fbForth 1.0

A File-Based Implementation of TI Forth

Lee Stewart

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

Original Dedication of TI Forth

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

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

—*from original TI Forth Manual*

Table of Contents

1 Introduction

1.1 Original Introduction to TI Forth

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

The address for the Forth Interest Group is:

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

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

> *Starting FORTH (1st Ed.)*
> by Leo Brodie
> published by Prentice Hall
>
> *Using FORTH*
> by Forth Inc.
>
> *Invitation to FORTH*
> by Katzan
> published by Petrocelli Books

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

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

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

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

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

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

—from original TI Forth Manual

1.2 Author's Introduction

My source for the text of the original *TI Forth Instruction Manual*, much of which is included in this document, was a series of sixteen files named A, B, C, …, P in TI-Writer format, which I had purchased from the MANNERS (Mid-Atlantic Ninety-NinERS) TI Users Group shortly after TI put TI Forth into the public domain. I do not know who deserves the credit for originating these files; but, it was always my understanding they came from TI and that the printed document we all received with the TI Forth system was prepared in and printed from TI Writer. However, the A – P files have differences from the original printed document. I have taken the liberty of incorporating most of the original into this **fbForth 1.0***: A File-Based Implementation of TI Forth*.

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

Though, in coding **fbForth**, 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
January, 2014
Silver Run, MD

1.3 *Starting* fbForth

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

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

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

The **fbForth** system consists of two files on the system disk, *viz.*, FBFORTH and FBLOCKS. FBFORTH is the program file in compressed object format and FBLOCKS is the system blocks file.

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

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

The **fbForth** welcome screen will display, “Type MENU for load options.” Loading a block in the “Start Block” column below loads all routines necessary to perform a particular group of tasks:

Start Block	Loads Forth Words Necessary to:	Chapter
13	Run the text-mode, 40/80-column **fbForth** editor.	3
19	Copy a range of blocks[1] to the same or another blocks file.	5
21	Execute **DUMP** and **VLIST**.	5
23	Trace the execution of Forth words.	5
24	Use floating-point arithmetic.	7
4	Change display screen to any of the 76 available VDP modes.	6
30	Change display screen to Text or Text80 mode.	6
31	Change display screen to Graphics mode.	6
32	Change display screen to Multicolor mode.	6
33	Change display screen to Graphics2 (bitmap) mode.	6
34	Change display screen to either of the two Split-screen modes.	6
47	Use the file I/O capabilities of the TI-99/4A.	8
51	Send output to an RS232 (or similar) device.	8
6	Run the 64-column **fbForth** editor.	3
53	Write routines in **fbForth** TMS9900 Assembler.	9
36	Use the graphics capabilities of the TI-99/4A.	6
59	Save dictionary overlays to diskette.	11
20	Access the **fbForth** equivalents of TMS9900 Assembler mnemonics for the CRU: LDCR, STCR, SBO, SBZ and TB.	11

To load a particular package, simply type its block number, exactly as it appears in the list, followed by **LOAD** . For example, to load the graphics package, type **36 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] A Forth block (TI Forth uses ‘screen’) consists of 16 lines of 64 characters for a total of 1024 characters. When a Forth block is loaded from a blocks file, 1024 characters are copied from the file into a RAM block buffer. This is explained in more detail later in this document.

1.4 **fbForth** *Terminal Response*

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

```
ok:n
```

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

```
2 4 6  ok:3
```

1.5 Changing How **fbForth** *Starts*

When **fbForth** boots up, it always looks for DSK1.FBLOCKS and complains if it does not find it. Upon finding it, **fbForth** 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** to load it after it boots up.

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**. Those with no Forth experience should begin by reading a book such as *Starting FORTH, (1st Ed.)* by Leo Brodie. Appendix C “Differences between Starting FORTH (1st Ed.) and fbForth” is designed to be used side by side with *Starting FORTH, (1st Ed.)* and lists the differences between the Forth language described in the book (poly-Forth) and **fbForth**.

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

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

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

(n_1 n_2 --- n_3)

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

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

(*n* ---) (R: --- *n*)

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

SOME SYMBOLS USED IN THIS DOCUMENT

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

2.1 Stack Manipulation

The following are the most common stack manipulation cells:

`-DUP`	(n --- $n\ n$ | n)	Duplicate only if non-zero
`.S`	(---)	Non-destructively display stack contents
`>R`[2]	(n ---) (R: --- n)	Move top item on stack to return stack
`DEPTH`	(--- s*tack-depth*)	Number of cells on parameter stack
`DROP`	(n ---)	Discard top of stack
`DUP`	(n --- $n\ n$)	Duplicate top of stack
`OVER`	($n_1\ n_2$ --- $n_1\ n_2\ n_1$)	Make copy of second item on top
`R`	(--- n) (R: n --- n)	Copy top item of return stack to stack
`R>`	(--- n) (R: n ---)	Move top item on return stack to stack
`ROT`	($n_1\ n_2\ n_3$ --- $n_2\ n_3\ n_1$)	Rotate third item to top
`SP!`	(---)	Clear stack, resetting it to its base **`S0`**
`SWAP`	($n_1\ n_2$--- $n_2\ n_1$)	Exchange top two stack items

2.2 Arithmetic and Logical Operations

The following are the most common arithmetic and logical operations:

`*`	($n_1\ n_2$ --- n_3)	Multiply
`*/`	($n_1\ n_2\ n_3$ --- *quot*)	Like **`*/MOD`** but giving *quot* only
`*/MOD`	($n_1\ n_2\ n_3$ --- *rem quot*)	$n_1 * n_2 / n_3$ with 32 bit intermediate
`+`	($n_1\ n_2$ --- n_3)	Add
`-`	($n_1\ n_2$ --- n_3)	Subtract ($n_1 - n_2$)
`/`	($n_1\ n_2$ --- n_3)	Divide n_1 by n_2 and leave quotient n_3
`/MOD`	($n_1\ n_2$ --- *rem quot*)	Divide n_1 by n_2 giving remainder & quotient
`1+`	(n_1 --- n_2)	Increment by 1
`2+`	(n_1 --- n_2)	Increment by 2
`1-`	(n_1 --- n_2)	Decrement by 1
`2-`	(n_1 --- n_2)	Decrement by 2
`ABS`	(n --- $\|n\|$)	Absolute value

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

AND	(n_1 n_2 --- n_3)	Bitwise logical AND n_3
D+	(d_1 d_2 --- d_3)	Add double precision numbers
DABS	(d --- $\|d\|$)	Absolute value of 32-bit number
DMINUS	(d_1 --- d_2)	Leave two's complement of 32-bits
MAX	(n_1 n_2 --- n_1 | n_2)	Maximum
MIN	(n_1 n_2 --- n_1 | n_2)	Minimum
MINUS	(n_1 --- n_2)	Leave two's complement
MOD	(n_1 n_2 --- n_3)	Modulo (remainder from n_1 / n_2)
OR	(n_1 n_2 --- n_3)	Bitwise logical OR n_3
SGN	(n --- -1 | 0 | +1)	Sign of n as -1 | 0 | +1
SLA	(n_1 n_2 --- n_3)	Shift n_1 left arithmetic n_2 bits giving n_3
SRA	(n_1 n_2 --- n_3)	Shift n_1 right arithmetic n_2 bits giving n_3
SRC	(n_1 n_2 --- n_3)	Shift n_1 right circular n_2 bits giving n_3
SRL	(n_1 n_2 --- n_3)	Shift n_1 right logical n_2 bits giving n_3
SWPB	(n_1 --- n_2)	Swap the bytes of n_1 producing n_2
XOR	(n_1 n_2 --- n_3)	Bitwise logical exclusive OR n_3
U*	(u_1 u_1 --- ud_2)	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:

<	(n_1 n_2 --- *flag*)	True if n_1 less than n_2 (signed)
=	(n_1 n_2 --- *flag*)	True if top two numbers are equal
>	(n_1 n_2 --- *flag*)	True if n_1 greater than n_2
0<	(n --- *flag*)	True if top number is negative
0=	(n --- *flag*)	True if top number is 0 (*i.e.,* NOT)
U<	(u_1 u_2 --- *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
?	(*addr* ---)	Print the contents of address (same as **@ .**)
@	(*addr* --- *n*)	Replace word address by its contents
C!	(*b addr* ---)	Store *b* at address (store a byte)
C@	(*addr* --- *b*)	Fetch the byte at *addr*
CMOVE	(*from_addr to_addr u* ---)	Block move *u* bytes.
BLANKS	(*addr u* ---)	Fill *u* bytes with blanks beginning at *addr*
ERASE	(*addr u* ---)	Fill *u* bytes beginning at *addr* with 0s
FILL	(*addr u b* ---)	Fill *u* bytes with *b* beginning at *addr*
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** while others are from nonresident words that are part of the default system blocks file, FBLOCKS.

2.5.1 IF … THEN

IF … THEN		**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** .
IF	(*flag* ---)	
ENDIF		Synonym for **THEN** .

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

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

2.5.2 IF … ELSE … THEN

IF … ELSE … THEN

IF (*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** resident dictionary. **CLOAD** loads a block from the current blocks file only if the word that follows **CLOAD** in the input stream cannot be found in the dictionary. It is a state-smart word that can be used in a word definition as well as on the command line. It is used in the following way:

20 CLOAD MYWORD ,

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

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

2.5.3 BEGIN … AGAIN

BEGIN … AGAIN

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

2.5.4 BEGIN … UNTIL

BEGIN … UNTIL

UNTIL (*flag* ---)

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

END

Synonym for **UNTIL** .

The following example from FBLOCKS is from block 22 of the memory dump utility. **VLIST** lists words in the **CONTEXT** vocabulary starting with the last defined word pointed to by **CONTEXT** and following the linked list of words and vocabularies until it finds the first word at the top of the chain that has a pointer (link field address or *lfa*) of 0. This topmost word will always be **EXECUTE** in **fbForth**. See Chapter 12 "fbForth Dictionary Entry Structure" for an explanation of **fbForth** word fields and their abbreviations (*lfa*, *nfa*, *cfa* and *pfa*). If you know the *pfa*, you can get the other three field addresses for a given word. You can get the *pfa* if you know the *nfa*. These facts are used in the following example:

[3] This loop may be exited by executing **R> DROP** one level below.

```
: VLIST                           « Define VLIST to list the CONTEXT vocabulary.
   ( --- )                        « Takes no parameters and leaves none.
   80 OUT !                       « Store maximum expected  character count in OUT .
   CONTEXT @ @                    « Get nfa of last defined word in CONTEXT vocabulary.
   0 SWAP                         « Start word counter at 0 and swap nfa to top of stack.
   BEGIN                          « Start indefinite loop.
      DUP C@ 3F AND               « Dup nfa.  Get length byte’s least-significant 5 bits.
      OUT @ +                     « Add name length to OUT .
      SCRN_WIDTH @ 3 -            « Get screen width – 3 for spaces and end of line.
      > IF                        « Will line be too long?
         CR 0 OUT !               « Yes.  Go to next line and zero character count.
      THEN
      DUP ID.                     « Dup nfa.  Display name.
      SWAP 1+ SWAP                « Get word count to top.  Increment it.  Swap nfa back.
      PFA LFA @                   « Get lfa from pfa.  Get next word’s nfa from lfa.
      SPACE                       « Emit a space (updates OUT in the process).
      DUP 0=                      « Dup new nfa.  Leave true if 0, else false.
      PAUSE                       « Pause if keystroke.  Return true if <BREAK>, else false.
   OR UNTIL                       « OR above flags.  Exit loop if true, else repeat.
   DROP CR . ." words listed"     « Drop leftover nfa.  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 non-zero (*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                     « Define FIREDOWN to display loop iterations between press
                                 and release of left joystick’s fire button.
   ( --- )                     « No parameters in or out.
   BEGIN                       « Start indefinite loop awaiting fire button press.
      1 JOYST DROP DROP        « Get state of joystick/keyboard #1.  Save only char value.
   18 = UNTIL                  « Repeat loop until char is fire-button value (18).
   0                           « Initialize counter on stack.
   BEGIN                       « Start indefinite loop awaiting release of fire button.
      1 JOYST DROP DROP        « Get state of joystick/keyboard #1.  Save only char value.
   18 = WHILE                  « Continue with loop while char value = 18, else exit.
      1+                       « Increment loop counter on stack.
   REPEAT                    « Repeat loop.
   CR . ."  iterations."     « Display # of iterations on next screen line.
;
```

2.5.6 DO … LOOP

DO … 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.
DO	(*lim strt* ---)	
I	(--- *n*)	Used between **DO** and **LOOP**. Places value of loop counter on stack.
J	(--- *n*)	Used when **DO LOOP**s are nested. Places value of next outer loop counter on the stack.
LEAVE	(---)	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 8x8 array *addr* of numbers to search and a number *n* to match. The result will be only a *false* flag if there is no match, but a *true* flag, row *r* and column *c* of the array if there is a match.

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

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

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

The following example from FBLOCKS is from block 41 of the graphics primitives using decimal numbers instead of hexadecimal. It initializes the screen in multicolor graphics mode.

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

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

2.5.7 DO … +LOOP

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

DO (*lim strt* ---)

+LOOP (*n* ---)

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

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

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

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

2.5.8 CASE … OF … ENDOF … ENDCASE

```
CASE
      n1 OF … ENDOF
      n2 OF … ENDOF
      …
      nm OF … ENDOF
      …
ENDCASE

      CASE    ( n --- )
```

Looks for a number (n_1, n_2, …, n_m) matching n. If there is a match, executes the code between the **OF … 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.

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

The following example from FBLOCKS is from block 39 of the graphics primitives. It uses the console’s keyboard scanning routine KSCAN to check for joystick and fire-button status of left and right joysticks or corresponding keys on left and right sides of the keyboard:

```
HEX                              « Use radix 16.
: JKBD                           « Define JKBD to scan for joystick input.
   ( kbd --- chr xst yst )       « In: Keyboard kbd = 1 or 2. Out: Value chr of key
                                   struck, joystick x-status xst and y-status yst.
   8374 C!                       « Store kbd for keyboard # to scan.
   ?KEY DROP 8375 C@             « Check for keystroke.  DROP char returned and get
                                   KSCAN’s returned value.
   DUP 12 =                      « Duplicate chr and check for fire button.
   OVER 0FF =                    « Duplicate chr again and check for “no keystroke”.
   OR IF                         « Was fire-button or no key depressed?
      8377 C@ 8376 C@            « Yes.  Leave xst and yst on stack on top of chr.
   ELSE                          « No.
      DUP                        « Duplicate chr for input to CASE .
      CASE
         04 OF 0FC   4 ENDOF     « chr = 4 (NW)?   xst = FCh,  yst = 4
         05 OF   0   4 ENDOF     « chr = 5 (N)?    xst = 0,    yst = 4
         06 OF   4   4 ENDOF     « chr = 6 (NE)?   xst = 4,    yst = 4
         02 OF 0FC   0 ENDOF     « chr = 2 (W)?    xst = FCh,  yst = 0
         03 OF   4   0 ENDOF     « chr = 3 (E)?    xst = 4,    yst = 0
         0F OF 0FC 0FC ENDOF     « chr = Fh (SW)?  xst = FCh,  yst = FCh
         00 OF   0 0FC ENDOF     « chr = 0 (S)?    xst = 0,    yst = FCh
         0E OF   4 0FC ENDOF     « chr = Eh (SE)?  xst = 4,    yst = FCh
         DROP DROP 0 0 0 0       « Illegal chr:  Drop both copies and leave four 0s.
      ENDCASE                    « Remove top 0, leaving three 0s.
   THEN
   0 8374 C!                     « Restore previous keyboard #.
;
```

Other more extensive examples of the **CASE** structure appear in FBLOCKS in both the 64-column editor (**EDT** in block 12) and the 40/80-column editor (**VED** in block 18). They each are set up with an infinite **BEGIN … AGAIN** loop that continuously monitors the keyboard until the exit key, ***<FCTN+9>***, is struck. ***<FCTN+9>***’s ASCII value is **0Fh**, so the **OF** clause that follows **0Fh** executes its contents, ultimately executing **QUIT** to get back to the terminal command line interpreter.

