( b vaddr --- )
Writes a single byte \(b\) into the given VDP address vaddr.
(bn---)
Writes the given byte \(b\) into the specified VDP write-only register \(n\).
( b vaddr --- )
Performs a logical XOR on the byte at the specified VDP address vaddr and the given byte \(b\). The result byte is stored back into the VDP address vaddr.

\section*{WARNING}

Resident
( --- addr )
A user variable (initialized by COLD to 1 at system startup), containing a value controlling messages.

If WARNING \(>0\), full-text system error messages are displayed by MESSAGE and ERROR, which executes MESSAGE .
If WARNING \(=0\), messages will be presented by number ( \(\boldsymbol{m s g} \boldsymbol{\# n}\) ). 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.
If WARNING \(<0\) when ERROR executes, ERROR will execute (ABORT), which can be redefined to execute a user-specified procedure instead of the default ABORT .
See MESSAGE , (ABORT), ERROR and ?ERROR for more detail.
WBLK
( addr blk --- )
Write a block to the current blocks file.
WHERE
Resident
( \(n_{1} n_{2}---\) )
When an error occurs on a LOAD instruction, typing WHERE will bring you into the 40 column editor and place the cursor at the exact location of the error. WHERE consumes the two numbers, \(n_{1}\) and \(n_{2}\), left on the stack by the LOAD error.

WHERE (EDITOR2 Vocabulary)
( \(n_{1} n_{2}---\) )
When an error occurs on a LOAD instruction, typing WHERE will bring you into the 64column editor and place the cursor at the exact location of the error. WHERE consumes the two numbers, \(n_{1}\) and \(n_{2}\), left on the stack by the LOAD error.

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

WHILE
[immediate word]
Resident
Occurs in a colon-definition in the form:
BEGIN ... WHILE (true part) ... REPEAT
Compile time: ( \(a d d r_{1} 1---a d d r_{1} 1 a d d r_{2} 4\) )
WHILE emplaces (0BRANCH) and leaves \(a d d r_{2}\) of the reserved offset. The stack values will be resolved by REPEAT. The values 1 and 4 are used for error checking.
Runtime: (flag --- )
WHILE selects conditional execution based on flag. If flag is true (non-zero), WHILE continues execution of the true part through to REPEAT, which then branches back to BEGIN. If flag is false (zero), execution skips to just after REPEAT, exiting the structure.
WIDTH
Resident
( --- addr )
A user variable containing the maximum number of letters saved in the compilation of a definition's name. It must be \(1-31\), with a default value of 31 . The name character count and its natural characters are saved up to the value in WIDTH. The value may be changed at any time within the above limits.
WITHIN More Useful Stack Words etc. [41]
( \(n_{1} n_{2} n_{3}---\) flag )
Result flag is true (1) if \(n_{2} \leq n_{1}<n_{3}\) and false (0) otherwise.

\section*{WLITERAL}
[immediate word]
Resident
Compile time ( --- ) Runtime ( --- addr) Interpreting ( --- addr )
( IS: <space-delimited string> )
During compilation, WLITERAL compiles SLIT and the space-delimited string, which follows WLITERAL in the input stream, into the dictionary. At runtime, SLIT will push to the stack the address of the string's length byte and change IP to point to the Forth word following the string.
During execution, WLITERAL simply pushes to the stack the address of the string's length byte.

Used in the form: WLITERAL cccc
( char --- )
Read the text characters from the input stream being interpreted until a delimiter char is found, storing the packed character string (see footnote 4 on page 22) beginning at the dictionary buffer HERE. WORD leaves the character count in the first byte followed by the input characters and ends with two or more blanks. Leading occurrences of char are ignored. If BLK is zero, text is taken from the terminal input buffer, otherwise from the disk block stored in BLK. See BLK , IN .

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

\section*{( --- addr )}

A user variable containing the wrapping flag for SCROLL. A non-zero value signals SCROLL to wrap the disappearing row or column of the panel set up by PANEL to the opposite side of the panel. The initial value of WRAP is 0 .
WRT
( count --- )
Performs the file I/O write operation. You must specify the number of bytes count to be written.

XMLLNK
Resident
( addr --- )
Links a Forth program to a routine in ROM or to a routine located in the memory expansion unit. A ROM address addr or XML vector must be specified as in the Editor/Assembler Manual.
( \(n_{1} n_{2}--n_{3}\) )
Leave \(n_{3}\), the bitwise logical exclusive OR (XOR) of \(n_{1}\) and \(n_{2}\).
[immediate word]
Resident
( --- )
Used in a colon-definition in the form:
: xxxx [ words ] more ;
Suspend compilation. The words after [ are executed, not compiled. This allows calculation or compilation exceptions before resuming compilation with ]. See LITERAL and ] .

\section*{[COMPILE]}
[immediate word]
Resident
( --- )
Used in a colon definition in the form: : xxxx [COMPILE] FORTH ;
[COMPILE] will force the compilation of an immediate definition that would otherwise execute during compilation. The above example will select the Forth vocabulary when \(\mathbf{x x x x}\) executes rather than at compile time.
[DCHAR]
Resident
( addr cnt chr vaddr --- )
Helper routine for DCHAR and SPDCHAR.
\(\backslash\)
[immediate word]
Resident
( --- ) (IS: comment)
\(\backslash\) is used in the form:
\ ccoc

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

It starts a line comment that will not be compiled if it occurs in a definition. It causes the interpreter to ignore the rest of the line of the input stream (block or TIB). For blocks, a line is 64 characters, even though there are no actual terminator characters until the end of the block. \may occur during execution or in a colon definition. As with all Forth words, a blank after \(\backslash\) is required. This is most useful for commenting Forth source code in blocks.
( --- )
Resume compilation to the completion of a colon definition. See [ .
[immediate word]
( --- )
]DATA closes a DATA [ ... ]DATA construct that compiles numbers and leaves their beginning address and cell count on the stack. If compiling within another definition, ]DATA stores the cell count between the compiled DATA[] and the first number of the array.
\(\wedge\)
Resident
\(\left(f_{1} f_{2}---f_{3}\right)\)
Returns \(f_{3}\) on the stack as \(f_{1}\) raised to the \(f_{2}\) power. The operands must be floating point numbers.

\section*{Appendix E Differences: fbForth 2.0, fbForth 1.0 and TI Forth}

This appendix will detail fbForth \(\mathbf{2 . 0}\) changes from fbForth \(\mathbf{1 . 0}\) and TI Forth. This will include words that have been added, removed, re-purposed, deprecated and whose descriptions have changed (usually means "clarified"). All of those words, except those removed, will also be discussed elsewhere in the manual where appropriate, including the fbForth \(\mathbf{2 . 0}\) 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.

\section*{E. 1 TI Forth Words not in fbForth 2.0}

Descriptions of words appearing in the comments here that are part of fbForth \(\mathbf{2 . 0}\) may be found in Appendix D "The fbForth 2.0 Glossary".
```

!"
(!")
-64SUPPORT Now type MENU for options: 6 LOAD
-ASSEMBLER Now type MENU for options: 21 LOAD
-BSAVE Words loaded are now part of resident dictionary.
-CODE Words loaded are now part of resident dictionary.
-COPY CPYBLK replaces contents. Now type MENU for options: 4 LOAD
-CRU Now type MENU for options: 5 LOAD
-DUMP Now type MENU for options: 16 LOAD
-EDITOR Words loaded are now part of resident dictionary.
-FILE Words loaded are now part of resident dictionary.
-FLOAT Words loaded are now part of resident dictionary.
-GRAPH Words loaded are now part of resident dictionary.
-GRAPH1 Words loaded are now part of resident dictionary.
-GRAPH2 Words loaded are now part of resident dictionary.
-MULTI Words loaded are now part of resident dictionary.
-PRINT Now type MENU for options:19 LOAD
-SPLIT Words loaded are now part of resident dictionary.
-SYNONYMS Words loaded are now part of resident dictionary except FORMAT-DISK,
which has been removed.

```
ASCII Collating Sequence: ! " \# \$ \% \& ( ) * + , . / digits : ; < = > ? @ ALPHA [ \] ^ - `alpha \{ | \}~
\begin{tabular}{ll}
-TEXT & Words loaded are now part of resident dictionary. \\
-TRACE & Now type MENU for options: 18 LOAD \\
-VDPMODES & Words loaded are now part of resident dictionary. \\
>ARG & No longer a high-level Forth word \\
>FAC & No longer a high-level Forth word \\
ARG & No longer a high-level Forth word \\
B/BUF\$ & User variable no longer used. \\
B/SCR\$ & User variable no longer used. \\
CHAR-CNT! & No longer a high-level Forth word \\
CHAR-CNT@ & No longer a high-level Forth word \\
CHK-STAT & No longer a high-level Forth word \\
CLR-STAT & No longer a high-level Forth word \\
DDOT & No longer a high-level Forth word \\
DISK-HEAD & \\
DISK_HI & User variable no longer used. \\
DISK_LO & User variable no longer used. \\
DISK_SIZE & User variable no longer used. \\
DR0 & \\
DR1 & \\
DR2 & \\
DRIVE & \\
DTEST & \\
EDITOR1 & No EDITOR1 vocabulary any longer \\
F.R & Use F. or FFMT . to compose a replacement definition if needed. \\
FAC & No longer a high-level Forth word \\
FAC->S & No longer a high-level Forth word \\
FAC> & No longer a high-level Forth word \\
FAC>ARG & No longer a high-level Forth word \\
FADD & No longer a high-level Forth word \\
FDIV & No longer a high-level Forth word \\
FF. & Use FFMT . \\
FF.R & Use FFMT . to compose a replacement definition if needed. \\
\hline
\end{tabular}

ASCII Collating Sequence: ! " \# \$ \% \& ( ) * + , . / digits : ; < = > ? @ ALPHA [ \ ] ^ _` alpha \{ | \} ~
\begin{tabular}{ll} 
FLD & Unused user variable removed. \\
FMUL & No longer a high-level Forth word \\
FORMAT-DISK & \\
FORTH-COPY & \\
FORTH_LINK & User variable no longer used. Its function is part of FORTH (Forth vocabulary \\
& \begin{tabular}{l} 
declaration word).
\end{tabular} \\
FSUB & No longer a high-level Forth word \\
GET-FLAG & No longer a high-level Forth word \\
OFFSET & User variable no longer used. \\
PUT-FLAG & No longer a high-level Forth word \\
RDISK & Replaced by RBLK. \\
S->FAC & No longer a high-level Forth word \\
SCOPY & Replaced by CPYBLK. \\
SCRTCH & Never should have been implemented! \\
SETFL & No longer a high-level Forth word \\
SMOVE & Replaced by CPYBLK. \\
STR & No longer a high-level Forth word \\
STR. & No longer a high-level Forth word \\
VAL & No longer a high-level Forth word \\
WDISK & Replaced by WBLK.
\end{tabular}

\section*{E. 2 fbForth 1.0 Words not in fbForth 2.0}

Descriptions of words appearing in the comments here that are part of fbForth \(\mathbf{2 . 0}\) may be found in Appendix D "The fbForth 2.0 Glossary".
\begin{tabular}{ll}
\hline >ARG & No longer a high-level Forth word \\
>FAC & No longer a high-level Forth word \\
>ROA & \begin{tabular}{l} 
No longer needed because the GPL/XML floating point routines that \\
modified the rollout area have been replaced.
\end{tabular} \\
ARG & No longer a high-level Forth word \\
BPOFF & No longer needed \\
CHAR-CNT! & No longer a high-level Forth word \\
CHAR-CNT@ & No longer a high-level Forth word \\
CHK-STAT & No longer a high-level Forth word \\
CLR-STAT & No longer a high-level Forth word \\
DBF & No longer needed \\
DDOT & No longer a high-level Forth word \\
EDITOR1 & No EDITOR1 vocabulary any longer \\
F.R & Use F. or FFMT . to compose a replacement definition if needed. \\
FAC & No longer a high-level Forth word \\
FAC->S & No longer a high-level Forth word \\
FAC> & No longer a high-level Forth word \\
FAC>ARG & No longer a high-level Forth word \\
FADD & No longer a high-level Forth word \\
FDIV & No longer a high-level Forth word \\
FF. & Use FFMT . \\
FF.R & Use FFMT . to compose a replacement definition if needed. \\
FLD & Unused user variable removed. \\
FMUL & No longer a high-level Forth word \\
FSUB & No longer a high-level Forth word \\
GET-FLAG & No longer a high-level Forth word \\
LCT & No longer needed
\end{tabular}

\footnotetext{
ASCII Collating Sequence: ! " \# \$ \% \& ( ) * + , . / digits : ; < = > ? @ ALPHA [ \ ] ^ _`alpha \{ | \} ~
}
\begin{tabular}{ll} 
MGT & No longer needed \\
PUT-FLAG & \begin{tabular}{l} 
No longer a high-level Forth word \\
No longer needed because the GPL/XML floating point routines that \\
modified the rollout area have been replaced.
\end{tabular} \\
ROA & \begin{tabular}{l} 
No longer needed because the GPL/XML floating point routines that \\
modified the rollout area have been replaced.
\end{tabular} \\
ROA> & No longer a high-level Forth word
\end{tabular}

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

\section*{E. 3 New and Modified Words in fbForth 2.0}

This list contains all the new words added since fbForth \(\mathbf{1 . 0}\) 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 or their definitions and/or descriptions have changed. Detailed descriptions of words listed here may be found in Appendix D "The fbForth 2.0 Glossary".