2.6 Input and Output to/from the Terminal

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

.	(*n* ---)	Print a signed number
."	(---)	Print a string terminated by **"**
.R	(n_1 n_2 ---)	Print n_1 right-justified in field of width n_2

?KEY	(--- *n*)	Read keyboard. If no key pressed, *n* = 0 else *n* = ASCII keycode.
?TERMINAL	(--- *flag*)	Test if ***<BREAK>*** (***<CLEAR>*** on TI-99/4A) pressed
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 !**
COUNT	(*addr* --- *addr*+1 *n*)	Move length byte from a packed character string[4] at *addr* to stack and increment *addr*—suitable for **TYPE**
CR	(---)	Perform a Carriage Return + Line Feed
D.	(*d* ---)	Print double-precision number
D.R	(*d n* ---)	Print double-precision number right-justified in field of width *n*
DECIMAL	(---)	Sets the base to Decimal (Base 10)
EMIT	(*c* ---)	Type character from stack to terminal
EXPECT	(*addr n* ---)	Read *n* characters (or until **CR**) from terminal to *addr*
HEX	(---)	Sets the base to Hexadecimal (Base 16)
KEY	(--- *c*)	Wait for a keystroke and put its ASCII value on the stack.
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*

2.7 Numeric Formatting

Advanced numeric formatting control is possible with the following words:

NUMBER	(*addr* --- *d*)	Convert string at *addr* to *d* number
<#	(---)	Start output string conversion
#	(d_1 --- d_2)	Convert next, least-significant digit of d_1 leaving d_2
#S	(*d* --- 0 0)	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* --- *addr u*)	Terminate conversion, ready for **TYPE**

[4] A packed character string is a string of characters with a leading length byte. Several **fbForth** words expect or produce such strings.

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 Glossary" for more information.

2.8 Block-Related Words

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

B/BUF	(--- *n*)	Constant: Block size in bytes (always 1024 in **fbForth**)
BLK	(--- *addr*)	User variable containing current block number (contains 0 for terminal input)
BLOCK	(*n* --- *addr*)	Leave address of block *n*, reading it from the current blocks file if necessary
CLEAR	(*n* ---)	Fill block *n* with blanks
CLR_BLKS	(n_1 n_2 ---)	**CLEAR** a range of blocks from block n_1 to block n_2
CPYBLK	(---)	Copy a range of blocks from one blocks file to the same or a different blocks file from information in input stream
EMPTY-BUFFERS	(---)	Erase all buffers
FLUSH	(---)	Write all updated (dirty) buffers to disk
LIST	(*n* ---)	List block *n* to terminal
LOAD	(*n* ---)	Interpret block *n*
MKBFL	(---)	Create a blocks file from string and number in input stream
SCR[5]	(--- *addr*)	User variable containing block number most recently referenced by **LIST** or **EDIT**
UPDATE	(---)	Mark last buffer accessed as updated (dirty)
USEBFL	(---)	Select a different blocks file from input stream

[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.

2.9 Defining Words

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

: xxx	(---)	Begin colon definition of **xxx**[6]
;	(---)	End colon definition
VARIABLE xxx	(*n* ---)	Create variable with initial value *n*
xxx	(--- *addr*)	Returns address when executed
CONSTANT xxx	(*n* ---)	Create constant with value *n*
xxx	(--- *n*)	Returns *n* when executed
CODE xxx … NEXT,	(---)	Define assembly language primitive named **xxx**
ASM: xxx … ;ASM	(---)	Ditto: **ASM:** ≡ **CODE** and **;ASM** ≡ **NEXT,**
: xxx <BUILDS … **;CODE … NEXT,**		Create new defining word **xxx** with execution-time assembly/machine code routine
: xxx <BUILDS … **DOES>ASM: … ;ASM**		Ditto: **DOES>ASM:** ≡ **;CODE** and **;ASM** ≡ **NEXT,**
: xxx <BUILDS … **DOES> … ;**		Create new defining word **xxx** with execution-time high level Forth routine

2.10 Miscellaneous Words

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

' xxx	(--- *addr*)	Leave parameter field address (*pfa*) of **xxx** . If compiling, compile address. (tick)
(	(---)	Begin comment. Terminated by **)**
,	(*n* ---)	Compile *n* into the dictionary (comma)
ABORT	(---)	Error termination
ALLOT	(*n* ---)	Leave *n*-byte gap in dictionary
CONTEXT	(--- *addr*)	Leave address of pointer to context vocabulary (searched first)
CURRENT	(--- *addr*)	Leave address of pointer to current vocabulary (new definitions placed there)
DEFINITIONS	(---)	Set **CURRENT** to **CONTEXT**
FORGET xxx	(---)	Forget all definitions back to and including **xxx**[6]
FORTH	(---)	Set **CONTEXT** to main Forth vocabulary

[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 **FORGET**ting it.

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

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

3 How to Use the fbForth Editors

Words introduced in this chapter:

CLEAR
CLR_BLKS
CPYBLK
ED@
EDIT
FLUSH
MKBFL
TEXT
TEXT80
USEBFL
WHERE

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 · 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
```

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
USEBFL DSK2.MYBLOCKS
```

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

3.1 Forth Block Layout Caveat

As indicated above, Forth blocks are laid out in 16 lines of 64 characters each. However, you should be aware that the lines have no actual delimiters, *i.e.*, there are no carriage-return or line-feed characters at the end of a Forth-block line. This means that one line wraps around to the next line with no intervening white-space such that a word ending on one line will be concatenated with a word that starts on the next line if there is no intervening space. This will usually be nonsense to the system and generate an error message when the block is loaded,

[7] DF128 refers to the file format: **D**isplay data type, **F**ixed 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.

indicating that the unintended word has not been defined. Worse, it can result in an unintended existing word such as **-DUP** instead of **- DUP** or **+LOOP** instead of **+ LOOP** .

3.2 The Two fbForth Editors

There are two Forth editors available in the **fbForth** system blocks file, FBLOCKS. The first, which is loaded by **13 LOAD** , 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. 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; however, the characters are very small. It will be referred to as the 64-column editor.

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

3.3 Editing Instructions

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
```

will prepare block 1 for use by the editor.

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

```
1 5 CLR_BLKS
```

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

```
1 EDIT
```

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

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

```
ED@
```

[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.

will return you to the last block you were editing.

Upon entering the editor, the cursor is located in column 0 of line 0. It is customary to use line 0 for a comment describing the contents of that block. Type a comment that says "**PRACTICE BLOCK**" or something to that effect. Do not forget that all comments must begin with a '**(** '[11] and end with a '**)**'.

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

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

key	function
<FCTN+S>, (←)	moves the cursor one position to the left.
<FCTN+D>, (→)	moves the cursor one position to the right.
<FCTN+E>, (↑)	moves the cursor up one position.
<FCTN+X>, (↓)	moves the cursor down one position.
<DELETE> (***<FCTN+1>***)	deletes the character on which the cursor is placed.
<INSERT> (***<FCTN+2>***)	inserts a space to the left of the cursor moving the rest of the line right one space. Characters may be lost off the end of the line.
<AID> (***<FCTN+7>***)	erases from the cursor to the end of a line and saves the erased characters in **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.
<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. **64-column editor:** places the cursor in the upper left corner
<ERASE> (***<FCTN+3>***) ***<REDO>*** (***<FCTN+8>***)	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.
<CTRL+8>	will insert a blank line just above the line the cursor is on.
<CTRL+V>	will tab forward by words.
<FCTN+V>	will tab backwards by words.

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

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

The Forth editor allows you to move forward or backward a 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.

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 on a single block. This aids in clarity and the good Forth-programming practice of keeping word definitions short.

3.4 Changing Foreground/Background Colors of 64-Col Editor

The black-on-gray color scheme of the 64-column editor can be changed to whatever foreground/background pair you would like by changing block 33 of FBLOCKS, where **GRAPHICS2** is defined. You may wish to change it to dark blue on white. To effect that, change the color table fill hexadecimal value **010** (black on transparent) on line 7 to **040** (dark blue on transparent) and **0FE** (white on gray) on line 13 to **0FF** (white on white)—the left nybble doesn't matter except in text mode. The only problem with these changes to bitmap mode is that they also affect the colors used in bitmap mode outside the 64-column editor. The original values for the above two bytes were **0F0** and **0F1** for a white-on-black bitmap.

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

You can also change the default colors for text mode to something other than dark blue on white when typing **TEXT** after leaving the 64-column editor by changing **04F** on line 9 of block 30 to another color pair, with the foreground color in the left nybble and the background color in the right nybble, *e.g.*, **01E** for black on gray. Again, the original byte was **0F4**, white on dark blue.

3.5 Block-Copying Utility

You can copy a range of blocks to the same or another blocks file with **CPYBLK** . This utility is not part of the resident dictionary, so you will need to load block 19 (**19 LOAD**) from FBLOCKS. Typing **MENU** will show you this option as well as ensure that FBLOCKS is the current blocks file. Usage instructions are displayed after **CPYBLK** is loaded:

```
19 LOAD

CPYBLK copies a range of blocks to the
same or another file, e.g.,
      CPYBLK 5 8 DSK1.F1 9 DSK2.F2
will copy blocks 5-8 from DSK1.F1 to
DSK2.F2 starting at block 9.
 ok:0
```

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

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

4 Memory Maps

The following diagrams illustrate the memory allocation in the TI-99/4A system. For more detailed information, see the *Editor/Assembler Manual*.[12]

The VDP memory can be configured in many ways by the user. The **fbForth** 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 transcendental functions in all modes for a rollout area. If transcendentals are used during Graphics2 (bitmap) or Text80 modes, this portion of the color or screen image tables must be saved by the user before using the transcendental function and restored afterward. Note that the VDP RAM is accessed from the 9900 only through a memory mapped port and is not directly in the processor's address space.

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

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

4.1 VDP Memory Map

Address						Address
`0000h`	Graphics & Multicolor Screen Image Table *bytes:*	`300h`	Text Screen Ta	Mode Image ble	Bitmap Color Table `1800h`	`0000h`
`0300h`	Sprite Attribute List	`80h`	40 Columns `TEXT`	80 Columns `TEXT80`		
`0380h`	Color Table	`20h`				
`03A0h`	Unused	`20h`	`3C0h`	`780h`		
`03C0h`	VDP Rollout Area	`20h`	*[Transcendental function use: Save/restore memory to avoid corruption of bitmap and 80-column text modes]*			
`03E0h`	Value Stack	`80h`				
`0460h`	PABS etc.	`320h`				
`0780h`	Sprite Motion Table	`80h`	*[Value Stack for* `TEXT80`*]*			

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

Address			Address
0800h	Pattern & Sprite Descriptor Tables 0 – 127 400h		
0C00h	128 – 255 400h		
1000h	**fbForth**'s Disk Buffer 80h		
1080h	**fbForth** System Messages 11Ch		
119Ch	True Lowercase Characters F8h		
1294h	Zero Pattern Patch 4		
1298h	PAB for Current Blocks File 46h		
12DEh	PAB for Second Blocks File 46h		
1324h	Default System Blocks File Path 38h		
135Eh	Unused *[PABS points here for* TEXT80*]* 227Ah		
		Bitmap Screen Image Tab. 300h	1800h
		Sprite Attribute List 80h	1B00h
		User PABs, *etc.* E2h	1B80h
		Stack for VSPTR 40h	1C62h
		fbForth's Disk Buffer 80h	1CA2h
		fbForth System Messages 11Ch	1D22h
		True Lowercase Characters F8h	1E3Eh
		Zero Pattern Patch 4h	1F36h
		PAB for Current Blocks File 46h	1F3Ah
		PAB for Second Blocks File 46h	1F80h
		Def. Sys. Blocks File Path 38h	1FC6h
		Bitmap Pattern Descriptor Table 1800h	2000h
35D8h	Disk Buffer Region for 3 Simultaneous Disk Files A28h		
		Sprite Descriptor Table 1DEh	3800h
		Disk Buffer Region: 2 Files 622h	39DEh
3FFFh			3FFFh

4.2 CPU Memory

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

4.3 CPU RAM Pad

Address[13]	
8300h 831Fh	**fbForth**'s Workspace (see § 9.2)
8320h 832Dh	–FREE– Eh
832Eh 8347h	**fbForth**'s Inner Interpreter, etc.
8348h 8349h	–FREE– 2
834Ah 8351h	FAC (Floating Point Accumulator)
8354h	Floating Point Error
8355h	Floating Point String↔Number Conversion Options use these 3 bytes
8356h 8357h	Subroutine Pointer for DSRs
835Ch 8363h	ARG (Floating Point Argument Register)
836Eh 836Fh	VSPTR (Value Stack Pointer)
8370h 8371h	Highest Available Address of VDP RAM
8372h	Least Significant Byte of Data Stack Pointer
8373h	Least Significant Byte of Subroutine Stack Pointer
8374h	Keyboard Number to be Scanned
8375h	ASCII Keycode Detected by Scan Routine
8376h	Joystick Y-status
8377h	Joystick X-status
8379h	VDP Interrupt Timer
837Ah	Number of Sprites that can be in Automotion
837Bh	VDP Status Byte Bit 0[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 5th Sprite on a Line
837Ch	GPL Status Byte Bit 0 High Bit Bit 1 Greater than Bit Bit 2 On when Keystroke Detected (COND) Bit 3 Carry Bit Bit 4 Overflow Bit
837Dh	VDP Character Buffer
837Eh	Current Screen Row Pointer
837Fh	Current Screen Column Pointer
8380h	Default Subroutine Stack
83A0h	Default Data Stack
83C0h	Random Number Seed (Begin Interrupt Workspace)
83C2h	Flag Bit 0 Disable All of the Following Bit 1 Disable Sprite Motion

[13] Locations omitted are not used by **fbForth**, but may be used by system routines.

[14] Bit 0 = high order bit.

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

4.4 Low Memory Expansion

2000h	XML Vectors	0010h bytes
2010h	**fbForth** Disk Buffers	1414h
3424h	99/4 Support for **fbForth**	05E6h
3A0Ah	User Variable Area	0080h
3A8Ah	Message Table Index	001Ah
3AA4h	Assembler Support	02B2h
3D56h 3FFFh	↑ Return Stack	02AAh

4.5 High Memory Expansion

A000h	Resident **fbForth** Vocabulary 20C8h	
C0C8h	User Dictionary Space ↓ ↑ Parameter Stack	3ED8h
FFA0h FFF1h	Terminal Input Buffer	0052h

5 System Synonyms and Miscellaneous Utilities

Words introduced in this chapter:

'

,

.S

: (traceable)

C,

CLS

DSRLNK

DUMP

GPLLNK

INDEX

MYSELF

RANDOMIZE

RND

RNDW

SEED

TRACE

TRIAD

TRIADS

TROFF

TRON

UNTRACE

VAND

VFILL

VLIST

VMBR

VMBW

VMOVE

VOR

VSBR

VSBW

VWTR

VXOR

XMLLNK

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

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

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

5.1 System Synonyms

The system synonyms are part of the resident dictionary in **fbForth**. These utilities allow you to

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

5.1.1 VDP RAM Read/Write

The first group of instructions enables you to read from and write to VDP RAM. Each of the following **fbForth** words implements the Editor/Assembler (E/A) utility with the same name.

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

VSBW (*b vaddr* ---)

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

base	*byte*	*vaddr*	instr
HEX	**A3**	**380**	**VSBW**

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

VMBW (*addr vaddr count* ---)

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

base	*addr*	*vaddr*	*count*	instr
HEX	**PAD**	**808**	**4**	**VMBW**

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

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

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

base	*vaddr*	instr
HEX	**781**	**VSBR**

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

VMBR (*vaddr addr count* ---)

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

base	*vaddr*	*addr*	*count*	instr
HEX	**300**	**PAD**	**20**	**VMBR**

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

VFILL (*vaddr count byte* ---)

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

base	*vaddr*	*count*	*byte*	instr
HEX	**300**	**20**	**0**	**VFILL**

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

VMOVE ($vaddr_1$ $vaddr_2$ *count* ---)

Copies *count* bytes from one location ($vaddr_1$) in VDP RAM to another ($vaddr_2$).

base	$vaddr_1$	$vaddr_2$	*count*	instr
HEX	**1500**	**1640**	**100**	**VFILL**

copies 256 (**100h**) bytes from $vaddr_1$ to $vaddr_2$. 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.

5.1.2 Extended Utilities: GPLLNK, XMLLNK and DSRLNK

The next group of instructions allows you to implement the Editor/Assembler instructions GPLLNK, XMLLNK and DSRLNK. To assist the user, the Forth instructions have the same names as the Editor/Assembler utilities. Consult the *Editor/Assembler Manual*, § 16.2.2 – § 16.2.4 for more details.

GPLLNK (*addr* ---)

Allows you to link your program to Graphics Programming Language (GPL) routines. You must place on the stack the address of the GPL routine to which you wish to link as well as provide what additional information that routine may require.

base	set up FAC for call	*addr*	instr
HEX	**900 834A !**	**16**	**GPLLNK**

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

XMLLNK (*addr* ---)

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

base	*addr*	instr
HEX	**800**	**XMLLNK**

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

Note: The above FP multiplication routine requires the FP multiplier in FAC and the FP multiplicand in ARG. The product is returned in FAC. Th FP library (Chapter 7) uses the code in the above example for FP multiplication.

DSRLNK (---)

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

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

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

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

5.1.3 VDP Write-Only Registers

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

VWTR (*b n* ---)

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

base	*b*	*n*	instr
HEX	**F5**	**7**	**VWTR**

The above instruction writes **F5h** into VDP write only register number 7. This particular register controls the foreground and background colors in text and text80 modes. The foreground color is ignored in other modes. Executing the above instruction will change the foreground color to white and the background color to light blue.

5.1.4 VDP RAM Single-Byte Logical Operations

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

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

base	*b*	*vaddr*	instr
HEX	**F0**	**804**	**VAND**
HEX	**F0**	**804**	**VOR**
HEX	**F0**	**804**	**VXOR**

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

5.2 Disk Utilities

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

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

5.3 Listing Utilities

There are three words defined in **fbForth** starting in block 51 of FBLOCKS, which make listing information from a Forth blocks file very simple. The following descriptions refer to FBLOCKS dated 12DEC2013 or later to insure that you can print the first 3 blocks. If the file contains a number of blocks not evenly divisible by 3, printing the last 1 or 2 blocks will cause a file error message to be printed when **TRIAD** tries to read past the end of the blocks file.

TRIAD (*blk* ---)

The first, called **TRIAD**, requires a block number on the stack. When executed, it will end with a block number evenly divisible by three. Blocks that contain non-printable information will be skipped. If your RS232 printer is not on Port 1 and set at 9600 Baud, you must modify the word **SWCH** on your System disk.

TRIADS (blk_1 blk_2 ---)

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

INDEX (blk_1 blk_2 ---)

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

5.4 Debugging

5.4.1 Dump Information to Terminal

Loading block 21 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 Editors” for some discussion of 80-column text mode).

VLIST (---)

The **fbForth** word **VLIST** lists to your terminal the names of all words currently defined in the **CONTEXT** vocabulary. This instruction requires no parameters and may be halted and started again by pressing any key as with **INDEX** in the previous section. When finished or aborted with ***<BREAK>***, **VLIST** displays the number of words listed.

DUMP (*addr count* ---)

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

base	*addr*	*count*	instr
HEX	**2010**	**20**	**DUMP**

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

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

Press any key to temporarily stop execution in order to read the information before it scrolls off the screen. Press any key to continue. To exit this routine permanently, press ***<BREAK>***.

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

.S (---)

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

```
.S
| 1 8 189  ok:3
```

5.4.2 Tracing Word Execution

This section is based on the following article available at *www.forth.org* :

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

A special set of instructions in block 23 of FBLOCKS allows you to trace the execution of any colon definition. Executing the **TRACE** instruction will cause all following colon definitions to be compiled in such a way that they can be traced. In other words, the Forth word **:** takes on a new meaning. To stop compiling under the **TRACE** option, type **UNTRACE**. When you have finished debugging, recompile the routine under the **UNTRACE** option.

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

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

```
DECIMAL  ok:0
TRACE  ok:0                  (compile next definition with TRACE option)
: CUBE DUP DUP * * ;  ok:0   (routine to be traced)
UNTRACE  ok:0                (don’t compile next definition with TRACE option)
: TEST CUBE ROT CUBE ROT CUBE ;  ok:0
TRON  ok:0                   (want to execute with a TRACE)
5 6 7 TEST                   (put parameters on stack and execute TEST)
CUBE                         (TRACE begins)
| 5 6 7                      (stack contents upon entering CUBE)
CUBE
| 6 343 5                    (stack contents upon entering CUBE)
CUBE
| 343 125 6  ok:3
.S                           (check final stack contents)
| 343 125 216  ok:3          (stack contents after final CUBE )
```

5.4.3 Recursion

Normally, a Forth word cannot call itself before the definition has been compiled through to a **;** because the smudge bit is set, which prevents the word from being found during compilation. To allow recursion, **fbForth** includes the special word **MYSELF** .

MYSELF (---)

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

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

```
DECIMAL  ok:0
TRACE  ok:0              (compile following definition under TRACE option)
: FACT DUP 1 > IF DUP 1 - MYSELF * ENDIF ;  ok:0
UNTRACE  ok:0
TRON  ok:0
5 FACT                   (put parameter on stack and execute FACT)
FACT                     (TRACE begins)
| 5
FACT
| 5 4
FACT
| 5 4 3
FACT
| 5 4 3 2
FACT
| 5 4 3 2 1  ok:1
.S                       (check final stack contents)
| 120  ok:1
```

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

5.5 Random Numbers

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

RNDW (--- *n*)

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

RNDW

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

RND (n_1 --- n_2)

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

base	n_1	instr
DECIMAL	**13**	**RND**

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

RANDOMIZE (---)

To guarantee a different sequence of random numbers each time a program is run, the **RANDOMIZE** instruction must be used. **RANDOMIZE** places an unknown seed into the random number generator. The seed is calculated by clearing the VDP status register by reading it at **8802h** and entering a counter loop that increments the counter and checks the VDP status register for the next VDP interrupt, at which point it exits the loop and stores the counter in the seed location **83C0h**.

SEED (*n* ---)

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

```
4 SEED
```

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

5.6 Miscellaneous Instructions

' (--- *pfa*)

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

, (*n* ---)

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

C, (*b* ---)

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

CLS (---)

CLS is part of **fbForth**'s resident dictionary. Use this word to clear the display screen. **CLS** clears the display screen by filling the screen image table with blanks. The screen image table runs from **SCRN_START** to **SCRN_END** . **CLS** may be used inside a colon definition or directly from the keyboard. **CLS** will not clear bitmap displays or sprites.

6 An Introduction to Graphics

Words introduced in this chapter:

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

6.1 Graphics Modes

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

0) **Text80 Mode—**This is the same as text mode described below except that, in text80 mode, the screen is 80 columns by 24 lines. The user should insure that the system in use is capable of displaying 80-columns before invoking it, *i.e.*, it should be equipped with an F18A VDP (available at *http://codehackcreate.com/*) or similar device.

1) **Text Mode—**Standard ASCII characters are available, and new characters may be defined. All characters have the same foreground and background color. The screen is 40 columns by 24 lines. Text mode is used by the Forth 40/80-column screen editor.

2) **Graphics Mode—**Standard ASCII characters are available, and new characters may be defined. Each character set may have its own foreground and background color.

3) **Multicolor Mode—**The screen is 64 columns by 48 rows. Each standard character position is now 4 smaller boxes which can each have a different color. ASCII characters are not available and new characters cannot be defined.

4) **Bitmap Mode (Graphics2)—**This mode is available only on the TI-99/4A. Bitmap mode allows you to set any pixel on the screen and to change its color within the limits permitted by the TMS9918a. The screen is 256 columns by 192 rows.

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

6) **Split2 Mode—**This mode is the other of the two unique graphics modes created by using graphics2 mode in a non-standard way [see (5)]. Split2 sets the top one sixth of the screen as a text window and the rest in graphics2 mode.

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

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

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

TEXT80 (---)

TEXT (---)

GRAPHICS (---)

MULTI (---)

GRAPHICS2 (---)

SPLIT (---)

SPLIT2 (---)

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

VDPMDE (--- *addr*)

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

```
GRAPHICS VDPMDE @ .
2  ok:0
```

6.2 fbForth *Graphics Words*

Many **fbForth** words have been defined to make graphics handling much easier for the user. As many words are mentioned, an annotation will appear underneath them denoting which of the modes they may be used in (T G M B). These denote text, graphics, multicolor and bitmapped (graphics2, split, split2) modes, respectively—'T' includes text80.

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

6.3 Color Changes

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

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

`SCREEN` (*color* ---)

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

base	*color*	instr	
`HEX`	**`B`**	**`SCREEN`**	or
`DECIMAL`	**`11`**	**`SCREEN`**	
		(T G M B)	

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

```
HEX 1B SCREEN
```

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

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

base	*fg*	*bg*	*charset*	instr	
`HEX`	**`4`**	**`D`**	**`1A`**	**`COLOR`**	or
`DECIMAL`	**`4`**	**`13`**	**`26`**	**`COLOR`**	
				(G)	

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

6.4 Placing Characters on the Screen

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

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

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

For example,

base	*col*	*row*	*count*	*char*	instr	
HEX	**A**	**11**	**5B**	**2A**	**HCHAR**	or
DECIMAL	**10**	**17**	**91**	**42**	**HCHAR**	
					(T G)	

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

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

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

GCHAR (*col row --- char*)

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

base	*col*	*row*	instr	
HEX	**F**	**11**	**GCHAR**	or
DECIMAL	**15**	**17**	**GCHAR**	
			(T G)	

2Ah or 42 would be left on the stack.

6.5 Defining New Characters

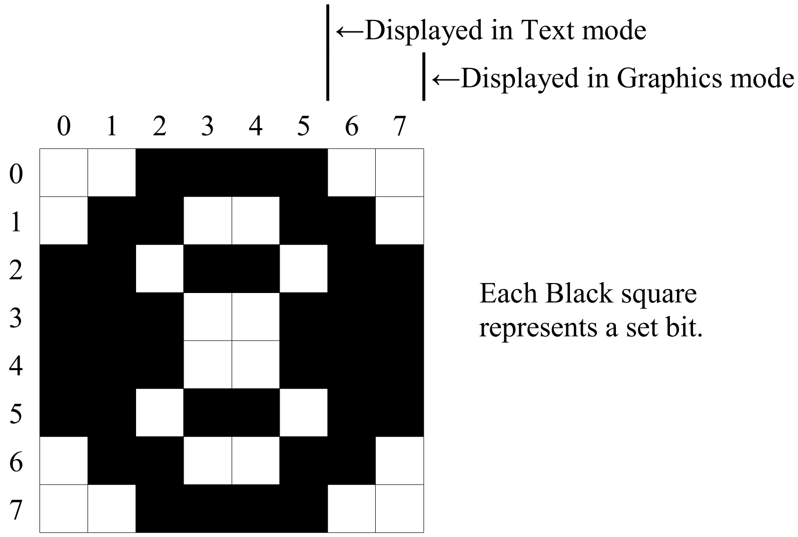
Each character in graphics mode is 8 x 8 pixels in size. Each row makes up one byte of the 8-byte character definition. Each set bit (1) takes on the foreground color while the others remain the background color.

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

For example, these 8 bytes:

	3C66h	**DBE7h**	**E7DBh**	**663Ch**
Rows	0 – 1	2 – 3	4 – 5	6 – 7

define this character:



Each Black square represents a set bit.

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

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

base	n_1	n_2	n_3	n_4	*char*	instr	
HEX	**3C66**	**DBE7**	**E7DB**	**663C**	**7B**	**CHAR**	or
DECIMAL	**15426**	**56295**	**59355**	**26172**	**123**	**CHAR**	
						(T G)	

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

CHARPAT (*char* --- n_1 n_2 n_3 n_4)

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

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.

6.6 Sprites

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

6.6.1 Magnification

Sprites may be defined in 4 different sizes or magnifications:

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

The default magnification is 0.

MAGNIFY (*n* ---)

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

n	instr
2	**MAGNIFY**
	(G M B)

will change all sprites to double size and unmagnified.

6.6.2 Sprite Initialization

SSDT (*vaddr* ---)

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

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

base	*vaddr*	instr	
HEX	**2000**	**SSDT**	or
DECIMAL	**8192**	**SSDT**	
		(G M B)	

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

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

6.6.3 Using Sprites in Bitmap Mode

SATR (--- *vaddr*)

When using sprites in any of the bitmap modes (graphics2, split, split2), a little extra work is required. After entering the desired VDP mode, the location of the Sprite Attribute List must be changed to **1B00h** as follows:

```
HEX 1B00 ' SATR !
```

The base address of the Sprite Descriptor Table must also be changed using the **SSDT** instruction. It must be based at **3800h**:

```
HEX 3800 SSDT
```

Only 59 character numbers will be available for sprite patterns 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 line 6 of block 33 in FBLOCKS from **2 FILES** to **1 FILES** because **fbForth** 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.

3800h	Sprite Patterns 0-58
39DDh	**01DEh**
39DEh	Start of Disk Buffer Region for 2 files

6.6.4 Creating Sprites

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

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

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

base	n_1	n_2	n_3	n_4	*char*	instr	
HEX	**OFOF**	**2424**	**FOFO**	**4242**	**0**	**SPCHAR**	or
DECIMAL	**3855**	**9252**	**61680**	**8770**	**0**	**SPCHAR**	
						(G M B)	

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

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

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

base	*dotcol*	*dotrow*	*color*	*char*	*spr*	instr	
HEX	**6B**	**4C**	**5**	**10**	**1**	**SPRITE**	or
DECIMAL	**107**	**76**	**5**	**16**	**1**	**SPRITE**	
						(G M B)	

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

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

SPRPAT (*char spr* ---)

base	*char*	*spr*	instr	
HEX	**14**	**1**	**SPRPAT**	or
DECIMAL	**20**	**1**	**SPRPAT**	
			(G M B)	

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

SPRCOL (*color spr* ---)

base	*color*	*spr*	instr	
HEX	**C**	**2**	**SPRCOL**	or
DECIMAL	**12**	**2**	**SPRCOL**	
			(G M B)	

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

SPRPUT (*dotcol dotrow spr* ---)

base	*dotcol*	*dotrow*	*spr*	instr	
HEX	**28**	**4F**	**1**	**SPRPUT**	or
DECIMAL	**40**	**79**	**1**	**SPRPUT**	
				(G M B)	

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

6.6.5 Sprite Automotion

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

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

MOTION (*xvel yvel spr* ---)

base	*xvel*	*yvel*	*spr*	instr	
HEX	**FC**	**6**	**1**	**MOTION**	or
DECIMAL	**-4**	**6**	**1**	**MOTION**	
				(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.

n	instr
6	**#MOTION**
	(G M)

will set sprites #0 – #5 in motion.

n	instr
0	**#MOTION**

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

SPRGET (*spr --- dotcol dotrow*)

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

spr	instr
2	**SPRGET**
	(G M B)

will return on the stack the horizontal (*dotcol*) and vertical (*dotrow*) positions of sprite #2. The sprite does *not* have to be in automotion to use this instruction.

6.6.6 Distance and Coincidences between Sprites

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

SPRDIST (spr_1 spr_2 --- *n*)

spr_1	spr_2	instr
2	**4**	**SPRDIST**
		(G M B)

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

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

base	*dotcol*	*dotrow*	*spr*	instr
DECIMAL	**65**	**21**	**5**	**SPRDISTXY**
				(G M B)

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

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

COINC (*spr_1 spr_2 tol --- flag*)

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

spr_1	*spr_2*	*tol*	instr
7	**9**	**2**	**COINC**
			(G M B)

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

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

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

base	*dotcol*	*dotrow*	*spr*	*tol*	instr
DECIMAL	**63**	**29**	**8**	**3**	**COINCXY**
					(G M B)

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

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

COINCALL (*--- flag*)

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

6.6.7 Deleting Sprites

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

DELSPR (*spr* ---)

spr	instr
2	**DELSPR**
	(G M B)

will remove sprite #2 from the screen by altering its description in the Sprite Attribute List (see VDP Memory Map in Chapter 4). It sets sprite #2 to sprite pattern #0 and sets the sprite off screen at ***x*** = 1, ***y*** = 192. It zeroes the velocity of sprite #2 in the Sprite Motion Table, but does not alter the number of sprites the computer thinks are defined by virtue of not setting ***y*** = **D0h**, the ***y***-value that undefines all sprites with numbers greater than or equal to the lowest-numbered sprite with that value.

DELALL (---)

DELALL

(G M B)

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

6.7 Multicolor Graphics

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

MINIT (---)

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

When in multicolor mode, the columns are numbered 0 – 63 and rows are numbered 0 – 47. A multicolor character is ¼ the size of a standard character; therefore more of them fit across and down the screen.

MCHAR (*color col row* ---)

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

base	*color*	*col*	*row*	instr	
HEX	**B**	**1A**	**2C**	**MCHAR**	or
DECIMAL	**11**	**26**	**44**	**MCHAR**	

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

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

6.8 Using Joysticks

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

The **JOYST** instruction allows you to use joysticks in your **fbForth** program. **JOYST** accepts input from joystick #1 and the left side of the keyboard (n_1 = 1) or from joystick #2 and the right side of the keyboard (n_1 = 2). Return values depend on the value in

JMODE (see below). If **JMODE** = 0 (default), **JOYST** executes **JKBD** (see below for more detail), which returns the character code *char* of the key pressed, the ***x*** status n_2 and the ***y*** status n_3. If **JMODE** ≠ 0, **JOYST** executes **JCRU** , which checks only the joysticks and returns a single value with 0 or more of the 5 least significant bits set. See **JCRU** below for their meaning.

JMODE (--- *addr*)

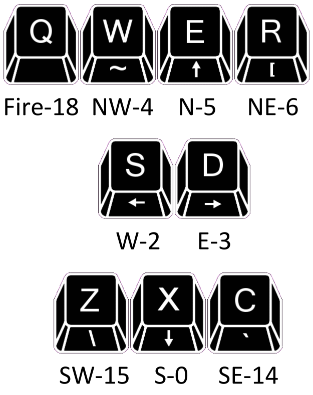
JMODE is a user variable that uses offset **26h** of the user variable table. It is used by **JOYST** to determine whether to execute **JKBD** (= 0) or **JCRU** (≠ 0). The default value is 0. See **JOYST** , **JKBD** and **JCRU** in this section.

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

Executed by **JOYST** when **JMODE** = 0, **JKBD** allows input from joystick #1 and the left side of the keyboard ($n_1 = 1$) or from joystick #2 and the right side of the keyboard ($n_1 = 2$). Values returned are the character code *char* of the key pressed, the ***x*** status n_2 and the ***y*** status n_3. A “Key Pad” exists on each side of the keyboard and may be used in place of joysticks. Map directions (N, S, E, W, NE, *etc*.) are used on the diagrams below to indicate the corresponding display-screen directions (up, down, right, left, diagonally-up-and-right, *etc.*) The following diagrams show which keys have which function.

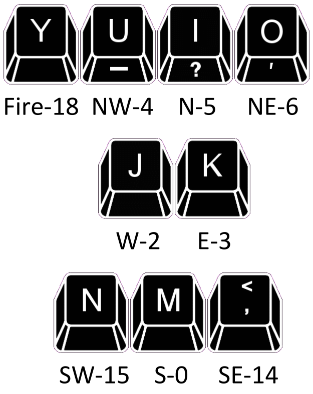
When Joystick #1 is specified, these keys on the left side of the keyboard are valid ▪▪━━►

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



When Joystick #2 is specified, these keys on the right side of the keyboard are valid ▪▪━━►

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



The **JKBD** instruction (or **JOYST** with **JMODE** = 0) returns 3 numbers on the stack: a character code *char* on the bottom of the stack, an ***x***-joystick status n_2 and a ***y***-joystick status n_3 on top of the stack. The joystick positions are illustrated in the diagram on page 53.

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

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

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

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

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

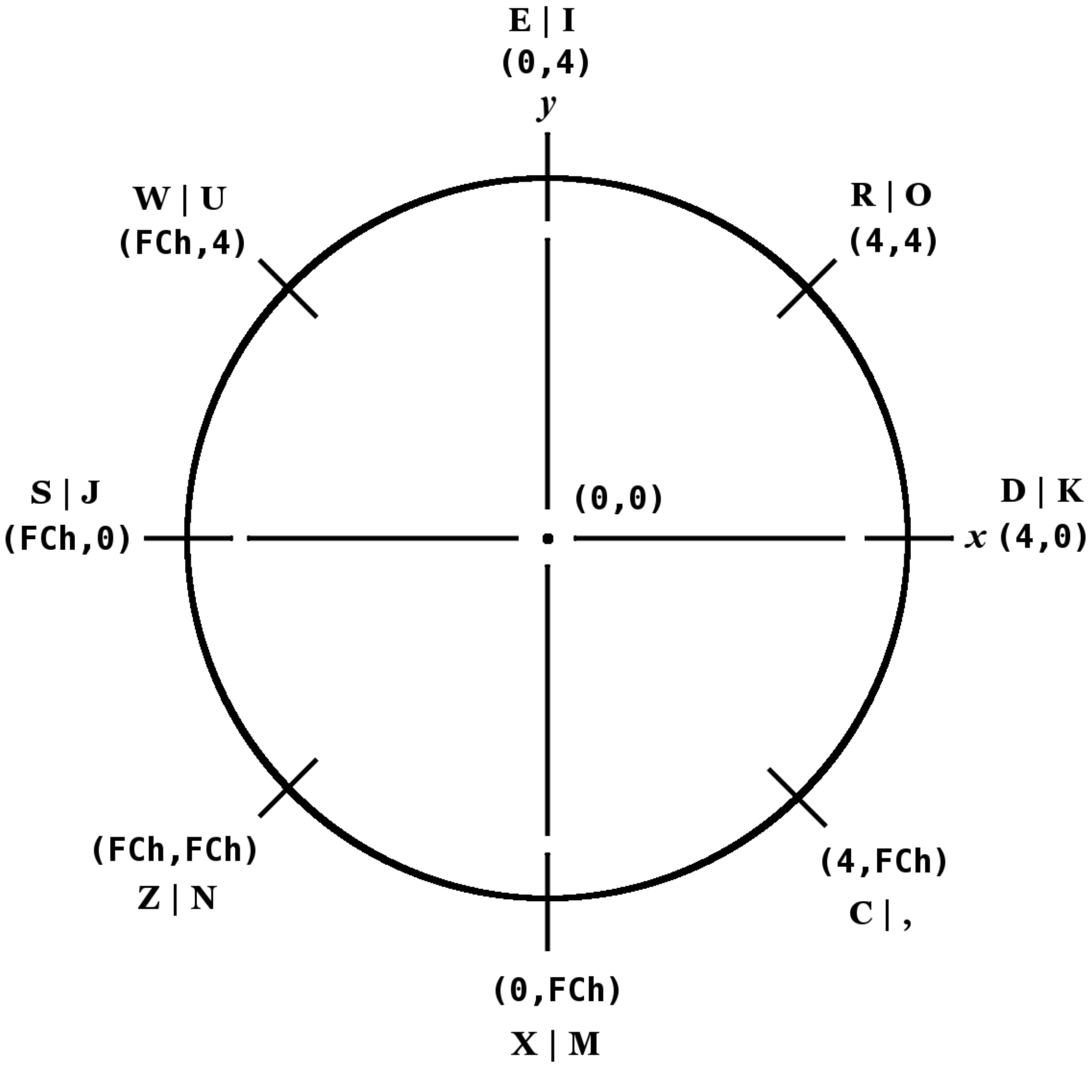
The reason, of course, that **FCh** is used in **fbForth** (and TI Forth before it) is that **FCh** is how -4 is represented in a single byte in the byte-oriented GROM joystick table where it is stored.

JCRU (n_1 --- n_2)

Executed by **JOYST** when **JMODE** ≠ 0, **JCRU** allows input from joystick #1 (n_1 = 1) or #2 (n_1 = 2). The value n_2 returned will have 0 or more of the 5 least significant bits set for direction and fire-button status. Bit values are 1 = Fire, 2 = W, 4 = E, 8 = S and 16 = N. Two-bit directional combinations are 18 = NW (N + W or 16 + 2), 20 = NE, 10 = SW and 12 = SE.

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

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



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

6.9 Dot Graphics

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

Three dot drawing modes are available:

DRAW (---)

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

UNDRAW (---)

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

DTOG (---)

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

DMODE (--- *addr*)

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

DOT (*dotcol dotrow*---)

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

base	*dotcol*	*dotrow*	instr
DECIMAL	**34**	**12**	**DOT**

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

DCOLOR (--- *addr*)

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

LINE ($dotcol_1$ $dotrow_1$ $dotcol_2$ $dotrow_2$ ---)

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

base	$dotcol_1$	$dotrow_1$	$dotcol_2$	$dotrow_2$	instr
DECIMAL	**23**	**12**	**56**	**78**	**LINE**

The above instruction will plot a line from left to right between (23,12) and (56,78). The line instruction calls **DOT** to plot each point; therefore, you must set **DMODE** and **DCOLOR** before using **LINE** if you do not want different plotting mode and FG-BG dot colors.

6.10 Special Sounds

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

BEEP (---)

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

HONK (---)

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

6.11 Constants and Variables Used in Graphics Programming

The following constants and variables are defined in the graphics routines. The values of **COLTAB** , **PDT** , **SATR** and **SPDTAB** must be changed if you are operating in graphics2, split or split2 mode. See the VDP Memory Map in Chapter 4. Even though the VRAM tables these constants represent are changed when executing **GRAPHICS2** , **SPLIT** and **SPLIT2** , these constants are not updated by those words and are, therefore, the user's responsibility to insure they have the proper values for the graphics primitives loaded from block 36*ff*.

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

7 The Floating Point Support Package

Words introduced in this chapter:

`>ARG`
`>F`
`>FAC`
`>ROA`
`?FLERR`
`ATN`
`COS`
`EXP`
`F!`
`F*`
`F+`
`F-`
`F->S`
`F.`
`F.R`
`F/`
`FO<`
`FO=`
`F<`
`F=`
`F>`
`F@`
`FAC->S`
`FAC>`
`FAC>ARG`
`FADD`
`FDIV`
`FDUP`
`FF.`
`FF.R`
`FLERR`
`FMUL`
`FOVER`
`FSUB`
`FSWAP`
`INT`
`LOG`
`PI`
`ROA`
`ROA>`
`S->F`
`S->FAC`
`SETFL`
`SIN`
`SQR`
`TAN`
`VAL`

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

7.1 Floating Point Stack Manipulation

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

FDUP	$(f \text{ --- } f\ f)$
FDROP	$(f \text{ --- })$
FOVER	$(f_1\ f_2 \text{ --- } f_1\ f_2\ f_1)$
FSWAP	$(f_1\ f_2 \text{ --- } f_2\ f_1)$

7.2 Floating Point Fetch and Store

Floating point numbers can be stored and fetched by using

F! (*f addr* ---)

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

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

7.3 Floating Point Conversion Words

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

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

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

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

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

7.4 Floating Point Number Entry

In addition, the word

>F (--- *f*)

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

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

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

7.5 Floating Point Arithmetic

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

F+ (f_1 f_2 --- f_3)

`F-`	(f_1 f_2 --- f_3)	Puts on the stack the result (f_3) of $f_1 - f_2$.
`F*`	(f_1 f_2 --- f_3)	Puts on the stack the result (f_3) of f_1 x f_2.
`F/`	(f_1 f_2 --- f_3)	Puts on the stack the result (f_3) of f_1 / f_2.

`PI` (--- f)

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

7.6 Floating Point Comparison Words

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

`F0<`	(f --- *flag*)	*flag* is true if f on stack is negative
`F0=`	(f --- *flag*)	*flag* is true if f on stack is zero
`F>`	(f_1 f_2 --- *flag*)	*flag* is true if $f_1 > f_2$
`F=`	(f_1 f_2 --- *flag*)	*flag* is true if $f_1 = f_2$
`F<`	(f_1 f_2 --- *flag*)	*flag* is true if $f_1 < f_2$

7.7 Formatting and Printing Floating Point Numbers

`F.` (f ---)

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

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

`F.R` (f n ---)

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

Two additional words are used for more specific formatting:

`FF.` (f n_1 n_2 ---)

`FF.` requires two integers on the stack above the floating point number f. They control the maximum number of digits (n_1) to convert and the number of digits (n_2) following the decimal point.

`FF.R` (f n_1 n_2 n_3 ---)

`FF.R` adds the printing field width (n_3), in which the output is right justified. As for **`FF.`** , n_1 is the maximum number of digits to convert and n_2 is the number of digits following the decimal point.

It should be noted that the exponential format of the output string 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!

7.8 Transcendental Functions

The following transcendental functions are also available:

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

Caution! A conflict exists when using transcendentals while in bitmap and text80 modes. The contents of the VDP Rollout Area (**3C0h** – **3DFh**) must be saved before transcendentals or floating point prints are executed and restored upon completion. **>ROA** and **ROA>** have been provided for your use to save and restore this area. Floating point prints already use those words. You will notice the screen flickering in the Rollout Area any time these functions are used. Of course, if you don't save/restore the Rollout Area, the screen will be garbled in that area.

Note: The transcendentals also use the area known as the value stack for calculations. This area is pointed to by **836Eh** (VSPTR); however, the graphics mode-changing words move it out of the way. See Memory Maps in Chapter 4 for locations.

7.9 Interface to the Floating Point Routines

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

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

FAC	(--- *addr*)	constant that puts the address of FAC on the stack.
ARG	(--- *addr*)	constant that puts the address of ARG on the stack.

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

>FAC	(*f* ---)	moves *f* to FAC.

>ARG (*f* ---) moves *f* to ARG.

FAC> (--- *f*) is used to move data from FAC to the stack.

SETFL (f_1 f_2 ---)

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

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

FADD (---)

FSUB (---)

FMUL (---)

FDIV (---)

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

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

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

Several miscellaneous words include:

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

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

VAL (---)

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

FLERR (--- *n*)

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

?FLERR (---)

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