\section*{\#MOTION}
(ABORT)
*/
*/MOD
.BASE Display current radix in decimal.
/
/MOD
0> Leaves true flag if number on stack is less than 0.
; CODE Now, also terminates CODE: .
>DEG
Converts number on stack from radians to degrees.
\(>F\)
>MAP Map SAMS memory.
>RAD
Converts number on stack from degrees to radians.
?FLERR
ALIGN Insures that HERE is on an even address boundary.
APPND
ASCII Pushes to the stack the ASCII value of the next character in the input stream.
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
BOOT
BSAVE
CASE

\begin{tabular}{ll}
\hline CEIL & Returns the floating point (FP) integer closest to but \(>\) the number on the stack. \\
\hline CF? & Returns a flag indicating whether a nanoPEB or CF7+ is attached. \\
\hline CFMOUNT & Mounts a CF volume in virtual DSK1, DSK2 or DSK3. \\
\hline CFVOLS & Returns volumes mounted in virtual DSK1, DSK2 and DSK3. \\
CHAR & \\
CHARPAT & \\
CLSE & \\
\hline CODE: & \\
\hline COINC & \\
COINCALL & \\
COINCXY & \\
COLD & \\
COLOR & \\
COLTAB & \\
COS & \\
\hline Can only change this constant via user variable number 22h.
\end{tabular}

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

\section*{ELSEOF Catchall default OF for CASE .}

ENDCASE
ERROR
EULER_E FP constant for \(e\).
EXP
EXP10 Returns 10 raised to the power of the FP number on the stack.
F!
F*
F+
F-
F->S
F-D"
F.

F/
F0<
F0=
F<
F=
F>
F>R Transfers the FP number on the stack to the return stack.
F@
FABS
Returns absolute value of FP number on stack.
FCONSTANT
Defines an FP constant.
FDROP
FDUP
FFMT.
Formats and displays/prints an FP number on the stack.
FILE
FLERR

FLOOR
FM/MOD
FMINUS
FNT

Returns the floating point (FP) integer closest to but < the number on the stack.
M/ with floored integer division.
Negates the FP number on the stack.
Loads the current font (default in cartridge ROM or user-specified).
\begin{tabular}{|c|c|}
\hline FONTED & Starts the new Font Editor. \\
\hline FOVER & \\
\hline FP1 & FP constant for 1. \\
\hline FP10 & FP constant for 10. \\
\hline FPB & Pushes VRAM address of user screen font file PAB to stack. \\
\hline FRAC & Returns fractional part of FP number on the stack. \\
\hline FROT & Rotates the third FP number on the stack to the top of the stack. \\
\hline FSWAP & \\
\hline FVARIABLE & Defines an FP variable. \\
\hline FXD & \\
\hline GCHAR & \\
\hline GRAPHICS & \\
\hline GRAPHICS2 & \\
\hline HCHAR & \\
\hline HONK & \\
\hline INPT & \\
\hline INT & \\
\hline INTRNL & \\
\hline ISR & \\
\hline JCRU & \\
\hline JKBD & \\
\hline JMODE & \\
\hline JOYST & \\
\hline LD & \\
\hline LINE & \\
\hline LN10INV & FP constant for \(1 / \ln (10)\) \\
\hline LOG & \\
\hline LOG10 & Returns the decimal logarithm of the FP number on the stack. \\
\hline M/ & Now, does either symmetric (default) or floored integer division, depending on \(\mathbf{S} \mid \mathbf{F}\). \\
\hline MAGNIFY & \\
\hline MCHAR & \\
\hline
\end{tabular}

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

\section*{MINIT}

MOTION
MULTI
N \(>\) S Converts a number from the input stream and pushes it to the stack.
OPN
OUTPT
PAB-ADDR
PAB-BUF
PAB-VBUF
PABS

PAGE
PANEL
PDT
PI
PLAY
PLAYING?
R \(>F\)
RAD/DEG
RD
REC-LEN
REC-NO
RLTV
RP@
RSTR
S"

S->F
S0\&TIB!
SAMS!
SAMS?

SAY

Returns address of top of return stack.

Accepts a "-terminated string from the input stream, storing it as a packed (counted) string.

SATR Can only change this constant via user variable number 24h.
Moves TIB (same as stack base) for SAMS use.
Initializes SAMS.
Leaves a copy of the SAMS flag.
Can only change this constant via user variable number \(\mathbf{2 4 h}\).
Speaks a counted list of existing speech-synthesizer words .

\section*{SCREEN}
\begin{tabular}{ll}
\hline SCRFNT & \begin{tabular}{l} 
User variable containing a flag indicating whether to load the default font (zero \\
flag) or a user-defined font (nonzero flag).
\end{tabular} \\
\begin{tabular}{ll} 
SCROLL & Scrolls a display screen panel one row or column in a specified direction.
\end{tabular} \\
\hline SET-PAB & \\
SIN & \\
\hline SM/REM & M/ with symmetric integer division. \\
SMTN & Can only change this constant via user variable number 26h. \\
\hline SOUND & Starts a sound on a given channel with a given pitch and volume.
\end{tabular}

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

TEXT80

```

TOKEN
TRUNC

USEFFL
UNDRAW
UPDT
VCHAR
VDPMDE
VLIST
VMODE \(\quad\) Sets the VDP mode to the mode of the number on the stack (0 - 6)
VRBL
WARNING
WHERE
WRT
WRAP [DCHAR]
]DATA Terminates a block of numbers begun with DATA [

\section*{Appendix F User Variables in fbForth 2.0}

The purpose of this appendix is to detail the User Variables in fbForth \(\mathbf{2 . 0}\) 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 .
As of fbForth 2.0:8, there is one new word ( WRAP ) added to this table and one word ( FLD ) no longer part of this table. \(\mathbf{S} \mid \mathbf{F}\) was added in fbForth 2.0:9.

\section*{F. 1 fbForth 2.0 User Variables (Address Offset Order)}
\begin{tabular}{|c|c|c|c|}
\hline Name 0 & Offset & Initial Value & Description \\
\hline UCONS\$ & 06h & 366Ch & Base of User Var initial value table \\
\hline S0 & 08h & FFA0h & Base of Stack \\
\hline R0 & 0Ah & 3FFEh & Base of Return Stack \\
\hline U0 & 0Ch & 36B6h & Base of User Variables \\
\hline TIB & 0Eh & FFA0h & Terminal Input Buffer address \\
\hline WIDTH & 10h & 31 & Name length in dictionary \\
\hline DP & 12h & A000h & Dictionary Pointer \\
\hline SYS\$ & 14h & 30DEh & Address of System Support \\
\hline CURPOS & 16h & 0 & Cursor location in VDP RAM \\
\hline INTLNK & 18h & 3020h & Pointer to Interrupt Service Linkage \\
\hline WARNING & 1Ah & 1 & Message Control \\
\hline C/L\$ & 1Ch & 64 & Characters per Line \\
\hline FIRST\$ & 1Eh & 2010h & Beginning of Disk Buffers \\
\hline LIMIT\$ & 20h & 3020h & End of Disk Buffers \\
\hline COLTAB & 22h & 380h & Color Table address in VRAM. COLTAB gets addr. \\
\hline SATR & 24h & 300h & Sprite Attribute Table address in VRAM. SATR gets addr. \\
\hline SMTN & 26h & 780h & Sprite Motion Table address in VRAM. SMTN gets addr. \\
\hline PDT & 28h & 800h & Pattern Descriptor Table addr in VRAM. PDT gets addr. \\
\hline FPB & 2Ah & 80h & User font file PAB offset from FRB. FPB gets addr. \\
\hline DISK_BUF & 2Ch & 1000h & VDP location of 128B Forth Record Buffer (FRB) \\
\hline PABS & 2Eh & 460h & VDP location for PABs \\
\hline SCRN_WIDTH & 30h & 40 & Display Screen Width in Characters \\
\hline SCRN_START & 32h & 0 & Display Screen Image Start in VDP \\
\hline SCRN_END & 34h & 960 & Display Screen Image End in VDP \\
\hline ISR & 36h & 0 & Interrupt Service Pointer \\
\hline ALTIN & 38h & 0 & Alternate Input Pointer \\
\hline ALTOUT & 3Ah & 0 & Alternate Output Pointer \\
\hline VDPMDE & 3Ch & 1 & VDP Mode \\
\hline BPB & 3Eh & C6h & Blocks PABs offset from FRB. BPB gets address. \\
\hline BPOFF & 40h & 0 & Current Blocks file offset from BPB. (0 or 70h) \\
\hline SPDTAB & 42h & 800h & Sprite Descriptor Table addr in VRAM. SPDTAB gets addr. \\
\hline
\end{tabular}
\begin{tabular}{|c|c|c|c|}
\hline Name O & Offset & Initial Value & Description \\
\hline SCRFNT & 44h & -1 & Flag for default/user font ( \(\neq 0\) ) or console font ( \(=0\) ) \\
\hline JMODE & 46h & 0 & Flag for whether JOYST executes JKBD \((=0)\) or JCRU \((\neq 0)\) \\
\hline WRAP & 48h & 0 & Flag for no wrap ( \(=0\) ) or wrap \((\neq 0)\); used by SCROLL \\
\hline S|F & 4Ah & 0 & Flag for Symmetric ( \(=0\) ) or Floored Integer Division \((\neq 0)\) \\
\hline FENCE & 4Ch & & Dictionary Fence \\
\hline BLK & 4Eh & & Block being interpreted \\
\hline IN & 50h & & Byte offset in text buffer \\
\hline OUT & 52h & & Incremented by EMIT \\
\hline SCR & 54h & & Last Forth Block (Screen) referenced \\
\hline CONTEXT & 56h & & Pointer to Context Vocabulary \\
\hline CURRENT & 58h & & Pointer to Current Vocabulary \\
\hline STATE & 5Ah & & Compilation State \\
\hline BASE & 5Ch & & Number Base for Conversions \\
\hline DPL & 5Eh & & Decimal Point Location \\
\hline CSP & 60h & & Stack Pointer for error checking \\
\hline R\# & 62h & & Editing Cursor location \\
\hline HLD & 64h & & Holds address during numeric conversion \\
\hline USE & 66h & & Next Block Buffer to Use \\
\hline PREV & 68h & & Most recently accessed disk buffer \\
\hline ECOUNT & 6Ah & & Error control \\
\hline VOC-LINK & 6Ch & & Vocabulary linkage \\
\hline [user to define] & e] 6Eh & & - available to user- \\
\hline [user to define] & e] 70h & & - available to user- \\
\hline [user to define] & e] 72h & & - available to user- \\
\hline [user to define] & e] 74 h & & - available to user- \\
\hline [user to define & e] 76h & & - available to user- \\
\hline [user to define] & e] 78h & & - available to user- \\
\hline [user to define] & e] 7Ah & & - available to user- \\
\hline [user to define] & e] 7Ch & & - available to user- \\
\hline [user to define] & e] 7Eh & & - available to user- \\
\hline
\end{tabular}

\section*{F. 2 fbForth 2.0 User Variables (Variable Name Order)}
\begin{tabular}{|c|c|c|c|}
\hline Name & Offset & Initial Value & Description \\
\hline ALTIN & 38h & 0 & Alternate Input Pointer \\
\hline ALTOUT & 3Ah & 0 & Alternate Output Pointer \\
\hline BASE & 5Ch & & Number Base for Conversions \\
\hline BLK & 4Eh & & Block being interpreted \\
\hline BPB & 3Eh & C6h & Blocks PABs offset from FRB. BPB gets address. \\
\hline BPOFF & 40h & 0 & Current Blocks file offset from BPB. (0 or 70h) \\
\hline C/L\$ & 1Ch & 64 & Characters per Line \\
\hline COLTAB & 22h & 380h & Color Table address in VRAM. COLTAB gets addr. \\
\hline CONTEXT & 56h & & Pointer to Context Vocabulary \\
\hline CSP & 60h & & Stack Pointer for error checking \\
\hline CURPOS & 16h & 0 & Cursor location in VDP RAM \\
\hline CURRENT & 58h & & Pointer to Current Vocabulary \\
\hline DISK_BUF & 2 Ch & 1000h & VDP location of 128B Forth Record Buffer (FRB) \\
\hline DP & 12h & A000h & Dictionary Pointer \\
\hline DPL & 5Eh & & Decimal Point Location \\
\hline ECOUNT & 6Ah & & Error control \\
\hline FENCE & 4Ch & & Dictionary Fence \\
\hline FIRST\$ & 1Eh & 2010h & Beginning of Disk Buffers \\
\hline FPB & 2Ah & 80h & User font file PAB offset from FRB. FPB gets addr. \\
\hline HLD & 64h & & Holds address during numeric conversion \\
\hline IN & 50h & & Byte offset in text buffer \\
\hline INTLNK & 18h & 3020h & Pointer to Interrupt Service Linkage \\
\hline ISR & 36h & 0 & Interrupt Service Pointer \\
\hline JMODE & 46h & 0 & Flag for whether JOYST executes JKBD ( \(=0\) ) or JCRU ( \(\ddagger 0\) ) \\
\hline LIMIT\$ & 20h & 3020h & End of Disk Buffers \\
\hline OUT & 52h & & Incremented by EMIT \\
\hline PABS & 2Eh & 460h & VDP location for PABs \\
\hline PDT & 28h & 800h & Pattern Descriptor Table addr in VRAM. PDT gets addr. \\
\hline PREV & 68h & & Most recently accessed disk buffer \\
\hline R\# & 62h & & Editing Cursor location \\
\hline R0 & 0Ah & 3FFEh & Base of Return Stack \\
\hline S0 & 08h & FFA0h & Base of Stack \\
\hline SATR & 24h & 300h & Sprite Attribute Table address in VRAM. SATR gets addr. \\
\hline SCR & 54h & & Last Forth Block (Screen) referenced \\
\hline SCRFNT & 44h & -1 & Flag for default/user font ( \(\neq 0\) ) or console font ( \(=0\) ) \\
\hline SCRN_END & 34h & 960 & Display Screen Image End in VDP \\
\hline SCRN_START & - 32h & 0 & Display Screen Image Start in VDP \\
\hline SCRN_WIDTH & 30h & 40 & Display Screen Width in Characters \\
\hline SMTN & 26h & 780h & Sprite Motion Table address in VRAM. SMTN gets addr. \\
\hline SPDTAB & 42h & 800h & Sprite Descriptor Table addr in VRAM. SPDTAB gets addr. \\
\hline STATE & 5Ah & & Compilation State \\
\hline SYS\$ & 14h & 30DEh & Address of System Support \\
\hline S|F & 4Ah & 0 & Flag for Symmetric ( \(=0\) ) or Floored Integer Division ( \(\ddagger=0\) ) \\
\hline TIB & 0Eh & FFA0h & Terminal Input Buffer address \\
\hline
\end{tabular}


\section*{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 and the "Stack-Based String Library" for fbForth V2.0" are the only options that do conditional loads of other blocks.

Though most load options load many more word definitions than are indicated below at "Words loaded:", only those of interest to the user and described in the glossary are listed.

\section*{G. 1 Option: 64-Column Editor}

Starting screen: 6
Words loaded: EDIT ED@ WHERE
CLIST CLINE

\section*{G. 2 Option: CPYBLK -- Block Copying Utility}

Starting screen: 4
Words loaded: SCMP CPYBLK

\section*{G. 3 Option: Memory Dump Utility}

Starting screen: 16
Words loaded: DUMP

\section*{G. 4 Option: TRACE -- Colon Definition Tracing}

Starting screen: 18

Words loaded: TRACE
UNTRACE TRON
TROFF
: (alternate)

\section*{G. 5 Option: Printing Routines}

Starting screen: 19
\begin{tabular}{llll} 
Words loaded: & SWCH & UNSWCH & ?ASCII \\
& TRIAD & TRIADS & INDEX
\end{tabular}

\section*{G. 6 Option: TMS9900 Assembler}

Starting screen: 21
Words loaded: Entire Assembler vocabulary. See Chapter 9 of the manual.

\section*{G. 7 Option: CRU Words}

Starting screen: 5
\begin{tabular}{llll} 
Words loaded: & SBO & SBZ & TB \\
& LDCR & STCR &
\end{tabular}

\section*{G. 8 Option: More Useful Stack Words etc.}

Starting screen: 41
Words loaded: 2DUP 2DROP NIP TUCK CELLS -ROT
PICK ROLL WITHIN <> \$. EXIT

\section*{G. 9 Option: Stack-based String Library}

Starting Screen: 42
Words loaded: Entire String Stack Library. See Chapter 14 of the manual.

\section*{G. 10 Option: DIR -- Disk Catalog Utility}

Starting screen: 36
Words loaded: DIR

\section*{G. 11 Option: CAT -- Disk Catalog Utility}

Starting screen: 58
Words loaded: CAT

\section*{G.12 Option: TI Forth Block Utilities}

Starting screen: 61
Words loaded: TIFBLK TIFIDX TIF2FBF TIFVU

\section*{G. 13 Option: ASM>CODE -- Code Output Utility}

Starting screen: 39
Words loaded: ASM>CODE

\section*{G. 14 Option: Compact Flash Utilities}

Starting screen: 69
Words loaded: CF? CFMOUNT CFVOLS

\section*{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:
\begin{tabular}{ll} 
SBO & - a CRU instruction \\
SBZ & - a CRU instruction \\
TB & -a CRU instruction \\
LDCR & - a CRU instruction \\
STCR & - a CRU instruction \\
DDOT & - used by the dot plotting routine \\
SMASH & - used by CLINE and CLIST \\
TCHAR & - definitions for the tiny characters \\
JCRU & -joystick access via the CRU
\end{tabular}

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:
\begin{tabular}{ll} 
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 & \(<==\)\begin{tabular}{l} 
Recover space in dictionary used by \\
\\
\end{tabular}
\end{tabular}
2. The comment, ( \(\left.{ }^{\wedge} 0\right)(\operatorname{Shift}+0)\), on line 5 is a substitute for ()), a syntax error.

For clarity of the code presentation, a few of the blocks below show the code of some of the numbered lines spanning multiple lines on the page:
```

BLOCK \#40
0 ~ ( ~ S o u r c e ~ f o r ~ C R U ~ w o r d s . . . R 1 2 ~ i s ~ C R U ~ r e g i s t e r ) ~ B A S E - > R ~ H E X ~
1 ASM: SBO ( addr --- )
2 *SP+ R12 MOV,
R12 R12 A,
3 SBO,
; ASM
4 ASM: SBZ ( addr --- )
5 *SP+ R12 MOV,
R12 R12 A,
0 SBZ,
; ASM
7 ASM: TB ( addr --- flag )
8 *SP R12 MOV,
R12 R12 A,
*SP CLR,
0 TB,
EQ IF,
*SP INC,
THEN,
;ASM R->BASE -->
14
15
BLOCK \#41
0 ( Source for CRU words ) BASE->R HEX
1 ASM: LDCR ( n1 n2 addr --- )
2
*SP+ R12 MOV,
R12 R12 A,
*SP+ R1 MOV,
3
*SP+ R0 MOV,
R1 000F ANDI,
NE IF,
R1 0008 CI,
LTE IF,
R0 SWPB,
THEN,
THEN,
R1 06 SLA,
R1 3000 ORI,
R1 X,
;ASM R->BASE -->
12
13
14
15

```
```

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,
R2 R2 MOV,
NE IF,
R02 0008 CI,
LTE IF,
R0 SWPB,
THEN,
THEN,
R0 *SP MOV,
;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 , FDFE , 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 ~ ( ~ S o u r c e ~ f o r ~ S M A S H ~ ) ~ B A S E - > R ~ H E X ~
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,
*R3+ R6 MOVB,
R5 6 SRL,
R6 6 SRL,
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,
1 2
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,
EQ UNTIL,
REPEAT,
;ASM

```
BLOCK #45
    0 ( definitions of tiny chars with true lowercase) BASE->R HEX
    1 0EEE VARIABLE TCHAR DATA[ EEEE
    20000 0000 ( ) 044444404 ( !) 0AA0 0000 ( ") 08AE AEA2 ( #)
    3 04EC 46E4 ( $) 0A24 448A (%) 06AC 4A86 ( &) 0480 0000 ( ')
    4 0248 8842 ( () 0842 2248 ( ^0) 04EE 4000 ( *) 0044 E440 ( +)
    50000 0048 ( ,) 0000 E000 ( -) 0000 0004 ( .) 0224 4488 ( /)
    6 04AA EAA4 ( 0) 04C4 4444 ( 1) 04A2 488E ( 2) 0C22 C22C ( 3)
    7 02AA AE22 ( 4) 0E8C 222C ( 5) 0688 CAA4 ( 6) 0E22 4488 ( 7)
    8 04AA 4AA4 ( 8) 04AA 622C ( 9) 0004 0040 ( :) 0004 0048 ( ;)
    90024 8420 ( <) 000E 0E00 ( =) 0084 2480 ( >) 04A2 4404 ( ?)
    10 04AE AE86 ( @) 04AA EAAA ( A) 0CAA CAAC ( B) 0688 8886 ( C)
    11 0CAA AAAC ( D) 0E88 C88E ( E) 0E88 C888 ( F) 04A8 8AA6 ( G)
    13 0888 888E ( L) 0AEE AAAA ( M) 0AAE EEAA (N) OEAA AAAE ( O)
    14 0CAA C888 ( P) 0EAA AAEC ( Q) 0CAA CAAA ( R) 0688 422C ( S)
    15 -->
```