```
?FLERR ? floating point error
```

Note: A few floating point operations, unfortunately, do not reset the floating point error location, **8354h**, before they run. If you are testing for the error, you should probably reset it yourself after you've dealt with the error, which you can do with

```
HEX 0 8354 C!
```

7.10 Floating Point Error Codes

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

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

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

Words introduced in this chapter:

- `APPND`
- `CHAR-CNT!`
- `CHAR-CNT@`
- `CHK-STAT`
- `CLR-STAT`
- `CLSE`
- `DLT`
- `DOI/O`
- `DSPLY`
- `F-D"`
- `FILE`
- `FXD`
- `GET-FLAG`
- `I/OMD`
- `INPT`
- `INTRNL`
- `LD`
- `N-LEN!`
- `OPN`
- `OUTPT`
- `PAB-ADDR`
- `PAB-BUF`
- `PAB-VBUF`
- `PABS`
- `PUT-FLAG`
- `RD`
- `REC-LEN`
- `REC-NO`
- `RLTV`
- `RSTR`
- `SCRTCH`[15]
- `SET-PAB`
- `SQNTL`
- `STAT`
- `SV`
- `SWCH`
- `UNSWCH`
- `UPDT`
- `VRBL`
- `WRT`

This chapter will explain the means by which different types of data files native to the TI-99/4A are accessed with **fbForth**. 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 this version of **fbForth** 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** system (as opposed to TI Forth) uses only normally formatted disks for the **fbForth** program (FBFORTH) and 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 may destroy them.

8.1 Switching VDP Modes After File Setup

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

[15] **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.

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.

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 293*ff*. The following diagram illustrates the structure of a PAB:

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

8.3 File Setup and I/O Variables

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

PAB-ADDR (--- *addr*)

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

PAB-BUF (--- *addr*)

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

PAB-VBUF (--- *addr*)

Holds address in VDP RAM of the first byte of a region of adequate length to store data 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 **8370h** and the new value put there after executing **1 FILES** . This should be safe as long as you do not read/write blocks because **fbForth** only opens a file to read/write one block. The blocks file is closed the rest of the time.

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

FILE (*vaddr*$_1$ *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:

Using The Defining Word, FILE

0 VARIABLE MY-BUF 78 ALLOT	(Create 80 character RAM buffer)
PABS @ 10 +	(PAB starts 10 bytes into VRAM region for PABS and this address will be stored in **PAB-ADDR**)
MY-BUF	(RAM address to be stored in **PAB-BUF**)
6000	(A free area at **1770h** in VRAM to be stored in **PAB-VBUF**)
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.)
JOE	(Use the file's identifying word (FID) before using any other file I/O words)

SET-PAB (---)

The word that creates the PAB skeleton is **SET-PAB** . It creates a PAB at the address shown in **PAB-ADDR** and zeroes the first ten bytes. It then places the contents of the variable **PAB-VBUF** into its PAB location at bytes 2 and 3. Obviously, **PAB-ADDR** and **PAB-VBUF** must be set up before **SET-PAB** is invoked, which is done by executing the file identifying word (**JOE** , in the above example) before **SET-PAB** . **SET-PAB** should be executed only once for each file and should immediately follow the first invocation of the file ID word.

8.4 File Attribute Words

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

	Options From		
Attribute Type	**TI Basic**	**fbForth**	**Description**
File Type	SEQUENTIAL	**SQNTL***	Records may only be accessed in sequential order
	RELATIVE	**RLTV**	Accessed in sequential or random order. Records must be of fixed length
Record Type	FIXED	**FXD***	All records in the file are the same length
	VARIABLE	**VRBL**	Records in the same file may have different lengths
Data Type	DISPLAY	**DSPLY***	File contains printable or displayable characters
	INTERNAL	**INTRNL**	File contains data in machine or binary format
Mode of Operation	INPUT	**INPT**	File contents can be read from, but not written to
	OUTPUT	**OUTPT**	File contents can be written to, but not read from
	UPDATE	**UPDT***	File contents can be written to *and* read from
	APPEND	**APPND**	Data may be added to the end of the file, but cannot be read

* Default if attribute is not specified

REC-LEN (*b* ---)

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

F-D" (---)

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

```
F-D" RS232.BA=9600"
F-D" DSK2.FILE-ABC"
```

8.5 Words that Perform File I/O

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

From TI Basic	From fbForth	DSR Opcode
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

OPN (---)

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

CLSE (---)

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

REC-NO (*n* ---)

Before using the **RD** and **WRT** instructions with a relative file, you must place the desired, zero-based record number *n* into the PAB. To do this, place the record number *n* on the stack and execute the word **REC-NO** . If your file is sequential, you need not do this.

RD (--- *n*)

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

WRT (*n* ---)

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

RSTR (*n* ---)

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

LD (*n* ---)

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

SV (*n* ---)

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

DLT (---)

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

STAT (--- *b*)

returns the status byte *b* (PAB+8, labeled "Screen Offset" in the PAB diagram above) of the current device/file from the PAB pointed to by **PAB-ADDR** after calling the DSR's STATUS opcode (9), which actually gets the status and writes it to PAB+8. Incidentally, the term "Screen Offset" for PAB+8 is from its use by the cassette interface, which must put prompts on the screen, to get the offset of screen characters with respect to their normal ASCII values. The table below, excerpted from the *Editor/Assembler Manual*, p. 298, shows the meaning of each bit of the status byte:

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

The words that follow are available for the advanced user and their utility can be worked out by examining their definitions in block 47*ff* in FBLOCKS. They are lower-level words that are used in the definitions of the above file I/O words.

GET-FLAG (--- *b*)

retrieves to the stack the flag/status byte *b* from byte 1 the current PAB. The high-order 3 bits are used for DSR error return, except for "bad device name". With the "bad device name" error, this error return will be 0; but, the GPL status byte (**837Ch**) will have the COND bit set (**20h**). The low-order 5 bits are set by routines that set the file type prior to calling **OPN** , which reads these bits. See table below for the meaning of each bit of the flag/status byte:

Flag/Status Byte of PAB (Byte 1)

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

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

Error Code	Meaning
0	No error unless bit 2 of status byte at address **`837Ch`** 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.*

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

writes the flag/status byte *b* on the stack to the current PAB to clear the error bits and set the file type prior to calling **`OPN`** . See table after **`GET-FLAG`** for the meaning of each bit.

`CLR-STAT` (---)

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

`CHK-STAT` (---)

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

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

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

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

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

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

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

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

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

DOI/O (*n* ---)

executes the **DSRLNK** word with the I/O opcode *n* on the stack. The current PAB must be updated with the information required by opcode *n* before executing **DOI/O** . See Section 18.2.1 of the *Editor/Assembler Manual* for details or consult the definitions in block 47*ff* in FBLOCKS of the I/O words, **OPN** , **CLSE** , **RD** , **WRT** , **RSTR** , **LD** , **SV** , **DLT** and **STAT** , all of which use this low-level word in their definitions.

Examples of file I/O in use are available in block 51*ff* in FBLOCKS, which defines the alternate I/O capabilities for printing to the RS232 interface.

8.6 Alternate Input and Output

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

The words

SWCH (---) and

UNSWCH (---)

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

```
SWCH 45 LIST UNSWCH
```

you can list block 45 of the current blocks file to the printer. If your RS232 printer is not on port 1 and set at 9600 baud or you would rather print via the parallel port, you must modify the word **SWCH** in block 51 of FBLOCKS.

The user variables

ALTIN (--- *vaddr*) and

ALTOUT (--- *vaddr*)

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

8.7 File I/O Example 1: Relative Disk File

Instruction	Comment
HEX	Change number base to hexadecimal
0 VARIABLE BUFR 3E ALLOT	Create space for a 64 byte buffer which will be the **PAB-BUF**
PABS @ A +	PAB starts 10 bytes into **PABS** . This will be the **PAB-ADDR**
BUFR 1700	Place the **PAB-BUF** and **PAB-VBUF** on stack in preparation for **FILE**
FILE TESTFIL	Associates the name **TESTFIL** with these three parameters
TESTFIL	File name must be executed before using any other File I/O words
SET-PAB	Create PAB skeleton
RLTV	Make **TESTFIL** a relative file
DSPLY	Records will contain printable information
40 REC-LEN	Record length is 64 (**40h**) bytes
F-D" DSK2.TEST"	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	**TESTDATA** is assumed to be the beginning memory address of the information to be written to the file
10 0 DO	Want to write 16 (**10h**) records
DUP	Duplicate address
BUFR 40 CMOVE	Move 64 bytes of information into the **PAB-BUF**
I REC-NO	Place record number into PAB
40 WRT	Write one 64-byte record to the disk
40 +	Increment address for next record
LOOP DROP	Clear stack
;	End definition

FIL-WRT	Execute writing procedure
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**
BUFR 40 DUMP	Print out the read information to the monitor. (**DUMP** routines must be loaded from block 21 of FBLOCKS)
CLSE	Close the file

8.8 File I/O Example 2: Sequential RS232 File

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

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

`: PRNT FILE-INFO`	`FILE-INFO` is assumed to be the beginning memory address of the information to be sent to the printer
`  20 0 DO`	Will write 32 records
`         DUP`	Duplicate address
`         MYBUF 50 CMOVE`	Move 80 characters from `FILE-INFO` to `MY-BUF`
`         50 WRT`	Write one record to printer
`         50 +`	Increment address on stack
`       LOOP DROP`	Clear stack
`;`	End definition

`PRNT`	Execute write program
`CLSE`	Close the file called `PRNTR`

9 The **fbForth** TMS9900 Assembler

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

```
R1 R2 A,
```

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

9.1 TMS9900 Assembly Mnemonics

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

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

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

9.2 **fbForth***'s Workspace Registers*

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

9.3 Loading and Using the Assembler

The **fbForth** TMS9900 Assembler is located in blocks 53 – 58 of FBLOCKS and is loaded by typing **53 LOAD** . The words **CODE** and **;CODE** and their synonyms, **ASM:** and **DOES>ASM:** are in the resident dictionary and part of the Forth vocabulary. When the assembler is loaded, it is loaded into the Assembler vocabulary. To use the assembler, it must be the context vocabulary, which may be effected by typing **ASSEMBLER** or by using the words **CODE** , **ASM:** , **;CODE** or **DOES>ASM:** , each of which makes Assembler the context vocabulary.

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

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

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

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

which is the same as

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

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

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

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

which is the same as

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

Just as with **ASM:** , assembly code following **DOES>ASM:** should end with **;ASM** . Later, when the newly created defining word **DEF-WRD** is executed in the following form, a new word is defined:

```
DEF-WRD TEST
```

This will create the word **TEST** which has as its execution procedure the code following **DOES>ASM:** . An example using **DOES>ASM:** is shown in § 9.9 .

9.4 fbForth *Assembler Addressing Modes*

We will now introduce those words that permit this assembler to perform the various addressing modes of which the TMS9900 is capable. Each of the remaining examples will show the **fbForth** 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** addressing mode words may also be used. The TI Forth code can be used in **fbForth** with no changes.

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

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

and is equivalent to the following assembly code:

```
        B      *R15
```

9.4.1 Workspace Register Addressing

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

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

9.4.2 Symbolic Memory Addressing

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

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

9.4.3 Workspace Register Indirect Addressing

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

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

9.4.4 Workspace Register Indirect Auto-increment Addressing

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

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

9.4.5 Indexed Memory Addressing

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

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

9.4.6 Addressing Mode Words for Special Registers

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

Register Address	Indirect	Indexed	Indirect Auto-increment
W	*W	@(W)	*W+
RP	*RP	@(RP)	*RP+
IP	*IP	@(IP)	*IP+
SP	*SP	@(SP)	*SP+
UP	*UP	@(UP)	*UP+
NEXT	*NEXT	@(NEXT)	*NEXT+
R*n*	*R*n*	@(R*n*)	*R*n*+

9.5 Handling the fbForth 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** Assembler word **`*SP+`** references the contents of the top cell of the parameter stack and then increments the stack pointer **`SP`** to reduce the size of the stack by one cell. The following code copies the contents of the stack's top cell to register 0 and reduces the stack by one cell:

```
*SP+ R0 MOV,
```

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

```
SP DECT,
R1 *SP MOV,
```

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

9.6 Structured Assembler Constructs

This assembler also permits the user to write structured code, *i.e.*, code that does not use labels. This is done in a manner very similar to the way that **fbForth** 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** Assembler example below; but, the **`ENDIF,`** of the TI Forth example works, as well. Be sure you have FBLOCKS dated 12DEC2013 or later before you attempt to use **`THEN,`** .

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

9.7 Assembler Jump Tokens

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

9.8 Assembly Example for Structured Constructs

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

```
fbForth                      TI Forth                     Conventional Assembler
( GENERALIZED SHIFTER )      ( GENERALIZED SHIFTER )      * GENERALIZED SHIFTER
ASM: SHIFT                   CODE SHIFT                           DEF   SHIFT
   *SP+ R0 MOV,                 *SP+ 0 MOV,               SHIFT   MOV   *SP+,R0
   NE IF,                       NE IF,                            JEQ   L3
     *SP R1 MOV,                  *SP 1 MOV,                      MOV   *SP,R1
     R0 ABS,                      0 ABS,                          ABS   R0
     GTE IF,                      GTE IF,                         JLT   L1
        R1 R0 SLA,                   1 0 SLA,                     SLA   R1,0
     ELSE,                        ELSE,                           JMP   L2
        R1 R0 SRL,                   1 0 SRL,             L1      SRL   R1,0
     THEN,                        ENDIF,
     R1 *SP MOV,                  1 *SP MOV,              L2      MOV   R1,*SP
   THEN,                        ENDIF,
;ASM                         NEXT,                        L3      B     *R15
```

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

9.9 Assembly Example with DOES>ASM:

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

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

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

```
    9 CONSTANT XXX
```

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

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

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

: CONSTANT Start colon definition of **CONSTANT** .

<BUILDS **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 CFA. This will be the new word's PFA, 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 CFA 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

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

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

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

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

9.10 ASM: and DOES>ASM: without the Assembler

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

1. Write, test and debug your Forth word using the **fbForth** 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** Assembler is loaded by executing **53 LOAD** .
3. Ensure that the dump routines are loaded by executing **21 LOAD** .
4. Load the screen that contains the definition of your Forth word and continue with (5) in the appropriate section below.

9.10.1 ASM: without the Assembler

Refer to the example in § 9.4.5 for the following:

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

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

   will dump this to the screen:

   ```
   E42C:  C0A1  2000 C8A2  2016  .. ... .
   E434:  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 **045Fh**, corresponding to the **;ASM** instruction.