BLOCK \#46
0 ( definitions of tiny chars with true lowercase continued)
$10 E 444444$ ( T) 0AAA AAAE ( U) 0AAA AA44 ( V) 0AAA AEEA ( W)
2 0AA4 44AA ( X) 0AAA E444 (Y) 0E24 488E (Z) 06444446 ( [)
308844422 ( <br>) 0C44 444C ( ]) 044A A000 ( \$) 0000 000F ( _)
404200000 ( `) 000E 2EAE ( a) 088C AAAC (b) 00068886 ( c)
50226 AAA6 ( d) 0004 AE86 ( e) 0688 E888 ( f) 0006 A62C ( g)
6 088C AAAA ( h) 04044442 ( i) 0202 22A4 ( j) 088A ACAA ( k)
704444444 ( l) 000A EEAA ( m) 0008 EAAA ( n) 0004 AAA4 ( o)
8 000C AC88 (p) 0006 A622 ( q) 0008 E888 (r) 0006 842C ( s)
9 044E 4442 ( t) 000A AAA6 ( u) 000A AAA4 ( v) 000A AEEA ( w)
10 000A A4AA ( $x$ ) 000A A62C ( y) 000E 248E (z) 06448446 ( \{)
1104440444 ( |) 0C44 244C ( \}) 02 E 80000 ( ~) 0EEE EEEE ( DEL)
12 JDATA DROP DROP R->BASE ; $S$
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,
R12 6 LI,
R1 5 STCR,
R1 SWPB,
R1 INV,
R1 001F ANDI,
R1 *SP MOV,
83D6 @() CLR, ( defeat auto screen blanking without KSCAN) ; ASM

## Appendix I Error Messages

| Error\# | Message | Probable Causes |
| :--- | :--- | :--- |
| 1 | empty stack | Procedure being executed attempts to pop a number off the <br> parameter stack when there is no number on the parameter <br> stack. The error may have occurred long before it is <br> detected because Forth checks for this condition only when <br> control returns to the outer interpreter. |
| 2 | dictionary full | The user dictionary space is full. Too many definitions have <br> been compiled. |
| 4 | isn't unique | This message is more a warning than an error. It informs the <br> user that a word with the same name as the one just <br> compiled is already in the CURRENT or CONTEXT <br> vocabulary. |
| 5 | FBLOCKS not current | This message is displayed when fbForth 2.0 needs to read <br> from the system blocks file, FBLOCKS, and the user has <br> made another blocks file current with USEBFL. This is <br> likely the result of executing MENU without FBLOCKS <br> current. |
| 6 | disk error | This has several possible causes: No disk in disk drive, disk <br> not initialized, disk drive or controller not connected <br> properly, disk drive or controller not plugged in. The <br> diskette may be damaged with some sector having a hard <br> error. |
| 7 | The procedure being executed is leaving extra unwanted |  |
| numbers on the parameter stack resulting in a stack |  |  |


| Error\# | Message | Probable Causes |
| :---: | :---: | :---: |
|  |  | 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, etc. |
| 10 | floating point error | This error message will be issued only when ?FLERR is executed and a true flag is returned. <br> FLERR may be executed to fetch the floating point status byte. |
|  |  | code meaning |
|  |  | 01 Overflow |
|  |  | 02 Syntax |
|  |  | 03 Integer overflow on conversion |
|  |  | 04 Square root of negative |
|  |  | 05 Negative number to non-integer power |
|  |  | 06 Logarithm of a non-positive number |
|  |  | 07 Invalid argument in a trigonometric function |
| 17 | compilation only | Occurs when conditional constructs such as DO ... LOOP or IF ... THEN 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 DO has been left without a LOOP, an IF has no corresponding ENDIF or THEN, etc. |
| 20 | definition not finished | A ; was encountered and the parameter stack was not at the same height as when the preceding : was encountered. For example, an incomplete conditional construct such as : $\mathbf{x x}$ IF ; , will trigger this error message. |
| 21 | in protected dictionary | An attempt was made to FORGET a word with an address lower than or equal to that of TASK (last word in resident dictionary) or the contents of FENCE if that is higher. |
| 22 | use only when loading | This usually means an attempt was made to use --> on the command line. |
| 25 | bad jump token | Improper use of jump tokens or conditionals in the fbForth 2.0 TMS9900 Assembler. |

## Appendix J Contents of FBLOCKS

The contents of the fbForth $\mathbf{2 . 0}$ 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 14 blank blocks (blocks 14, 57, 68, $70-80$ ), which you can use as you wish.
Note, also, that the following file is dated 19APR2017:

```
BLOCK #1 ( old TIF #3)
    0 ( fbForth WELCOME SCREEN---LES 19APR2017)
    1 BASE->R HEX
    2 : MENU 1 BLOCK 2+ @ 6662 - 5 ?ERROR 2 LOAD ;
    3." FBLOCKS mod: 19APR2017"
    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 PAGE ." Load Options (19APR2017) fbForth 2.0:"
    ( Type build #) BASE->R HEX 6033 C@ EMIT R->BASE CR CR
        ." Description Load Block" CR
        ." ---------------------------------" CR
        ." CPYBLK -- Block Copying Utility.......4" CR
```



```
        ." 64-Column Editor...........................6" CR
        ." Memory Dump Utility.......................16" CR
        ." TRACE -- Colon Definition Tracing....18" CR
        ." Printing Routines.........................19" CR
        ." TMS9900 Assembler.........................21" CR
        ." More Useful Stack Words etc..........41" CR
        ." Stack-based String Library............42" CR
        ." DIR -- Disk Catalog Utility.............36" CR
        ." CAT -- Disk Catalog Utility...........58" CR
    15 ." TI Forth Block Utilities..............61" CR
```

```
BLOCK #3
    0 ." ASM>CODE -- Code Output Utility......39" CR
    ." Compact Flash Utilities................69" CR
    ." TMS9900 Assembler (v2.0:9 binary)....27" CR
    ." 64-Column Editor (v2.0:9 binary)......32" CR
    ." String Library (v2.0:9 binary).......52" CR CR
    ." Type <block> LOAD to load. " ;S
6
7
8
9
10
1 1
12
13
14
15
BLOCK #4 ( old TIF #39)
    0 ( Block Copy 17JUN2016 LES ) CR CR ." CPYBLK copies a range
    l of blocks to the same or another file, e.g.," CR CR ." CPYB
    2 LK 5 8 DSK1.F1 9 DSK2.F2" CR CR ." will copy blocks 5-8 from DS
    3 K1.F1 to DSK2.F2 starting at block 9." CR CR 0 CLOAD CPYBLK
    4 BASE->R DECIMAL 0 VARIABLE SFL 0 VARIABLE DFL 0 CONSTANT XD
    5 : SCMP OVER C@ OVER C@ OVER OVER - SGN >R MIN 1+ 0 SWAP 1 DO
    6 DROP OVER I + C@ OVER I + C@ - SGN DUP IF LEAVE THEN LOOP R>
    7 OVER 0= IF OR ELSE DROP THEN SWAP DROP SWAP DROP ;
    8 : GBFL BL WORD HERE DUP C@ 1+ =CELLS
    9 ALLOT SWAP ! ; : CPYBLK EMPTY-BUFFERS 1 ' XD ! HERE BPB BPOFF
    10 @ + 9 + DUP VSBR 1+ HERE SWAP DUP =CELLS ALLOT VMBR N>S N>S
    11 OVER OVER > IF SWAP THEN OVER - 1+ >R SFL GBFL N>S DFL GBFL SFL
    12 @ DFL @ SCMP 0= IF OVER OVER - DUP 0< SWAP R MINUS > + 2 = IF
    13 SWAP R + 1- SWAP R + 1- -1 ' XD ! THEN THEN CR R> 0 DO OVER DUP
    14 . OVER SFL @ (UB) SWAP BLOCK 2- ! DFL @ (UB) UPDATE FLUSH XD +
    15 SWAP XD + SWAP LOOP DROP DROP DUP (UB) DP ! ; R->BASE
BLOCK #5 ( old TIF #88)
    0 ( CRU WORDS 120CT82 LAO ) 0 CLOAD STCR
    1
    2 BASE->R HEX
    3 CODE: SBO C339 A30C 1D00 ;CODE
    4 CODE: SBZ C339 A30C 1E00 ;CODE
    5 CODE: TB C319 A30C 04D9 1F00 1601 0599 ;CODE
    6
    7 CODE: LDCR C339 A30C C079 C039 0241 000F 1304 0281
    8 0008 1501 06C0 0A61 0261 3000 0481 ;CODE
    9
10 CODE: STCR C339 A30C C059 04C0 0241 000F C081 0A61 0261 3400
11 0481 C082 1304 0282 0008 1501 06C0 C640 ;CODE
12
13 CR ." See Manual for usage." CR R->BASE
14
15
```

```
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 O VARIABLE CUR
        : !CUR 0 MAX 3FF MIN CUR ! ;
        : +CUR CUR @ + !CUR ;
        : +LIN CUR @ C/L / + C/L * !CUR ; DECIMAL
        : 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\mathrm{ COLUMN EDITOR ) BASE->R HEX
    1
    2 : CINIT
    3 SATR 2 0 DO DUP >R D000 SP@ R> 2 VMBW DROP 4 + LOOP DROP
        0000 0000 0000 0000 5 SPCHAR 0 CUR !
        F090 9090 9090 90F0 6 SPCHAR 0 1 F 5 0 SPRITE ; DECIMAL
    : PLACE CUR @ 64 /MOD 8 * 1+ SWAP 4 * 1- DUP 0< IF DROP 0 ENDIF
        SWAP 0 SPRPUT ;
    : UP -64 +CUR PLACE ;
    : DOWN 64 +CUR PLACE ;
    : LEFT -1 +CUR PLACE ;
    : RIGHT 1 +CUR PLACE ;
    : CGOTOXY ( col row --- ) 64 * + !CUR PLACE ;
    14
    15 R->BASE -->
BLOCK #8 ( old TIF #24)
    0 ( 64 COLUMN EDITOR ) BASE->R ." ."
    1
    DECIMAL
        : .CUR CUR @ C/L /MOD CGOTOXY ;
    : DELHALF PAD 64 BLANKS PTR PAD C/L R/C DROP - CMOVE ;
    : DELLIN R/C SWAP MINUS +CUR PTR PAD C/L CMOVE DUP L/SCR SWAP
        DO PTR 1 +LIN PTR SWAP C/L CMOVE LOOP
        0 +LIN PTR C/L 32 FILL C/L * !CUR ;
    : INSLIN R/C SWAP MINUS +CUR L/SCR +LIN DUP 1+ L/SCR 0 +LIN
        DO PTR -1 +LIN PTR SWAP C/L CMOVE -1 +LOOP
        PAD PTR C/L CMOVE C/L * !CUR ;
        : RELINE R/C SWAP DROP DUP LINE. UPDATE .CUR ;
        : +.CUR +CUR .CUR ;
    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 =
                UNTIL
            ENDIF
            BEGIN CUR @ IF 1- DUP -1 +CUR C@ BL > ELSE .CUR 1 ENDIF UNTIL
            BEGIN CUR @ IF 1- DUP -1 +CUR C@ BL = DUP IF 1 +.CUR ENDIF
                    ELSE .CUR 1 ENDIF
            UNTIL DROP ;
            : TAB PTR DUP C@ BL = 0=
            IF BEGIN 1+ DUP 1 +CUR C@ BL =
                UNTIL
            ENDIF
            CUR @ 1023 = IF .CUR 1
                                    ELSE BEGIN 1+ DUP 1 +CUR C@ BL > UNTIL .CUR
                                    ENDIF DROP ; R->BASE -->
BLOCK #10 ( old TIF #26)
    0 ( 64 COLUMN EDITOR ) BASE->R
    D 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 }3
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\mathrm{ COLUMN EDITOR 15JUL82 LAO ) 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
                        ENDIF
                        ELSE ( NEW KEY ) 1 ( FORCE LOOP EXIT ) ENDIF
        ELSE ( NO KEY PRESSED) RH RLOG ! 0 KC ! 0
        ENDIF
    15 UNTIL DUP OKEY ! ; R->BASE -->
```

```
BLOCK #12 ( old TIF #28 & #29)
    0 ( 64 COLUMN EDITOR ) BASE->R HEX ." ."
    1 : EDT VDPMDE @ >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
    901 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 0 VARIABLE LB FE ALLOT
    6 CODE: SMASH ( ADDR #CHAR LINE# --- LB VADDR CNT )
        C079 C0B9 C0D9 0204 LB , C644 0649 06C1 0221 2000 C641 C042
        0 5 8 1 ~ 0 2 4 1 ~ F F F E ~ 0 6 4 9 ~ 0 A 2 1 ~ C 6 4 1 ~ A 0 8 3 ~ 8 0 C 2 ~ 1 5 0 1 ~ 1 0 2 0 ~ 0 4 C 5 ~ 0 4 C 6 ~
        D173 D1B3 0965 0966 C025 TC , C066 TC , 0B41 020C 0004 C2C0
        024B F000 C1C1 0247 0F00 E1CB DD07 0BC0 0BC1 060C 16F4 05C5
        05C6 C305 024C 0002 16E7 10DD ;CODE
    DECIMAL
13 : CLINE LB 256 ERASE SMASH VMBW ;
14 : CLOOP DO I 64 * OVER + 64 I CLINE LOOP DROP ;
15 : CLIST BLOCK 16 0 CLOOP ; R->BASE
BLOCK #14
    0
    1
    2
    3
    4
    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*** B INARY CHARACTER DATAN * * * *
    7* * * ****
    8 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    9 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    1 0
    11
    12
    13
    14
    15
BLOCK #16 ( old TIF #42)
    0 ( DUMP ROUTINES 12JUL82 LCT...250CT2015 LES mod)
    1 0 CLOAD DUMP BASE->R HEX CR ." loading memory dump utility"
    2 : VM+ VDPMDE @ 0= IF + ELSE DROP THEN ;
    3 : DUMP8 -DUP
    IF
        BASE->R HEX 0 OUT ! OVER 4 U.R 3A EMIT
        OVER OVER 0 DO
            DUP @ 0 <# # # # # BL HOLD BL HOLD #> TYPE 2+ 2
            +LOOP DROP 1F 18 VM+ OUT @ - SPACES
        0 DO
            DUP C@ DUP 20 < OVER 7E > OR
            IF DROP 2E ENDIF
            EMIT 1+
        LOOP
        CR R->BASE ENDIF ; -->
BLOCK #17 ( old TIF #43)
    0 ( DUMP ROUTINES 12JUL82 LCT...250CT2015 LES mod) ." ."
    1 : DUMP CR 00 8 8 VM+ U/ >R SWAP R> -DUP
    IF 0
    DO 8 8 VM+ DUMP8 PAUSE IF SWAP DROP 0 SWAP LEAVE ENDIF LOOP
    4 ENDIF SWAP DUMP8 DROP ;
    5 ( .S and VLIST have been put in resident dictionary)
    6 R->BASE ;S