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

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

   or

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

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. In the above example, the constant **XRAM** was used, so we need to replace the value **2000h** with the reference that put it there. In this case **XRAM** is used three times to get the cells with **2000h**, **2016h** and **201Ah**. We need to replace the **2000h** with **XRAM** , the **2016h** with **XRAM 16 +** and the **201Ah** with **XRAM 1A +** to get

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

   or

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

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

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

9.10.2 DOES>ASM: without the Assembler

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

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

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

   will display this on the screen:

```
        BA6A  ok:0
```

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

```
        HERE ' CONST2 SWAP OVER - DUMP
```

   will dump this to the screen:

```
        E424:  B998  A992  BA6A 0649  .....j.I
        E42C:  C65A  045F             .Z._
        ok:0
```

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

7. The last cell should be **045Fh**, corresponding to the **;ASM** instruction.

8. Write the high-level part of the word through **DOES>ASM:** followed by the machine code after **BA6Ah** [the CFA of **(;CODE)** we found above in (5)]. Use the dump above for guidance to compile with **,** the hexadecimal value for each cell as follows, replacing **045Fh** with **;ASM** for clarity:

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

   or

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

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

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

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

10 Interrupt Service Routines (ISRs)

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

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

10.1 Installing an fbForth *Interrupt Service Routine*

1) Create and test an **fbForth** routine to perform the function: **MYISR**
2) Determine the Code Field Address (CFA) of the routine in (1): **' MYISR CFA**
3) Write the CFA from (2) into user variable **ISR** .
4) Write the contents of **INTLNK** into 83C4h (33732).

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

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

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

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

10.2 Example of an Interrupt Service Routine

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

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

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

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

10.3 Installing the ISR

To install this ISR, the following code may be executed:

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

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

10.4 Some Additional Thoughts Concerning the Use of ISRs

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

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

11 Potpourri

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

11.1 BSAVE and BLOAD

BSAVE (*addr* blk_1 --- blk_2)

The word **BSAVE** is used to save binary images of the dictionary. It is not part of the resident dictionary; so, you will need to load it from block 59 of FBLOCKS (**59 LOAD**). **BSAVE** requires two entries on the stack:

1) The lowest memory address *addr* in the dictionary image to be saved to disk.
2) The Forth block number blk_1 to which the saved image will be written.

BSAVE will use as many **fbForth** 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** block until the entire image is saved. **BSAVE** returns on the stack the number blk_2 of the first available Forth block after the image.

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

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

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

BLOAD is part of your **fbForth** 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** block number *blk* on the stack. Before performing the **BLOAD** function the 14th and 15th bytes are checked to see that they contain the letters

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

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

11.1.1 Using BSAVE to Customize How fbForth 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**. 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** 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** 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 8 blocks starting with block 5 of FBLOCKS to save your system image:

```
' TASK 5 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** 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 5 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 have lines 11 – 15 to add your code as long as it eventually ends with **5 BLOAD** . This will load your **BSAVE**d system and it will happen a lot faster than loading the text blocks because they now don't need to be interpreted.

If you load the definition for **BSAVE** as the last thing you do before using **BSAVE** , you can save the 170 bytes it uses by **FORGET**ting it after **BLOAD**ing block 5:

```
5 BLOAD FORGET BSAVE
```

11.1.2 An Overlay System with BSAVE and BLOAD

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

Such an application might be set up as follows:

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

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

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

```
: OVERLAYS ;
' OVERLAYS NFA FENCE !
```

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

```
: OVLY_STRT ;
```

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

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

1. Remove all system components from the dictionary that are not required by your application and that are newer than **TASK** . To start with a dictionary with only resident words:
   a) Execute **-DUMP** to load the definition for **VLIST** .
   b) Execute **VLIST** to get the name of the word immediately following **TASK** . Remember that **VLIST** lists the dictionary from **HERE** back to older words.

   c) **FORGET** that word to leave only the resident dictionary. If the word following **TASK** , *i.e.*, listed just before **TASK** by **VLIST** , is **XXX** , then execute **FORGET XXX** .

2. Load all system components required to run your application.
3. Load block 59 to use **BSAVE** to save the binary images for your application and its overlays, even though your application will never need it.
4. Load application.
5. Load first overlay.
6. **BSAVE** application using the address of **TASK** to a free Forth block:

```
' TASK 110 BSAVE .
```

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

```
' OVERLAYS 120 BSAVE .
```

8. 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 130 BSAVE .** (Obviously, 130 should be a different block for each additional overlay.)

11.1.3 An Easier Overlay System in Source Code

The above **BSAVE** / **BLOAD** method for setting up an overlay system can be very difficult to maintain because of the unforgiving nature of **BLOAD** . Any changes in the application other than the overlay section will almost certainly necessitate re-saving *all* of the overlays. An easier method to maintain is one such as described in *Starting FORTH (1st Ed.)*, p. 80*ff.* 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** 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.

You can save the 170 bytes occupied by **BSAVE** if you load it after the main application. **BSAVE**ing the main application will still include **BSAVE** ; but, it will be forgotten when the word **OVERLAYS** is forgotten upon loading the first overlay.

11.2 Conditional Loads

CLOAD (*blk* ---)

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

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

<CLOAD> (---)

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

SLIT (--- *addr*)

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

WLITERAL (---)

is used to compile **SLIT** and the desired character string into the current dictionary definition. See the **fbForth** 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** 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** block. This is done by using the command:

```
0 CLOAD TESTWORD
```

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

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

11.3 CRU Words

The five words below have been included to assist in performing CRU (Communications Register Unit) related functions. They allow the **fbForth** programmer to perform the **LDCR**, **STCR**, **TB**, **SBO** and **SBZ** operations of the TMS9900 without using the Assembler. See CRU documentation in the *Editor/Assembler Manual* for more information.

LDCR (n_1 n_2 *addr* ---)

Performs a TMS9900 **LDCR** instruction. The CRU base address *addr* will be shifted left one bit and stored in workspace register R12 prior to executing the TMS9900 **LDCR** instruction. The 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.

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

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

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

SBO (*addr* ---)

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

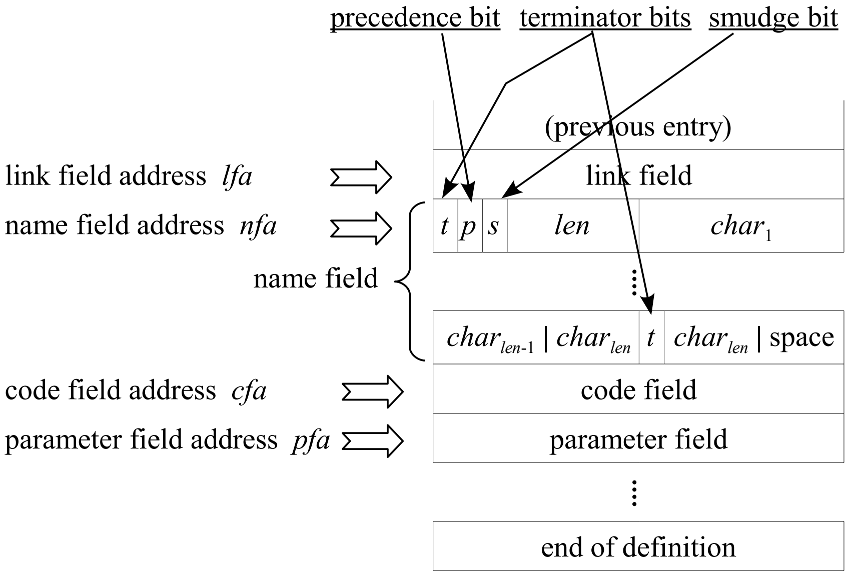
SBZ (*addr* ---)

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

12 fbForth Dictionary Entry Structure

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

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



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

12.1 Link Field

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

12.2 Name Field

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

The name field is a packed character string (see footnote 4 on page 17) in that the first byte is the length byte followed by the character string that represents the name. The three highest bits of the length byte are the beginning terminator bit (**80h**), the precedence bit (**40h**) and the smudge bit (**20h**). These are shown in the above figure as *t*, *p* and *s*, respectively. That leaves 5 bits for the character-length *len* of the name, which is the reason that **fbForth** words have a maximum length of 31 characters. The name field in **fbForth** always occupies an even number of bytes, *i.e.*, it begins and ends on a cell boundary. The last byte of the name field will be either the last character of the name or a space and will have the highest bit (**80h**) set as the ending terminator bit.

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

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

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

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

12.3 Code Field

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

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

12.4 Parameter Field

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

```
' cccc
```

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

```
-FIND cccc DROP DROP
```

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

The contents of the parameter field depend on the type of word defined. The following table shows common situations:

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

Appendix A ASCII Keycodes (Sequential Order)

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

…continued from previous page—

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

Appendix B ASCII Keycodes (Keyboard Order)

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

…continued from previous page—

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

Appendix C Differences between *Starting FORTH (1st Ed.) and* **fbForth**

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

Page	Word	Changes Required
63-82		The **fbForth** Editor is different (much better) than the editor described in this section. Read the section of this **fbForth** *Manual* describing the Editor.
83	**DEPTH**	**DEPTH** is defined in the resident **fbForth** dictionary.
84	**COPY**	**fbForth** has **CPYBLK** for this purpose, *q.v.*
84-5		Ignore Editor words.
89*ff*	**THEN**	**THEN** is in the **fbForth** vocabulary and is a synonym for the word **ENDIF** . Many people find **ENDIF** less confusing than **THEN** .
91	**0>**	This word is not in **fbForth** but can be created with the following definition: **: 0> 0 > ;**
91	**NOT**	This word is not in **fbForth**, but can be created with the following definition: **: NOT 0= ;**
101	**?DUP**	This word is identical to **-DUP** in **fbForth**. Use the following definition if necessary: **: ?DUP -DUP ;**
101*ff*	**ABORT"**	As with the Forth-79 Standard, **fbForth** provides **ABORT** instead of **ABORT"** .
102	**?STACK**	In **fbForth** this word automatically calls **ABORT** and prints the appropriate error message.
107	**2***	This word is not in **fbForth**, but can be created with the following definition: **: 2* DUP + ;**
107	**2/**	This word is not in **fbForth**, but can be created with the following definition: **: 2/ 1 SRA ;**
108	**NEGATE**	This word is not in **fbForth**, but can be created with the following definition: **: NEGATE MINUS ;**
110	**I**	This word exists in **fbForth** but also has a duplicate definition, **R** . **I** and **R** are identical in function. They both get a copy of the return stack top.
110	**I'**	This word is not in **fbForth**, but can be created with the following definition: (*Note*: **R** is a synonym for **I** .) **: I' R> R SWAP >R ;**

Page	Word	Changes Required
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.
112		Ignore the last two paragraphs. They do not apply.
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 11 1 DO`
		`    11 1 DO I J * 5 U.R LOOP CR LOOP ;`

134		When you execute the **DOUBLING** example, an extra number will be printed after 16384. This is because **+LOOP** behaves a little differently in **fbForth**.
136		In the definition of **COMPOUND** , the **CR** should precede **SWAP** instead of **LOOP** .
137	**XX**	When an error is detected in **fbForth**, the stack is cleared but then the contents of **BLK** and **IN** are saved on the stack to assist in locating the error. The stack may be completely cleared with the word **SP!** .
142	**PAGE**	This word is not in **fbForth**, but can be created with the following definition:
		`: PAGE CLS 0 0 GOTOXY ;`
161	**U/MOD**	This word is not in **fbForth**, but can be created with the following definition:
		`: U/MOD U/ ;`
161	**/LOOP**	This word is not in **fbForth**.
162	**OCTAL**	**OCTAL** does not exist in **fbForth**. See p. 163 for definition.
164-5		Numbers in **fbForth** may only be punctuated with periods. Commas, slashes and other marks are not permitted. Any number containing a period (**.**) is considered double-length. In later examples using **D.** and **UD.** , replace all punctuation in the inputs with decimal points. It is recommended that you not place more than one decimal place in each number if you want valid output.
166	**UD.**	This word is already defined in **fbForth**.
173	**D-**	This word is not in **fbForth**, but can be created with the following definition:
		`: D- DMINUS D+ ;`

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

Page	Word	Changes Required
174	`M+`	This word is not in **fbForth**, but can be created with the following definition: `: M+ 0 D+ ;`
174	`M/`	This word is different in **fbForth** and can be changed with the following definition: `: M/ M/ SWAP DROP ;`
174	`M*/`	Not available in **fbForth** 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**.
183*ff*		Variables in **fbForth** 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**, 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**, but can be created with the following definition: `: 2! >R R ! R> 2+ ! ;`
193	`2@`	This word is not in **fbForth**, but can be created with the following definition: `: 2@ >R R 2+ @ R> @ ;`
193	`2CONSTANT`	This word is not in **fbForth**, but can be created with the following definition: `: 2CONSTANT <BUILDS , , DOES> 2@ ;` This definition does *not* require a number on the stack.
199		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.

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

Page	Word	Changes Required
265	**RND**	**fbForth** has two random number generators: **RND** and **RNDW** . See Appendix D for descriptions. See also definitions for **SEED** and **RANDOMIZE** .
266	**MOVE**	In **fbForth**, **MOVE** moves *u* words in memory, not *u* bytes. **MOVE** can be redefined to conform to *Starting FORTH (1^{st} Ed.)*: **: MOVE 2/ MOVE ;**
266	**<CMOVE**	Not present in **fbForth**. Must be created with the Assembler if required. This word is used only when the source and destination regions of a move overlap and the destination is higher than the source.
270	**WORD**	In **fbForth**, the word **WORD** does not leave an address on the stack.
270	**TEXT**	This word is not available in **fbForth**, but can be defined as follows: **: TEXT 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.
277	**>BINARY**	This is named **(NUMBER)** in **fbForth**.
277		Because **WORD** does not leave an address on the stack, it is necessary to redefine **PLUS** as follows: **: PLUS 32 WORD DROP NUMBER + ." = " . ;**
279	**NUMBER**	This definition of **NUMBER** is not compatible with **fbForth**.
281	**-TEXT**	Not in **fbForth**. Use the definition on page 282.
292		**fbForth** uses the word pair **<BUILDS … DOES>** to define a new defining word. **<BUILDS** calls **CREATE** as part of its function.
297		To create a byte **ARRAY** in **fbForth**: **: ARRAY <BUILDS OVER , * ALLOT** **DOES> DUP @ ROT * + + 2+ ;**
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** .
301	**(DO)**	This is the runtime behavior of **DO** just as listed. **2>R** is not used, however.
301	**DO**	The given definition of **DO** is not compatible with **fbForth**. **fbForth**'s definition of **DO** is much more complex because of compile-time error checking.
303	**(LITERAL)**	The **fbForth** name for this word is **LIT** .
306		**fbForth** remains in compilation mode until a **;** is typed.

Appendix D The fbForth Glossary

fbForth 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".

The state of the top of the parameter stack (usually referred to simply as "the stack") before and after execution of an **fbForth** word is shown schematically as "(*before* --- *after*)", where "*before*" and "*after*" represent 0 or more cells relevant to the **fbForth** 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** 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** 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.

D.1 Explanation of Some Terms and Abbreviations

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

Term/Abbreviation	Meaning
addr, addr$_1$*, ...*	memory address
b	byte
col	column position
cccc , **nnnn** , **xxxx**	string representations
cfa	code field address
char	ASCII character code
count	count (length)
d, d$_1$*, d*$_2$*, ...*	signed double-precision numbers (2 cells each)
dotcol, dotcol$_1$*, dotcol*$_2$*, ...*	dot column position
dotrow, dotrow$_1$*, dotrow*$_2$*, ...*	dot row position
flag	Boolean flag
false	Boolean false flag (value = 0)
f, f$_1$*, f*$_2$*, ...*	floating point numbers (4 cells each)

Term/Abbreviation	Meaning
lfa	link field address
n, n_1, n_2, ...	signed single-precision numbers
nfa	name field address
pfa	parameter field address
row	row position
rem	remainder
blk	block number
spr	sprite number
str	string address
true	Boolean true flag (value ≠ 0)
tol	tolerance limit
u	unsigned single-precision number
ud	unsigned double-precision number (2 cells)
vaddr	VDP address

D.2 fbForth *Word Descriptions*

! Resident

(*n addr* ---)

Stores 16 bit-number *n* at address. Pronounced “store”.

!CSP Resident

(---)

Saves the stack position in user variable **CSP** . Used as part of compiler security.

Resident

(d_1 --- d_2)

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 **<#** .

#MOTION Graphics Primitives Library 36

(*n* ---)

Sets sprite numbers 0 to *n* – 1 in automotion.

#S Resident

(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 **<#** .

' *[immediate word]* Resident

(--- *pfa*)

Used in the form:

```
' nnnn
```

Searches the dictionary for **nnnn** and, if found, leaves the parameter field address *pfa* of the word. As a compiler directive, **'** , because it is an immediate word, executes in a colon definition to compile the address of a literal, *viz.*, the *pfa* of the found word. If the word is not found after a search of **CONTEXT** and **CURRENT** , the word is displayed followed by '?' to indicate the error. The stack is then cleared, the contents of **IN** and **BLK** are left on the stack and **QUIT** is called. Pronounced "tick".

(*[immediate word]* Resident

(---)

(is used in the form:

```
( cccc)
```

It starts a comment that will not be compiled if it occurs in a definition. It causes the interpreter to consume characters from the input stream until a ')' is found or the end of the input stream (block or TIB) is reached. May occur during execution or in a colon definition. A blank after the leading parenthesis is required. This is most useful for commenting Forth source code in blocks.

(+LOOP) Resident

(*n* ---)

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

(.") Resident

(---)

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

(;CODE) Resident

(---)

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

(ABORT) Resident

(---)

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

```
: (ABORT) ABORT ;
```

If you wished to redefine it to execute your error procedure, say **MY_ERROR_PROC** , you would replace **ABORT** with **MY_ERROR_PROC** as shown in the redefinition of **(ABORT)** below:

```
: (ABORT) MY_ERROR_PROC ;
```

(DO) Resident

(---)

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

(DOES>) Resident

(---)

The runtime procedure compiled by **DOES>** .

(FIND) Resident

(*addr nfa* --- *false* | *pfa b true*)

Searches the dictionary starting at the name field address *nfa*, looking for a match to the text at *addr*. The addresses, *addr* and *nfa*, both point to the length byte of packed character strings (see footnote 4 on page 17). Returns the parameter field address *pfa*, length byte *b* of name field, and *true* for a match. If no match is found, only *false* is left. [*Note:* See Chapter 12 about the length byte of a name field.]

(LINE) Resident

(*n blk* --- *addr count*)

Converts the line number *n* and the Forth block numbrer *blk* to the disk buffer address *addr* containing the data and the number *count* of characters. If the block is not in a block buffer, it is loaded from the current blocks file. If *count* is 64, the full-line text length of the block is indicated.

(LOOP) Resident

(---)

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

(NUMBER) Resident

(d_1 $addr_1$ --- d_2 $addr_2$)

The double number d_1 should be 0, *i.e.*, the stack should contain two 16-bit zeroes. The address $addr_1$ must point to the packed character string of the ASCII text to be converted to a double number, which will be left as d_2. The conversion begins at $addr_1 + 1$ 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 $addr_2$. **(NUMBER)** is used by **NUMBER** .

(OF) Resident

(---)

The run time procedure compiled by **OF** .

(UB) Resident

(*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 17) pointed to by *addr.*

***** Resident

(n_1 n_2 --- n_3)

Leaves the signed product of two signed numbers.

***/** Resident

(n_1 n_2 n_3 --- *quot*)

Leaves the quotient *quot* of $(n_1 * n_2) / n_3$, where all are signed numbers. Retention of an intermediate signed 32-bit product permits greater accuracy than would be available with the sequence :

$\boldsymbol{n_1\ n_2\ *\ n_3\ /}$

***/MOD** Resident

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

Leaves the quotient *quot* and remainder *rem* of the operation $(n_1 * n_2) / n_3$. An intermediate signed 32-bit product is used as for ***/** . In fact, ***/MOD** is used by ***/** .

+ Resident

(n_1 n_2 --- n_3)

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

+! Resident

(*n addr* ---)

Adds *n* to the value at the address. Pronounced "plus store".

+- Resident

(n_1 n_2 --- n_3)

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

+BUF Resident

($addr_1$ --- $addr_2$ *flag*)

Advance the disk buffer address $addr_1$ to the address of the next buffer $addr_2$. Boolean flag is false when $addr_2$ is the buffer presently pointed to by user variable **PREV** .

+LOOP *[immediate word]* Resident

Used in a colon definition in the form:

DO … *n* +LOOP

COMPILE TIME: (*addr* 3 ---)

+LOOP compiles the runtime word **(+LOOP)** and the branch offset computed from **HERE** to the address *addr* left on the stack by **DO** . The value 3 is used for compile-time error checking.

RUNTIME: (*n* ---)

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

, Resident

(*n* ---)

Store *n* into the next available dictionary memory cell, advancing the dictionary pointer. Pronounced "comma".

- Resident

(n_1 n_2 --- n_3)

Leave the difference n_3 of $n_1 – n_2$.

--> *[immediate word]* Resident

(---)

Continues interpretation with the next Forth block in the current blocks file. **-->** can only be used while loading blocks. Pronounced “next block”.

-DUP Resident

(n_1 --- n_1 | n_1 n_1)

Duplicate n_1 only if it is non-zero. This is usually used to copy a value just before **IF** , to eliminate the need for an **ELSE** clause to drop a **DUP**ed 0.

-FIND Resident

(--- *false* | *pfa len true*)

Accepts the next text word (delimited by blanks) in the input stream to **HERE** as a packed character string (see footnote 4 on page 17), searches the **CONTEXT** and then **CURRENT** vocabularies for a matching entry. If found, the dictionary entry’s parameter field address *pfa*, its length byte *len* and *true* are left. Otherwise, only *false* is left. [*Note:* See Chapter 12 about the length byte.]

-TRAILING Resident

(*addr* n_1 --- *addr* n_2)

Adjusts the character count n_1 of a character string at *addr* to suppress the output of trailing blanks by **TYPE** , *i.e.*, the characters at *addr* + n_2 to *addr* + n_1 are blanks. If the character string is a packed character string (see footnote on page), *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

(---)

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 (**."**) .

.LINE Resident

(*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.

.R 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 Resident

(---)

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 '|'.

/ Resident

(n_1 n_2 --- n_3)

Leaves the quotient n_3 of n_1 / n_2.

/MOD Resident

(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.

0 1 2 3 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.

0= Resident

(*n* --- *flag*)

Leaves a true flag if the number is equal to zero. Otherwise, **0=** leaves a false flag.

0BRANCH Resident

(*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– Resident

(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.

: *[immediate word]* Resident

(---)

Used in the form, called a colon definition:

```
: cccc … ;
```

Creates a dictionary entry defining **cccc** as equivalent to the sequence of Forth word definitions in '...' until the next **;** , **DOES>ASM:** or **;CODE** . The compiling process is done by the text interpreter as long as **STATE** is non-zero. Other details are that the **CONTEXT** vocabulary is set to the **CURRENT** vocabulary and that words with the precedence bit (see § 12.2 "Name Field") set are executed rather than being compiled.

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

: (*traceable*) *[immediate word]* TRACE — Colon Definition Tracing [23]

(---)

This is an alternate definition of **:** that adds the capability to colon definitions of being traced when they are executed. When a colon definition is compiled under the **TRACE** option, tracing output may be turned on with **TRON** and off with **TROFF** prior to executing the word so defined. After **TRON** is executed, each time the word is executed its name will be output along with the contents of the stack. See **TRACE** , **UNTRACE** , **TRON** and **TROFF** .

; *[immediate word]* Resident

(---)

Terminates a colon definition and stops further compilation. Compiles the runtime **;S** .

;ASM Resident

(---)

Synonym for Assembler word **NEXT,** . **;ASM** should be paired with **ASM:** to clearly surround assembly code or machine code:

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

;ASM puts **045Fh** (machine code for ALC: **B *R15**) at **HERE** and advances **HERE** . See Chapter 9 "The fbForth TMS9900 Assembler" for details. See also **ASM:** , **NEXT,** and **CODE** for more information.

;CODE *[immediate word]* Resident

(---)

Used with **<BUILDS** in the form:

```
: cccc <BUILDS … ;CODE <assembly mnemonics> NEXT,
```

Stops compilation and terminates a new defining word, **cccc** , by compiling **(;CODE)** . Sets the **CONTEXT** vocabulary to **ASSEMBLER** , assembling to machine code the assembly mnemonics following **;CODE** .

When **cccc** later executes in the form:

```
cccc  nnnn
```

the word **nnnn** will be created with its execution procedure given by the machine code following **(;CODE)** in the definition of **cccc** , *i.e.*, when **nnnn** is executed, it does so by jumping to that code in **cccc** . An existing defining word (**<BUILDS** in this case) must exist in **cccc** prior to **;CODE** . See Chapter 9 "The fbForth TMS9900 Assembler" for more details.

;S Resident

(---)

Stops interpretation of a Forth block. **;S** is also the runtime word compiled at the end of a colon definition, which returns execution to the calling procedure.

< Resident

(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.

The conversion is done on a 31-bit (positive) double number producing text at **PAD** (working downward toward **HERE**), eventually suitable for output by **TYPE** . The picture template between **<#** and **#>** represents the output picture from right to left, *i.e.*, the rightmost digit is processed first. The following is an example of generalized output from a double number on the stack that may be positive or negative:

```
SWAP OVER DABS <# #S SIGN #> TYPE
```

In the example above, **SWAP** puts the high-order cell, which contains the sign bit, on the bottom; **OVER** copies it back to its proper place on top, leaving 3 cells (*n d*) on the stack; and **DABS** forces *d* positive. This arrangement is what is expected by **SIGN** .

Important note: You should not execute words that change **HERE** or **PAD** until after you have finished formatting the number and retrieving the converted output.

See **#** , **#S** , **SIGN** , **#>** , **HLD** and **HOLD** for more information.

<BUILDS Resident

(---)

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

```
: cccc <BUILDS … DOES> … ;            or
: cccc <BUILDS … ;CODE … NEXT,        or clearer equivalent
: cccc <BUILDS … DOES>ASM: … ;ASM
```

Each time **cccc** is executed, **<BUILDS** defines a new word with a high-level (**DOES>**) or machine-code (**;CODE** or **DOES>ASM:**) execution procedure. Executing **cccc** in the form:

```
cccc  nnnn
```

uses **<BUILDS** to create a dictionary entry for **nnnn** . For the definition with **DOES>** , when **nnnn** is later executed, it has the address of its parameter area on the stack and executes the words after **DOES>** in **cccc** . For the definition with **DOES>ASM:** , when **nnnn** is later executed, it executes the words after **DOES>ASM:** in **cccc** . **<BUILDS** allows runtime procedures to be written in high-level code with **DOES>** or in assembler code with **;CODE** or **DOES>ASM:** . See **DOES>ASM:** for equivalence with **;CODE** .

<BUILDS is simply defined as

```
: <BUILDS CREATE SMUDGE ;
```

<CLOAD> Resident

(---)

The runtime procedure compiled by **CLOAD** .

= Resident

(n_1 n_2 --- *flag*)

Leaves a true flag if $n_1 = n_2$. Otherwise, it leaves a false flag.

=CELLS Resident

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

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

> Resident

(n_1 n_2 --- *flag*)

Leaves a true flag if $n_1 > n_2$. Otherwise, it leaves a false flag.

>ARG Floating Point Math Library [24]

(*f* ---)

Moves a floating point number *f* from the stack into the **ARG** register.

>F *[immediate word]* Floating Point Math Library [24]

(--- *f*)

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

>FAC Floating Point Math Library [24]

(*f* ---)

Moves a floating point number from the stack into the **FAC** register.

>R Resident

(*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 **R>** in the same definition.

>ROA Floating Point Math Library [24]

(---)

Saves VDP Rollout Area to **ROA** , *q.v.*.

? Resident

(*addr* ---)

Prints the value contained at address *addr* in free format according to the current radix stored in **BASE** . This word is short for the two words, **@ .** .

?COMP Resident

(---)

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.*,

```
9 0 DO I . LOOP ENTER 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.

?CSP Resident

(---)

This word is used in the definitions of **;** , **;CODE** and **DOES>ASM:** 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.*,

```
: XXXX IF ; ENTER ; ?  definition not finished
```

?ERROR Resident

(*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** 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 definition. **?EXEC** issues the error message, “execution only”, as in

```
: XXXX : … ; ENTER : ?  execution only
```

?FLERR Floating Point Math Library [24]

(---)

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 cross-referencing the code in the error table in § 7.10 “Floating Point Error Codes”.