    7
    8
    9
    10
    11
    12
    13
    14
    15
```

```
BLOCK #18 ( old TIF #44)
    0 ( TRACE COLON WORDS-FORTH DIMENSIONS III/2 P.58 260CT82 LCT)
    10 CLOAD (TRACE) CR ." loading colon definition tracing "
    2 FORTH DEFINITIONS
    30 VARIABLE TRACF ( CONTROLS INSERTION OF TRACE ROUTINE )
    40 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 )
    ?EXEC !CSP CURRENT @ CONTEXT ! CREATE [ ' : CFA @ ] LITERAL
    HERE 2- ! TRACF @ IF ' (TRACE) CFA DUP @ HERE 2- ! , ENDIF ]
        ; IMMEDIATE
BLOCK #19 ( old TIF #72)
    0 ( ALTERNATE I/O SUPPORT FOR RS232 PNTR 12JUL82 LCT...mod LES)
    0 CLOAD INDEX CR ." loading printing routines"
    0 0 0 FILE >RS232 BASE->R HEX
    : SWCH >RS232 PABS @ 10 + DUP PAB-ADDR ! 1- PAB-VBUF !
    SET-PAB OUTPT F-D" RS232.BA=9600" OPN 3
    PAB-ADDR @ VSBW 1 PAB-ADDR @ 5 + VSBW PAB-ADDR @ ALTOUT ! ;
    : UNSWCH 0 ALTOUT ! CLSE ;
    : ?ASCII ( BLOCK# --- FLAG )
            BLOCK 0 SWAP DUP 400 + SWAP
            DO I C@ 20 > + I C@ DUP 20 < SWAP 7F > OR
                IF DROP 0 LEAVE ENDIF LOOP ;
    : TRIAD 0 SWAP SWCH 3 / 3 * 1+ DUP 3 + SWAP
    DO I ?ASCII IF 1+ I LIST CR ENDIF LOOP
    -DUP IF 3 SWAP - 14 * 0 DO CR LOOP
    ." fbForth --- a TI-Forth/fig-Forth extension" OC EMIT
    ENDIF UNSWCH ; R->BASE -->
BLOCK #20 ( old TIF #73)
    0 ( SMART TRIADS AND INDEX 15SEP82 LAO...mod LES )
    BASE->R DECIMAL
    : TRIADS ( from to --- )
        3 / 3 * 2+ SWAP 3 / 3 * 1+ DO I TRIAD 3 +LOOP ;
    : INDEX ( from to --- ) 1+ SWAP
        DO I DUP ?ASCII IF CR 4 .R 2 SPACES I BLOCK 64 TYPE ELSE DROP
        ENDIF PAUSE IF LEAVE ENDIF LOOP ; R->BASE ;S
```

```
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' ;
    40440 GOP B, 0680 GOP BL, 0400 GOP BLWP,
    5 04C0 GOP CLR, 0700 GOP SETO, 0540 GOP INV,
    60500 GOP NEG, 0740 GOP ABS, 06C0 GOP SWPB,
    70580 GOP INC, 05C0 GOP INCT, 0600 GOP DEC,
    80640 GOP DECT, 0480 GOP X,
    9 : GROP <BUILDS , DOES> @ SWAP 40 * + GOP' ;
    10 2000 GROP COC, 2400 GROP CZC, 2800 GROP XOR,
    11 3800 GROP MPY, 3C00 GROP DIV, 2C00 GROP 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\mathrm{ 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 : 00P <BUILDS , DOES> @ , ;
    30340 00P IDLE, 0360 00P RSET, 03C0 00P CKOF,
    4 03A0 00P CKON, 03E0 00P LREX, 0380 00P RTWP,
    5 : ROP <BUILDS , DOES> @ + , ; 02C0 ROP STST, 02A0 ROP STWP,
    6 : IOP <BUILDS , DOES> @ , , ; 02E0 IOP LWPI, 0300 IOP LIMI,
    7 : RIOP <BUILDS , DOES> @ ROT + , , ; 0220 RIOP AI,
    80240 RIOP ANDI, }0280\mathrm{ 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 SBO, 1E00 DOP SBZ, 1F00 DOP TB, -->
BLOCK #23 ( old TIF #77)
    0 ( ASSEMBLER 12JUL82 LCT)
    1 : GCOP <BUILDS , DOES> @ SWAP 000F AND 040 * + GOP' ;
    2 3000 GCOP LDCR, 3400 GCOP STCR,
    300 CONSTANT R0 01 CONSTANT R1 }02\mathrm{ CONSTANT R2 03 CONSTANT R3
    404 CONSTANT R4 05 CONSTANT R5 06 CONSTANT R6 07 CONSTANT R7
    508 CONSTANT R8 09 CONSTANT R9 0A CONSTANT R10 0B CONSTANT R11
    0C CONSTANT R12 0D CONSTANT R13 0E CONSTANT R14
    7 0F CONSTANT R15 08 CONSTANT UP 09 CONSTANT SP 0A CONSTANT W
    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 OC ;
    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
                DUP 0< OVER 10 > OR IF 19 ERROR ENDIF DUP
            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
        42 ?PAIRS HERE OVER - 2- 2 / SWAP 1+ C! ; IMMEDIATE
        : ELSE, ?EXEC 42 ?PAIRS 0 [COMPILE] CJMP HERE 2- SWAP 42
        [COMPILE] ENDIF, 42 ; IMMEDIATE
        : BEGIN, ?EXEC HERE 41 ; IMMEDIATE
        : UNTIL, ?EXEC SWAP 41 ?PAIRS [COMPILE] CJMP HERE - 2 / 00FF
        AND HERE 1- C! ; IMMEDIATE
    : AGAIN, ?EXEC 0 [COMPILE] UNTIL, ; IMMEDIATE
    : REPEAT, ?EXEC >R >R [COMPILE] AGAIN,
        R> R> 2- [COMPILE] ENDIF, ; IMMEDIATE
    : WHILE, ?EXEC [COMPILE] IF, 2+ ; IMMEDIATE
    ( : 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 for fbForth 2.0:9
    1 ." loading TMS9900 Assembler "
    2 BASE->R DECIMAL 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
    1 3
    14
    15
BLOCK #28 - BLOCK #31 TMS9900 Assembler Binary
    0 ********************************
    1 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
    2 * *******************************
    3 * * *
    4***
    5*** F O U R * * * *
    6 * * *
    7*** B L O C K S O F B I N A R Y C O D E * * * *
    8 * * *
    9*** F O U R * * * *
10*** ****
11 * * * * * * * *
12** * 13** ********************************
14***************************************
15 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

BLOCK \#32
0 \ 64-Column Editor BLOAD for fbForth 2.0:9
1 :" loading 64-column editor "
2 BASE->R 33 R->BASE BLOAD
3 : BLERR IF ." BLOAD error!" THEN ; BLERR FORGET BLERR ;S
4
5
6
7
8
9
10
11
12
13
14
15


```
BLOCK #38
    0 BASE->R
    1 : DIR
    2 Cat SET-PAB ( Initialize PAB skeleton)
    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!
        ( Get the catalog and display it)
        OPN ( open the catalog)
        DskInfo ( display disk info)
        DoDIR ( display file list)
        CLSE ( close the catalog) ;
    R->BASE CR
    ." DIR - Catalogs a disk." CR
    ." E.g., DIR DSK1." CR
14
15
```


## BLOCK \#39

0 ( ASM>CODE [port of Mark Wills' code] LES20JUN2016)
1 CR ." Loading ASM>CODE" 0 CLOAD ASM>CODE BASE @ HEX
20 VARIABLE pfa 0 VARIABLE STRPOS 0 VARIABLE FBUF 4E ALLOT
3 PABS @ FBUF 1200 FILE FileOut FileOut SET-PAB
4 : ClearBUF FBUF 50 BLANKS ; : SetFileName ( IS:fileName )
5 BL WORD HERE PAB-ADDR @ 9 + OVER C@ 1+ VMBW ;
6 : ApdERR ( 0 msg\# -- flag ) DROP PAB-ADDR @ 1+ VSBR 0E0 AND 7 OR R> R> DROP >R ; : instApdERR ' ApdERR CFA' (ABORT) ! 8 -1 WARNING ! ; : uninstApdERR ' ABORT CFA ' (ABORT) ! 1
9 WARNING ! ; : OpenFile ( -- ) FileOut DSPLY VRBL 50 REC-LEN
10 instApdERR 0 APPND OPN uninstApdERR IF OUTPT OPN THEN ;
11 : Asm? pfa @ DUP CFA @ = ;
12 : copyStr ( addr count -- ) STRPOS @ 5 * FBUF + SWAP CMOVE ;
13 : SetName ClearBUF S" CODE: " COUNT copyStr pfa @ NFA DUP C@
14 01F AND SWAP 1+ SWAP FBUF 6 + SWAP 0 DO OVER C@ 07F AND
15 OVER C! 1+ SWAP 1+ SWAP LOOP DROP DROP ; -->
BLOCK \#40
0 ( ASM>CODE..continued LES20JUN2016) ." ."
1 : FlushLine 40 WRT ClearBUF 0 STRPOS ! ; : PlaceCell pfa @ @
20 <\# \# \# \# \# \#> copyStr 1 STRPOS +! 2 pfa +! ;
3 : \&; S" ;CODE" COUNT copyStr ;
4 : ProcessWord SetName FlushLine BASE->R 10 BASE ! BEGIN pfa @ @
$5045 \mathrm{~F}=0=$ WHILE PlaceCell STRPOS @ 0C = IF FlushLine THEN
REPEAT \&; FlushLine R->BASE ;
7 : ASM>CODE ( IS:wordName fileName) CR -FIND IF DROP ELSE 0 THEN
pfa ! SetFileName pfa @ IF Asm? IF OpenFile ProcessWord CLSE
ELSE ." Not an assembly language word" THEN
ELSE ." Word not found" THEN ;
CR ." Usage: ASM>CODE <name> <file>"
CR ." E.g.: ASM>CODE MYWORD DSK1.MYWORD" CR BASE ! ;S
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
        George Smyth's Forth Forum)
    : ROLL ( [n]..[0] +n -- [n-1]..[0][n] )
        -DUP IF 1- SWAP >R MYSELF R> SWAP THEN ;
    : WITHIN ( n low high -- true|false ) OVER - >R - R> U< ;
    : <> ( a b -- 1|0 ) = 0= ; : $. BASE->R HEX U. R->BASE ;
    : EXIT ( -- ) [COMPILE] ;S ; IMMEDIATE
BLOCK #42
    ( Portable, Stack Based String Library for fbForth V2.0 )
    ( v 1.0 - Mark Wills Sept 2014.)
    ( Ported from the original TurboForth code by Mark Wills )
    ( Modified by Lee Stewart October 2014)
    BASE->R DECIMAL 41 R->BASE CLOAD $. 0 CLOAD $.S
    CR ." Loading String Library"
    0 CONSTANT ($sSize)
    HERE CONSTANT ($sEnd)
    ($sEnd) VARIABLE ($sp)
    0 VARIABLE ($temp1)
    0 0 VARIABLE ($depth)
    0 VARIABLE ($temp0)
    0 VARIABLE ($temp2)
    0 VARIABLE ($temp3) -->
    14
    15
```


## BLOCK \#43

```
        0 ( Throw codes for string library, mod: Lee Stewart)
        BASE->R DECIMAL ." ."
        : (throw) ( code -- )
        CASE
            ($sSize) 0= IF DROP 9999 THEN
            9900 OF ." String stack underflow" ENDOF
            9901 OF ." String too large to assign" ENDOF
            9902 OF ." String stack is empty" ENDOF
            9903 OF ." Need at least 2 strings on string stack" ENDOF
            9904 OF ." String too large for string constant" ENDOF
            9905 OF ." Illegal LEN value" ENDOF
            9906 OF ." Need at least 3 strings on string stack" ENDOF
            9908 OF ." Illegal start value" ENDOF
            9999 OF ." String stack not initialized" ENDOF
        ENDCASE
        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@) + ($sEnd)
    < < IF 9900 (throw) THEN ($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: -- $Caddr)
    2 <BUILDS ($r
    3 : CLEN$ ( $Caddr -- len ) cell+ @ ;
    4 : MAXLEN$ ( $Caddr -- max_len ) (lenOf$) ;
    5 : .$CONST ( $Caddr -- ) cell+ DUP (lenOf$)
    SWAP cell+ SWAP TYPE ;
    7 : :=" ( $Caddr tib:"string" -- ) DUP @ 34 WORD HERE COUNT
        SWAP >R 2DUP < IF 9901 (throw) THEN NIP 2DUP SWAP cell+
        ! >R [ 2 CELLS ] LITERAL + R> R> -ROT CMOVE ;
    ($") ( addr len -- ) ( ss: -- str ) DUP ($rUp) cell+ (set$SP)
        DUP ($sp@) ! ($sp@) cell+ SWAP CMOVE ($depth+) ;
    : (COMPILE$) ( addr len -- ) DUP >R PAD SWAP CMOVE HERE 6 CELLS
        COMPILE LIT + , COMPILE LIT R , COMPILE BRANCH HERE R
        ($rUp) + HERE - 2+ , PAD 12 - R HERE SWAP CMOVE R> ($rUp)
        ALLOT COMPILE ($") ; R->BASE -->
BLOCK #46
    0 ( String stack words) BASE->R DECIMAL ." ."
    1 : $" 34 WORD HERE COUNT STATE @ IF (COMPILE$) ELSE ($") THEN ;
        IMMEDIATE : >$ cell+ DUP (lenOf$) SWAP cell+ SWAP ($") ;
        PICK$ DEPTH$ 0= IF 9902 (throw) THEN
        (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 ($sp@) DUP (sizeOf$)
        HERE SWAP CMOVE 1 (addrOf$) DUP (sizeOf$) ($sp@) SWAP CMOVE
        HERE DUP (sizeOf$) ($sp@) DUP (sizeOf$) + SWAP CMOVE ;
    : NIP$ DEPTH$ 2 < IF 9903 (throw) THEN SWAP$ DROP$ ;
    : OVER$ DEPTH$ 2 < IF 9903 (throw) THEN 1 PICK$ ;
    (rot$) ($sp@) 3 (addrOf$) ($sp@) (sizeOf$)
        1 (addrOf$) (sizeOf$) 2 (addrOf$) (sizeOf$) + + CMOVE
        3 (addrOf$) ($sp) ! -3 ($depth) +! ; R->BASE -->
```

```
BLOCK #47
    0 ( String stack words) BASE->R DECIMAL ." ."
    1 : ROT$ DEPTH$ 3 < IF 9906 (throw) THEN
    1 PICK$ 1 PICK$ 4 PICK$ (rot$) ;
    : -ROT$ DEPTH$ 3 < IF 9906 (throw) THEN
        0 PICK$ 3 PICK$ 3 PICK$ (rot$) ;
    : LEN$ DEPTH$ 1 < IF 9902 (throw) THEN ($sp@) @ ;
    : >$CONST >R DEPTH$ 1 < IF 9902 (throw) THEN LEN$ R @ > IF 9904
        (throw) THEN ($sp@) DUP (sizeOf$) R> cell+ SWAP CMOVE DROP$ ;
        : +$ DEPTH$ 2 < IF 9903 (throw) THEN 1 (addrOf$) cell+ HERE 1
    (addrOf$) (lenOf$) CMOVE ($sp@) cell+ 1 (addrOf$) (lenOf$) HERE
    + LEN$ CMOVE HERE LEN$ 1 (addrOf$) (lenOf$) + DROP$ DROP$ ($") ;
        : MID$ DEPTH$ 1 < IF 9902 (throw) THEN DUP LEN$ > OVER 1 < OR
        IF 9905 (throw) THEN OVER DUP LEN$ > SWAP 0< OR IF 9908
        (throw) THEN SWAP ($sp@) cell+ + SWAP ($") ;
    : LEFT$ DEPTH$ 1 < IF 9902 (throw) THEN DUP LEN$ > OVER 1 < OR
    15 IF 9905 (throw) THEN 0 ($sp@) cell+ + SWAP ($") ; R->BASE -->
BLOCK #48
    0 ( String stack words)
    BASE->R DECIMAL ." ."
    : RIGHT$ DEPTH$ 1 < IF 9902 (throw) THEN DUP LEN$ > OVER 1 <
        OR IF 9905 (throw) THEN ($sp@) (lenOf$) OVER -
        ($sp@) cell+ + SWAP ($") ;
    : FINDC$ DEPTH$ 1 < IF 9902 (throw) THEN -1 ($temp0) ! ($sp@)
        cell+ ($sp@) (lenOf$) 0 DO DUP C@ 2 PICK = IF I ($temp0) !
        LEAVE THEN 1+ LOOP DROP DROP ($temp0) @ ;
    : FIND$ DEPTH$ 2 < IF 9903 (throw) THEN LEN$ ($temp1) ! 1
        (addrOf$) (lenOf$) ($temp0) ! DUP ($temp0) @ > IF DROP -1 EXIT
        THEN 1 (addrOf$) cell+ + ($temp2) ! ($sp@) cell+ ($temp3) !
        ($temp1) @ ($temp0) @ > IF DROP -1 EXIT THEN 0 ($temp0) @ 0 DO
        ($temp3) @ OVER + C@ ($temp2) @ I + C@ = IF 1+ DUP ($temp1) @
        = IF DROP I ($temp1) @ - 1+ -2 LEAVE THEN ELSE DROP 0 THEN
        LOOP DUP -2 = IF DROP ELSE DROP -1 THEN DROP$ ;
    : .$ DEPTH$ 0= IF 9902 (throw) THEN
        ($sp@) cell+ ($sp@) (lenOf$) TYPE DROP$ ; R->BASE -->
BLOCK #49
        ( String stack words) BASE->R DECIMAL ." ."
        : REV$ DEPTH$ 0= IF 9902 (throw) THEN ($sp@) DUP cell+ >R
        (lenOf$) R> SWAP HERE SWAP CMOVE ($sp@) (lenOf$) HERE 1- +
        ($sp@) cell+ DUP ($sp@) (lenOf$) + SWAP DO
        DUP C@ I C! 1- LOOP DROP ;
    : LTRIM$ DEPTH$ 0= IF 9902 (throw) THEN ($sp@) DUP (lenOf$) >R
        HERE OVER (sizeOf$) CMOVE 0 R> HERE cell+ DUP >R + R> DO I C@
        BL = IF 1+ ELSE LEAVE THEN LOOP DUP 0 > IF >R ($sp@) (lenOf$)
        DROP$ HERE cell+ R + SWAP R> - ($") ELSE DROP THEN ;
    : RTRIM$ DEPTH$ 0= IF 9902 (throw) THEN REV$ LTRIM$ REV$ ;
    : UCASE$ DEPTH$ 1 < IF 9902 (throw) THEN ($sp@) DUP (lenOf$) +
        cell+ ($sp@) cell+ DO I C@ DUP 97 123 WITHIN IF 32 - I
        C! ELSE DROP THEN LOOP ; : TRIM$ RTRIM$ LTRIM$ ;
    : LCASE$ DEPTH$ 1 < IF 9902 (throw) THEN ($sp@) DUP (lenOf$) +
        cell+ ($sp@) cell+ DO I C@ DUP 65 91 WITHIN IF
        32 + I C! ELSE DROP THEN LOOP ; R->BASE -->
```

```
BLOCK #50
    0 ( String stack words, mod: LES [CMP$ added])
    BASE->R DECIMAL ." ."
    : REPLACE$ DEPTH$ 3 < IF 9906 (throw) THEN LEN$ >R 0 FIND$ DUP
        ($temp0) ! -1 > IF ($sp@) cell+ HERE ($temp0) @ CMOVE 1
        (addrOf$) cell+ HERE ($temp0) @ + 1 (addrOf$) (lenOf$) CMOVE
        ($sp@) cell+ ($temp0) @ + R + HERE ($temp0) @ + 1 (addrOf$)
        (lenOf$) + LEN$ R> - ($temp0) @ - DUP >R CMOVE R> ($temp0)
        @ + 1 (addrOf$) (lenOf$) + DROP$ DROP$ HERE SWAP ($")
        ELSE R> DROP THEN ($temp0) @ ;
    : CMP$ DEPTH$ 2 < IF 9903 (throw) THEN 1 (addrOf$) cell+ ($sp@)
        cell+ 1 (addrOf$) (lenOf$) LEN$ OVER OVER - SGN >R MIN 0
        SWAP 0 DO DROP OVER I + C@ OVER I + C@ - SGN DUP IF LEAVE
        THEN LOOP R> OVER 0= IF OR ELSE DROP THEN -ROT DROP DROP ;
    : VAL$ ($sp@) DUP (lenOf$) >R cell+ PAD 1+ R CMOVE
        R PAD C! 32 PAD R> + 1+ C! PAD NUMBER DROP$ ;
                        R->BASE -->
BLOCK #51
    0 ( String stack words) BASE->R DECIMAL ." ." CR
    1 : $.S CR DEPTH$ 0 > IF ($sp@) DEPTH$ ." Index|Length|String"
        CR ." -----+-----+-----" CR 0 BEGIN DEPTH$ 0 > WHILE DUP
        6 .R ." |" LEN$ 6 .R ." |" .$ 1+ CR REPEAT DROP ($depth) !
        ($sp) ! CR ELSE ." String stack is empty." CR THEN
        ." Allocated stack space:" ($sEnd) ($sSize) + ($sp@) - 4 .R
        ." bytes" CR ." Total stack space:" ($sSize) 4 .R
        ." bytes" CR ." Stack space remaining:" ($sp@) ($sEnd) - 4
        .R ." bytes" CR ; R->BASE
    ." You MUST initialize the string stack before you can use the
    string library:" CR
        512 INIT$" CR
    ." will create a string stack with 512 bytes available." CR
    ." Example: $" 34 EMIT ." RED" 34 EMIT ." $" 34 EMIT
        ." GREEN" 34 EMIT ." $" 34 EMIT ." BLUE" 34 EMIT ." $.S"
    CR
BLOCK #52
    0 \ String Library BLOAD for fbForth 2.0:9
    1 ." loading string library " CR
    2 ." You MUST initialize the string stack before you can use the
    3 string library:" CR
    4 ." }512\mathrm{ INIT$" 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"
        CR
    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 #59
    0 BASE->R DECIMAL
    1 : getFree ( --- n ) 0 Buf2 ! Buf1 56 + DUP 200 + SWAP DO
        I @ 65535 XOR -DUP IF 16 0 DO DUP 1 AND Buf2 +! 1 SRL LOOP
        DROP THEN 2 +LOOP Buf2 @ ; : Head1 ( --- )
        ." ---------- ---- --- --- ----- -" CR ; : Head ( --- )
        " Name Size Typ B/R Bytes P" CR Head1 ; HEX
    : DskInfo ( dsk# --- ) SWPB 1+ 834C ! 0 getSect Buf1 100 getBuf
        CR Buf1 0A ." Disk Name: " TYPE CR ." Total: " Buf1 0A + @ 2-
        DUP U. ." Free: " getFree DUP U.
        ." Used: " - U. CR ; DECIMAL
    : Ftype ( --- ) Buf2 17 + C@ bpr ! Buf2 12 + C@ DUP 8 AND
        prot ! 247 AND CASE 0 OF ." D/F" ENDOF 128 OF
        ." D/V" ENDOF 2 OF ." I/F" ENDOF 130 OF ." I/V"
        ENDOF 1 OF ." PGM" sect @ 256 * Buf2 16 + C@ -DUP IF +
        256 - THEN 0 bpr ! 2 Tab 5 U.R ENDOF ." ???" 0 bpr !
        ENDCASE bpr @ -DUP IF 4 U.R THEN ; R->BASE -->
BLOCK #60
    0 BASE->R DECIMAL
    : DoCAT ( --- ) 0 LC ! 0 Total ! 0 FCount ! Head 1 getSect
        Buf1 256 getBuf Buf1 BEGIN LC @ 20 MOD 19 = IF KEY DROP
        CR Head THEN DUP @ -DUP WHILE getSect Buf2 20 getBuf Buf2 10
        TYPE Buf2 14 + @ DUP sect ! 1+ DUP 0 Tab 4 U.R Total +! 1 Tab
        Ftype prot @ IF 3 Tab ." Y" THEN CR 1 LC +! 1 FCount +! 2+
        REPEAT DROP Head1 FCount @ . ." files" 0 Tab Total @ 4 U.R
        ." sectors" CR ;
    : CAT ( dsk# --- ) BASE->R [COMPILE] DECIMAL
    9 CATPAB VBuf 2- 2 VMBW DskInfo DoCAT R->BASE ;
CR ." n CAT - Catalogs a disk. n = disk #." CR
    ." E.g., 1 CAT catalogs DSK1." CR R->BASE ;S
12
13
14
15
```

BLOCK \#61
0 ( TI Forth disk browser/copier..LES 04DEC2015) BASE->R HEX
1 CR ." loading TI Forth Viewer/Copier"
21154 CONSTANT VTIbuf 0110 VARIABLE TIPAB 1 VARIABLE Dsk
30 VARIABLE outBFL 10 ALLOT 0 VARIABLE curBFL 10 ALLOT
4 : GNUM BL WORD HERE NUMBER DROP ; : getDOidx ( -- lim idx )
5 GNUM GNUM OVER OVER > IF SWAP THEN 1+ SWAP ; : BlkBuf PREV @ 2+ ; : getDsk ( IS:DSKn) BL WORD HERE 4 + C@ 30 - Dsk ! ; : RdErr? ( err - - ) -DUP IF CR ." Disk I/O error " BASE->R [COMPILE] HEX . R->BASE ABORT THEN ; : DSRLNK10 0A 0E SYSTEM 8350 C@ RdErr? ; : getTIblock FLUSH TIPAB VTIbuf 2-2 VMBW VTIbuf 834E ! Dsk @ SWPB 1+ 834C ! 2 SLA BlkBuf DUP 400 + SWAP DO DUP 8350 ! 1+ VTIbuf 2- 8356 ! DSRLNK10 VTIbuf I 100 VMBR 100 +LOOP DROP ; : dnLeft CURPOS @ SCRN_WIDTH @ MOD IF CR THEN ; : EMITG ( $n--$ ) CURPOS @ VSBW CURPOS @ 1+ DUP SCRN_END @ < IF CURPOS ! ELSE DROP CR THEN ; : TYPEG ( addr cnt - - ) -DUP IF OVER + SWAP DO I C@ EMITG LOOP ELSE DROP THEN ; R->BASE -->

```
BLOCK #62
    0 ( TI Forth disk browser/copier..continued) BASE->R HEX ." ."
    1 : dspLine ( line# -- ) 40 * BlkBuf + 40 TYPEG ;
    2 : 64page? CURPOS @ 40 + SCRN_END @ > IF KEY DROP PAGE THEN ;
    3 : TIFBLK ( IS:blk# DSKn ) GNÜM getDsk getTIblock PAGE 10 0 DO
    4 64page? dnLeft I 2 .R ." | " I dspLine PAUSE IF LEAVE THEN
        LOOP ; : TIFIDX ( IS:startblk endblk DSKn) getDOidx getDsk
        PAGE DO I getTIblock 64page? dnLeft I 3 .R ." | " 0 dspLine
        PAUSE IF LEAVE THEN LOOP CR ." ...done" ; : gBFL ( -- ) BL
        WORD HERE outBFL HERE C@ 1+ CMOVE ; : saveCurBFL BPB BPOFF @
        + 9 + DUP VSBR curBFL SWAP 1+ VMBR ; : getBFL TIB @ 0F EXPECT
        0 IN ! gBFL ; : cpyTI2FB ( dstBlk# lim idx -- ) CURPOS @ >R
        DO J CURPOS ! I 3 .R I getTIblock DUP PREV @ ! UPDATE FLUSH
        1+ LOOP DROP R> DROP ;
    : TIF2FBF ( IS:srcStrtBlk srcEndBlk DSKn dstStrtBlk dstBlksFil)
    saveCurBFL getDOidx getDsk GNUM gBFL outBFL (UB) ROT ROT
    cpyTI2FB curBFL (UB) ; R->BASE -->
BLOCK #63
    0 ( TI Forth disk browser/copier..continued) BASE->R HEX ." ."
    : BOXCHRS DATA[ 0000 003C 3C30 3030 0000 00F0 F030 3030 3030
        303C 3C00 0000 3030 30F0 F000 0000 0000 00FC FC00 0000 0000
        00FC FC30 3030 3030 3030 3030 3030 3060 C070 380C 1830 40A0
        A8B4 5414 0800 40C0 4854 F414 0800 40A0 2854 F414 0800 C020
        4834 D414 0800 2060 A8F4 3414 0800 E080 6834 D414 0800 4080
        C8B4 5414 0800 0000 FC00 FC00 FC00 ]DATA C9 DCHAR ;
    D1CD VARIABLE TLDATA DATA[ CDCD CDCE CDCD CDCD D2CD CDCD CDCE
        CDCD CDCD D3CD CDCD CDCE CDCD CDCD D4CD CDCD CDCE CDCD CDCD
        D5CD CDCD CDCE CDCD CDCD D6CD CDCD CDCE CDCD CDCD D7CD CDCD
        ]DATA DROP DROP
    0 VARIABLE TIFblk 0 VARIABLE fbFblk 0 CONSTANT OFFSET
    : WINWID ( -- winwid ) SCRN_WIDTH @ 28 = IF 22 ELSE 40 THEN ;
    13 : CORNERS 3 3 1 0C9 HCHAR 3 14 1 0CB HCHAR 4 WINWID + DUP 3 1
    14 0CA HCHAR 14 1 OCC HCHAR ; : TOPLN ( -- ) OFFSET TLDATA + 4
    15 3 GOTOXY WINWID TYPEG ; R->BASE -->
BLOCK #64
    ( TI Forth disk browser/copier..continued) BASE->R HEX ." ."
    : BOTLN 4 14 WINWID OCD HCHAR ; : SIDELN ( col chr -- ) 4 10
    ROT VCHAR ; : SIDELNS 3 0CF SIDELN WINWID 4 + 0CF SIDELN ;
    : RPT ( chr cnt -- ) 0 DO DUP EMITG LOOP DROP ; : drawScrn
    PAGE 0D8 6 RPT ." TI Forth Block Viewer/Copier" 0D8 6 RPT
    VDPMDE @ 0= IF 0D8 28 RPT THEN ." TI Forth:DSK fbForth:" CR
        ." Block Block" 0 ' OFFSET ! CORNERS TOPLN BOTLN
    SIDELNS SCRN_WIDTH @ DUP 4 * BASE->R DECIMAL 10 0 DO DUP
        CURPOS ! I 3-.R OVER + LOOP R->BASE DROP DROP CR CR
        ." F4:+Block F6:-Block FD:+Panel FS:-Panel
        ." FT:TI# FF:fb# ^F:BlkFil ^S:TI>fb F9:Xit" ; : dspLnSeg
        ( line# -- ) 40 * BlkBuf OFFSET + + WINWID TYPEG ; : dspBlock
        SCRN_WIDTH @ 28 = IF 3 26 OFFSET CASE 00 OF OCF ODO ENDOF OF
        OF 0D0 0D0 ENDOF 1E OF OD0 OCF ENDOF ELSEOF OCF OCF ENDOF
        ENDCASE ROT SWAP SIDELN SIDELN TOPLN THEN 10 0 DO SCRN_WIDTH @
        I 4 + * 4 + CURPOS ! I dspLnSeg LOOP ; R->BASE
```

```
BLOCK #65
    0 ( TI Forth disk browser/copier..continued) BASE->R HEX ." ."
    1 : calcOff ( -1|0|+1 -- ) DUP IF 0F * OFFSET + DUP 0< IF DROP 1E
    THEN DUP 1E > IF DROP 0 THEN THEN ' OFFSET ! ; : dspPanel
    ( +1|-1 -- ) WINWID 22 = IF calcOff dspBlock ELSE DROP THEN ;
    : getCmd ( -- key ) ?KEY DUP IF BEGIN ?KEY 0= UNTIL THEN ;
    : dspBlk# ( n col row -- ) GOTOXY 3 .R ; : get# ( -- n ) TIB @
    3 EXPECT 0 IN ! BL WORD HERE NUMBER DROP ; : getBlk#
    ( min col row -- n ) ROT >R OVER OVER GOTOXY CURPOS @ DUP 3 20
    VFILL CURPOS ! get# DUP R < IF DROP R> ELSE R> DROP THEN DUP
    >R ROT ROT dspBlk# R> ; : nxtTIblk ( +1|-1 -- ) TIFblk +!
    TIFblk @ DUP 8 2 dspBlk# getTIblock 0 calcOff dspBlock ;
    : clrLstLn 0 17 SCRN_WIDTH @ 20 HCHAR 0 17 GOTOXY ;
    : keyPrompt ." ..tap key" KEY DROP clrLstLn ; R->BASE -->
13
14
15
BLOCK #66
    0 ( TI F
    : cmd ( get command key) BEGIN getCmd CASE 02 OF 1 nxtTIblk 0
    ENDOF OC OF TIFblk @ IF -1 nxtTIblk THEN 0 ENDOF 09 OF 1
    dspPanel 0 ENDOF 08 OF -1 dspPanel 0 ENDOF 5D OF 0 8 2 getBlk#
    DUP TIFblk ! getTIblock 0 calcOff dspBlock 0 ENDOF 7B OF 1 18
    2 getBlk# fbFblk ! 0 ENDOF 06 OF 18 1 GOTOXY CURPOS @ DUP 10
    20 VFILL CURPOS ! getBFL outBFL (UB) 0 ENDOF 13 OF fbFblk @
    DUP IF outBFL @ DUP IF SWAP TIFblk @ clrLstLn
    ." How many blocks? " get# OVER + SWAP clrLstLn cpyTI2FB
    ." done" keyPrompt ELSE SWAP DROP THEN THEN 0= IF clrLstLn
    ." fbForth block#|file not set!" keyPrompt THEN 0 ENDOF OF OF
    PAGE 1 ENDOF ELSEOF 0 ENDOF ENDCASE UNTIL ;
    : TIFVU ( IS:blk# DSKn) GNUM DUP TIFblk ! getDsk getTIblock
    VDPMDE @ 2 < IF saveCurBFL BOXCHRS drawScrn OC 1 GOTOXY Dsk @
    . TIFblk @ 8 2 dspBlk# dspBlock cmd curBFL (UB) ELSE CR
    ." TEXT or TEXT80 modes only!" THEN ; R->BASE -->
BLOCK #67
    0 ( TI Forth disk browser/copier..continued) BASE->R HEX ." ."
    1 CR CR ." USAGE:"
    2 CR ." TIFBLK <block#> DSKn"
    3 CR ." ex: TIFBLK 2 DSK2"
    4 CR ." TIFIDX <strtBlock#> <endBlock#> DSKn"
    5 CR ." ex: TIFIDX 9 40 dsk1"
    6 CR ." TIF2FBF <srcStrtBlk#> <srcEndBlk#>"
    CR ." DSKn <dstStrtBlk#> <dstFile>"
    CR ." ex: TIF2FBF 3 6 DSK3 9 DSK1.MYBLOCKS"
    9 CR ." TIFVU <block#> DSKn"
    10 CR ." ex: TIFVU 58 DSK2" CR CR R->BASE ;S
1 1
12
13
14
15
```

```
BLOCK #68
    0
    1
    2
    3
    4
    5
    6
    7
    8
    9
    10
    11
    12
    13
    14
    15
BLOCK #69
    0 \ Compact Flash Mount Utilities for nanoPEB/CF7+...
    1 0 CLOAD CFMOUNT BASE->R HEX CR ." Loading CF Utilities..."
    2 : CF? ( -- flag ) 3FF8 VSBR SWPB 3FF9 VSBR + AA03 = ;
    3 : CFE ( err# -- ) \ display selected error message and abort
    4 CASE
    5 1 OF ." No CF detected!" ENDOF
    6 2 OF ." DSK# must be 1-3!" ENDOF
    7 ENDCASE ABORT ;
    8 : CFVOLS ( -- volDSK1 volDSK2 volDSK3 ) \ get vol#s in DSKs
    9 CF? IF 3FFA PAD 6 VMBR PAD DUP 6 + SWAP DO I @ 2 +LOOP
    10 ELSE 1 CFE THEN ;
    11 : CFMOUNT ( vol# dsk# -- ) \ mount CF vol# in DSK<dsk#>
    12 CF? IF 3FFB SWAP CASE 1 OF ENDOF 2 OF 2+ ENDOF
    13 3 OF 4 + ENDOF ELSEOF 2 CFE ENDOF ENDCASE
    14 OVER SWPB OVER 1- VSBW VSBW ELSE 1 CFE THEN ;
    15
                                    R->BASE CR
BLOCK #70 - 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 KiB 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/ <br> Track | Total Sectors | Capacity |
| :---: | :---: | :---: | :---: | :---: | :---: |
| SSSD | 1 | single | 9 | 360 | 90 KiB |
| DSSD | 2 | single | 9 | 720 | 180 KiB |
| SSDD | 1 | double | 18 | 720 | 180 KiB |
| DSDD | 2 | double | 18 | 1440 | 360 KiB |
| Compact Flash $^{22}$ | 2 | double | 20 | $1600^{23}$ | 400 KiB |

The information in the following sections accrues to all the above formats:

## K. 1 Volume Information Block (VIB)

| Byte \# | $1{ }^{\text {st }}$ Byte | $2{ }^{\text {nd }}$ Byte | Byte \# |
| :---: | :---: | :---: | :---: |
| 0 |  |  |  |
|  | Disk Volume Name (10 characters padded on the right with blanks) |  |  |
| 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 |

[^6]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)



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" ${ }^{25}$. 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:

[^7]Bit \# $\quad$ Description

| $\mathbf{0}$ | Program or Data file ( $0=$ Data; $1=$ Program $)$ |
| :---: | :--- |
| $\mathbf{1}$ | Binary or ASCII data $(0=$ ASCII, DISPLAY file; $1=$ Binary, INTERNAL or program file $)$ |
| $\mathbf{2}$ | Reserved |
| $\mathbf{3}$ | PROTECT flag $(0=$ not protected; $1=$ protected $)$ |
| $\mathbf{4 - 6}$ | Reserved |
| $\mathbf{7}$ | FIXED/VARIABLE flag $(0=$ fixed-length records; $1=$ variable-length records $)$ |

The cluster blocks listed in bytes $28-255$ of the FDR each contain 2 12-bit (3-nybble ${ }^{26}$ ) 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: $n_{2} n_{1}$ Byte 2: $m_{1} n_{3}$ Byte 3: $m_{3} m_{2}$
The actual 12-bit numbers, then, are
Cluster Pointer: $n_{3} n_{2} n_{1} \quad$ Sector Offset: $m_{3} m_{2} m_{1}$
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
$1^{\text {st }}$ Cluster Pointer: 04Dh $\left(77_{10}\right)^{27} \quad$ Record Offset: 5F2h ( $1522_{10}$ )
$2^{\text {nd }}$ Cluster Pointer: $005 \mathrm{~h}\left(5_{10}\right) \quad$ 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 $1600^{\text {th }}$ sector of the disk ( $1^{\text {st }}$ cluster starts in sector 77 and ends 1522 sectors later in sector 1599 ). ${ }^{28}$

[^8]
## 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 radix100 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 63 h ) 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 636363636363 63]:

$$
\begin{aligned}
99.999999999999 \times 100^{63} & =99.999999999999 \times 10^{126} \\
& =9.9999999999999 \times 10^{127}
\end{aligned}
$$

- Largest negative floating point number [hexadecimal: 80 9D 6363636363 63]:

$$
\begin{aligned}
-99.999999999999 \times 100^{63} & =-99.999999999999 \times 10^{126} \\
& =-9.9999999999999 \times 10^{127}
\end{aligned}
$$

- Smallest positive floating point number [hexadecimal: 00010000000000 00]:

$$
01.000000000000 \times 100^{-64}=1.000000000000 \times 10^{-128}
$$

- Smallest negative floating point number [hexadecimal: FF FF 0000000000 00]:

$$
-01.000000000000 \times 100^{-64}=-1.000000000000 \times 10^{-128}
$$

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.

## Appendix M Bug Fixes as of fbForth 2．0：9

The following bug fixes have been made over a period of time and are in no particular time order：
The insert－blank－line function，＜CTRL＋8＞，in the 40／80－column editor would not blank the entire new line if the cursor were not located in the first column．
（ The character－copy function in the 40／80－column editor would cause fbForth $\mathbf{2 . 0}$ to crash if the line－insertion and line－deletion functions were used on the last line of a block． The problem was not testing for a copy－count of 0 before copying the first character， causing the count to pass 0 before the test if the function was passed a count of 0 ，which it is on the last line．

学 SGN would yield +1 for -32768 （ 8000 h ），the largest single－precision（16－bit）negative number possible on the TI－99／4A．
SSDT was improperly setting the address of the Sprite Pattern Descriptor Table．SSDT is the easiest way for a user to change the Sprite Pattern Descriptor Table in graphics mode to a different location from the the default 800 h ．The default， 800 h ，is coincident with the text Pattern Descriptor Table．It is easy enough to change the SSDT in code，but it is not trivial．Besides，SSDT not only changes the user variable read by the constant， SPDTAB，but also changes VDP register \＃6 to the proper value and executes DELALL to initialize sprites．

美 SPRPUT was setting the $x$ position to 255 （rightmost position）if $y$ was 0 ．
学 MOTION was setting the $x \mid y$ vector to -1 if the $y \mid x$ vector was negative．
If sprite automotion was not stopped in Graphics mode，blinking text appeared in Text， Text80，Bitmap and Split modes．Automotion was not stopped when changing VDP modes．For some reason，if sprite automotion is enabled and sprites are left defined， Text80，Bitmap and Split modes show blinking areas on the screen that correspond to those sprites，particularly those defined with patterns in the text PDT area．

学 BSAVE was not explicitly saving the pointer to the last word in each of the Forth and Assembler vocabularies．
学 BSAVE and BLOAD were not saving and loading，respectively，the vocabulary link fields of the Forth and Assembler vocabularies．

DELALL was only marking the first 8 sprites as deleted，i．e．，$y=\mathrm{D} 0 \mathrm{~h}$ ，when it should have been doing it for all 32 ！The upshot of this bug was that，as soon as sprite \＃7 was defined，all of the remaining sprites were suddenly defined as char 0 ，transparent and positioned at $(0,0)$ ！
（ CPYBLK（loaded from FBLOCKS）was copying blocks from previous blocks files if the corresponding blocks were in block buffers．EMPTY－BUFFERS was added to fix it．

学 M／was improperly setting the sign of the remainder to that of the divisor by default．


[^0]:    3 This loop may be exited by executing R> DROP one level below.

[^1]:    ASCII Collating Sequence: ! " \# \$ \% \& ( ) * + , . / digits : ; < = > ? @ HLPHA [ \ ] ^ _`alpha \{ | \}~

[^2]:    ASCII Collating Sequence: ! " \# \$ \% \& ( ) * + , . / digits : ; < = > ? @ ALPHA [ \ ] ^ _`alpha \{ | \}~

[^3]:    ASCII Collating Sequence: ! " \# \$ \% \& ( ) * + , . / digits : ; < = > ? @ ALPHA [ \ ] ^ _`alpha \{ | \}~

[^4]:    ASCII Collating Sequence: ! " \# \$ \% \& ( ) * + , . / digits : ; < = > ? @ HLPHA [ \ ] ^ _`alpha \{ | \}~

[^5]:    ASCII Collating Sequence: ! " \# \$ \% \& ( ) * + , . / digits : ; < = > ? @ ALPHA [ \ ] ^ _`alpha \{ | \}~

[^6]:    22 This is a third-party peripheral expansion device with 400 KiB virtual disks using Compact Flash memory on devices named nanoPEB and CF7+ (see website: http://webpages.charter.net/nanopeb/)

    231600 sectors is the maximum possible number of sectors that can be managed by the current specification.

[^7]:    24 A zero value for the EOF Offset indicates 256 bytes in the last sector.
    25 Software Specifications for the 99/4 Disk Peripheral (March 28, 1983), p. 19.

[^8]:    26 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.

    27 The subscript, 10 , indicates base 10 (decimal).
    28 This example is taken from one of my (Lee Stewart's) Compact Flash volumes.