?KEY Resident

(--- *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 Resident

(--- *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** 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

```
--> ENTER --> ?   use only when loading
```

?PAIRS Resident

(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 ;
```

?TERMINAL Resident

(--- *flag*)

Scans the terminal keyboard for actuation of the break key (***<BREAK>***). A true flag indicates actuation. On the TI-99/4A, ***<FCTN+4>***, ***<BREAK>*** and ***<CLEAR>*** are all the same key.

@ Resident

(*addr* --- *n*)

Leave the 16-bit contents *n* of *addr*.

A$$M TMS9900 Assembler [53]

(---)

This word is compiled into the **FORTH** vocabulary and marks the end of the **ASSEMBLER** vocabulary. It is used by **CLOAD** to determine whether the TMS9900 Assembler has been loaded.

ABORT Resident

(---)

ABORT is **fbForth**'s warm start. It clears the stacks, sets both **CONTEXT** and **CURRENT** to the **FORTH** vocabulary, enters the execution state and, after printing "**fbForth** 1.0", executes **INTERPRET** to get user input from the terminal.

ABS Resident

(n_1 --- n_2)

Leaves the absolute value of n_1 as n_2.

AGAIN *[immediate word]* Resident

Used in a colon definition in the form:

BEGIN … AGAIN

Compile time: (*addr* 1 ---)

AGAIN compiles **BRANCH** with an offset from **HERE** to *addr*, which it copies to the space reserved for it at *addr*. The value 1 is used for compile-time error checking.

Runtime: (---)

AGAIN forces execution to return to the corresponding **BEGIN** . There is no effect on the stack. Execution cannot leave the loop unless **R> DROP** is executed one level below by some word in the loop.

ALLOT Resident

(*n* ---)

Adds the signed number *n* to the dictionary pointer **DP** , which moves **HERE** by *n* bytes. It has the effect of reserving *n* bytes of dictionary space if it is positive and moving **HERE** backwards to reclaim memory if it is negative (*be careful!*).

ALTIN Resident

(--- *addr*)

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

ALTOUT Resident

(--- *addr*)

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

AND Resident

(n_1 n_2 --- n_3)

Leave the bitwise logical AND of n_1 and n_2 as n_3.

APPND File I/O Library [47]

(---)

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

ARG Floating Point Math Library [24]

(--- *addr*)

A constant which contains the address of the **ARG** register, `835Ch`.

ASM: Resident

(---)

Synonym for **CODE** intended to be paired with **;ASM** , a synonym for **NEXT,** . It is used as follows:

```
ASM: NEW-WORD <assembly mnemonics> ;ASM
```

See Chapter 9 The fbForth TMS9900 Assembler for details. See also **;ASM** , **CODE** and **NEXT,** .

ASSEMBLER *[immediate word]* Resident

(---)

The name of the **fbForth** 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** .

ATN Floating Point Math Library [24]

(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**), the byte count read from the current blocks file by **BLOCK** . It is included for backward compatibility with TI Forth

B/SCR Resident

(--- 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 Resident

(*addr* ---)

Calculates the backward branch offset from **HERE** to *addr* 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.

BASE Resident

(--- *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 **R->BASE** at execution time. See **R->BASE** .

BEEP Graphics Primitives Library [36]

(---)

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

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 (*addr* is compiled by **SLIT** in a colon definition).

BL Resident

(--- *char*)

A constant that leaves the ASCII value 32 (**20h**) for "blank".

BLANKS Resident

(*addr count* ---)

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

BLK Resident

(--- *addr*)

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

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:

(*bfnaddr* -14 --- *flag*)	passed by **USEBFL**
(*#blks bfnaddr* -16 --- *flag*)	passed by **MKBFL**
(*bufaddr blk#* -18 --- *flag*)	passed by **RBLK**
(*bufaddr blk#* -20 --- *flag*)	passed by **WBLK**

BLOAD Resident

(*blk* --- *flag*)

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

BLOCK Resident

(*n* --- *addr*)

Leaves the memory address of the block buffer containing block *n*. If the block is not already in memory, it is transferred from the current blocks file to whichever buffer was least recently written. If the block occupying that buffer has been marked as updated, it is written to the current blocks file before block *n* is read into the buffer. See also **BUFFER** , **R/W** , **UPDATE** and **FLUSH** .

BOOT Resident

(---)

Clears the stack, changes the radix to decimal, clears the error count, sets both **CURRENT** and **CONTEXT** to the Forth vocabulary, sets input stream to the terminal, makes the default blocks file (DSK1.FBLOCKS) current and loads block 1.

BPB Resident

(--- v*addr*)

Gets the offset in VRAM from the **fbForth** 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.

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 BSAVE -- Binary Save Routine [59]

(*addr* blk_1 --- blk_2)

Places a binary image (starting at blk_1 and going as far as necessary) of all dictionary contents between *addr* and **HERE** . The next available Forth block number blk_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 flush any dirty buffers before executing this command. Note that this is different behavior from TI Forth's **BSAVE** , which first flushes any dirty buffers. See **BLOAD** .

BUFFER Resident

(*n* --- *addr*)

Obtains the next memory buffer, assigning it to block *n*. If the contents of the buffer is marked as updated, it is written to the disk. The block is not read from the disk. The address left is the first cell within the buffer for data storage.

C! Resident

(*b addr* ---)

Stores the low-order byte (8 bits) of *b* (16-bit number on the stack) at *addr*.

C, Resident

(*b* ---)

Stores the low-order byte (8 bits) of *b* (16-bit number on the stack) into the next available dictionary byte (**HERE**), advancing the dictionary pointer one byte. This instruction should be used with caution on computers with byte-addressing, 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** .

C/L Resident

(--- *n*)

Returns on the stack the number of characters per line (stored in **C/L$**). The default value is 64 and usually represents the number of characters per line of a Forth block as it is edited (16 lines per 1024-byte block).

C/L$ Resident

(--- *addr*)

A user variable whose value is the number of characters per line. See **C/L** .

C@ Resident

(*addr* --- *b*)

Leaves the 8-bit contents *b* of memory address *addr* on the stack.

CASE *[immediate word]* Resident

Used in a colon definition to initiate the construct:

```
CASE
    n1 OF … ENDOF
    n2 OF … ENDOF
    …
ENDCASE
```

Compile time: (--- *csp* 4)

CASE gets the value *csp* of **CSP** to the stack for later restoration at the end of **ENDCASE** 's compile-time activity. It stores the current stack position in **CSP** to help **ENDCASE** track how many **OF … ENDOF** branch distances to process. It finally pushes 4 to the stack for compile-time error checking by **OF** and **ENDCASE** .

Runtime: (*n* --- *n*)

CASE itself does nothing with the number *n* on the stack; but, it must be there for **OF** or **ENDCASE** to consume. If *n* = $\mathbf{n_1}$, the code between the immediately following **OF** and **ENDOF** is executed. Execution then continues after **ENDCASE** . If *n* does not match any of the values preceding any **OF** , the code between the last **ENDOF** and **ENDCASE** is executed and may use *n*; but, one cell *must* be left for **ENDCASE** to consume or a stack underflow will result. Execution then continues after **ENDCASE** .

CFA Resident

(*pfa* --- *cfa*)

Converts the parameter field address *pfa* of a definition to its code field address *cfa* .

CHAR Graphics Primitives Library [36]

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

Defines character # *char* to have the pattern specified by the 4 numbers (n_1, n_2, n_3, n_4) on the stack. The definition for character #0 by default resides at **800h**. Each character definition is 8 bytes long with each number on the stack representing two bytes.

CHAR-CNT! File I/O Library [47]

(*n* ---)

Used in file I/O to store in the current PAB the character count of a record to be transmitted by **WRT** .

CHAR-CNT@ File I/O Library [47]

(--- *n*)

Used in file I/O to retrieve from the current PAB the character count of a record that has been read. Used by **RD** .

CHARPAT Graphics Primitives Library [36]

(*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 **800h**.

CHK-STAT File I/O Library [47]

(---)

Checks for errors following a file I/O operation. If an error has occurred, the message, "file I/O error" is displayed. If you wish to know the specific nature of the file I/O error, you can get the error code from the file's PAB to the stack with

```
HEX GET-FLAG 0E0 AND 5 SRA
```

Consult the table at error# 9 in Appendix I "Error Messages" for the specific error corresponding to the number on the stack left by the above **fbForth** code.

CLEAR Resident

(*blk* ---)

Gets a block buffer for block# *blk*, fills it with blanks and marks it as updated.

CLINE 64-Column Editor [6]

(*addr count n* ---)

Prints one line of tiny characters on the display screen. **CLINE** expects on the stack the address *addr* of the line to be written in memory, the number of characters *count* in that line, and the line number *n* on which it is to be written on the display screen. **CLINE** calls **SMASH** to do the actual work. See **SMASH** and **CLIST** .

CLIST 64-Column Editor [6]

(*blk* ---)

Lists the specified Forth block in tiny characters to the monitor. **CLIST** executes 16 calls to **CLINE** for the requisite 16 lines. See **CLINE** and **TCHAR** .

CLOAD *[immediate word]* Resident

(*blk* ---)

Used in the form:

blk **CLOAD WWWW**

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

CLR-STAT File I/O Library [47]

(---)

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

CLR_BLKS Resident

(*blk*$_1$ *blk*$_2$ ---)

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 Resident

(---)

Clears the display screen by filling the screen image table with blanks. The screen image table runs from **SCRN_START** to **SCRN_END** .

CLSE File I/O Library [47]

(---)

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

CMOVE Resident

(*addr*$_1$ *addr*$_2$ *count* ---)

Moves *count* number of bytes from *addr*$_1$ to *addr*$_2$. The contents of *addr*$_1$ is moved first, proceeding toward high memory. This is ***not*** overlap safe for *addr*$_1$ < *addr*$_2$.

CODE Resident

(---)

A defining word initializing the definition of a code (assembly) word. It sets the context vocabulary to Assembler. See Chapter 9 “The fbForth TMS9900 Assembler” for details. See also **ASM:** .

COINC Graphics Primitives Library [36]

(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 Graphics Primitives Library [36]

(--- *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 Graphics Primitives Library [36]

(*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 resets user variables to their startup values, including the dictionary pointer (to point to just after the resident dictionary) and restarts **fbForth** via **BOOT** (resets the current blocks file to the default DSK1.FBLOCKS, loading block 1) and **ABORT** , *q.v.* It may be called from the terminal to remove application programs and to restart **fbForth**.

COLOR Graphics Primitives Library [36]

(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 Graphics Primitives Library [36]

(--- *vaddr*)

A constant whose value is the beginning VDP address of the color table. The default value is **380h**.

COMPILE Resident

(---)

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 *cfa* 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 *cfa* of **WORD0** into the definition of **WORD2** .

CONSTANT Resident

(*n* ---)

A defining word used in the form:

```
n CONSTANT cccc
```

to create word **cccc** , with its parameter field containing *n*. When **cccc** 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.

COS Floating Point Math Library [24]

(f_1 --- f_2)

Calculates the cosine of f_1 radians and leaves the floating point result f_2 on the stack.

COUNT Resident

($addr_1$ --- $addr_2$ *b*)

Leave the byte address $addr_2$ and byte count *b* of the packed character string (see footnote 4 on page 17) beginning at $addr_1$. It is presumed that the first byte at $addr_1$ 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 [19]

(---)

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 ***dst-file*** is the destination blocks file.

Example:

```
CPYBLK 4 10 DSK1.FBLOCKS 25 DSK2.MYBLOCKS
4 5 6 7 8 9 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 Resident

(---)

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. Space for the parameter field is ***not*** reserved by **CREATE** . The new word is created in the **CURRENT** vocabulary. It should be noted that new word names should ***never*** exceed 31 characters in length in **fbForth**!

CSP 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

(d_1 d_2 --- d_3)

Leave the double number sum of two double numbers ($d_3 = d_1 + d_2$).

D+- Resident

(d_1 n --- d_2)

Apply the sign of n to the double number d_1, leaving it as d_2.

D. Resident

(*d* ---)

Print a signed double number from a 32-bit two's complement value *d*. The high-order 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 Resident

(d_1 --- d_2)

Leave the absolute value d_2 of a double number d_1.

DBF Resident

(--- *vaddr*)

Gets the current VRAM address *vaddr* of the default blocks filename. The value in user variable **2Ah** is the offset from the **fbForth** record buffer (address in **DISK_BUF**) for the default blocks filename. **DBF** adds this offset to the contents of **DISK_BUF** and pushes it to the stack.

DCOLOR Graphics Primitives Library [36]

(--- *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.

DDOT Graphics Primitives Library [36]

(*dotcol dotrow* --- *b vaddr*)

The assembly code routine called by **DOT** . It expects a dot column and a dot row on the stack and returns a byte *b* with only one bit set and a VDP address *vaddr*. The dot referenced by (*dotcol*,*dotrow*) is translated by **DDOT** to the address *vaddr* of the byte containing it and a mask *b* that locates the dot within the byte.

DECIMAL Resident

(---)

Set the radix in **BASE** for decimal input/output.

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

DELALL Graphics Primitives Library [36]

(---)

Delete all sprites. **DELALL** stops sprite motion, fills the sprite motion table with zeroes and stores **D0h** in the ***y*** position of all 32 sprites to leave them in an undefined state. **DELALL** does nothing to the sprite descriptor table.

DELSPR Graphics Primitives Library [36]

(*spr* ---)

Delete the specified sprite by positioning it off-screen at ***x*** = 1, ***y*** = 192; setting it to sprite pattern #0; and clearing its motion table entries.

DIGIT Resident

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

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

DEFBF Resident

(--- *addr*)

Gets the default blocks filename (DSK1.FBLOCKS) from VRAM to **PAD** and leaves the **PAD** address on the stack.

DEPTH Resident

(--- *n*)

Return the number of cells on the parameter stack. This word is used by the new command-line (**ok:*n***) response, where ***n*** indicates stack depth.

DISK_BUF Resident

(--- *addr*)

A user variable that points to the first byte in VDP RAM of the 128-byte **fbForth** record buffer.

DKB+ Resident

(*n* ---)

Defining word used to create words that calculate addresses from user variables containing offsets from **fbForth**'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 File I/O Library [47]

(---)

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.*, $d_2 = -d_1$.

DMODE Graphics Primitives Library [36]

(--- *addr*)

A variable that determines which dot mode is currently in effect. A **DMODE** value of 0 indicates DRAW mode, a value of 1 indicates UNDRAW mode and a value of 2 indicates DOT-TOGGLE mode. This variable is set by the **DRAW** , **UNDRAW** and **DTOG** words.

DO *[immediate word]* Resident

Occurs in a colon-definition in the form:

```
DO … LOOP
DO … +LOOP
```

Compile time: (*addr* 3 ---)

When compiling within the colon-definition, **DO** compiles **(DO)** , leaving the following address *addr* and the value 3 for later error checking by the compile-time action of **LOOP** or **+LOOP** .

Runtime: (*lim strt* ---)

DO begins a sequence with repetitive execution controlled by a loop limit *lim* and an index with initial value *strt*. **DO** removes these from the stack and puts them on the return stack, with the index on top. Upon reaching **LOOP** , the index is incremented by one. Until the new index equals or exceeds the limit, execution loops back to just after **DO** , otherwise the loop parameters are discarded and execution continues ahead. Both *lim* and *strt* are determined at runtime and may be the result of other operations. Within a loop, **I** will copy the current value of the index to the stack. See **I** , **LOOP** , **+LOOP** and **LEAVE** .

DOES> *[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

(---)

This is a synonym for **;CODE** , *q.v.*, intended to be paired with **;ASM** to form more readable **fbForth** Assembly language code as, for example,

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

See Chapter 9 The fbForth TMS9900 Assembler for details.

DOT Graphics Primitives Library [36]

(*dotcol dotrow* ---)

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.

DP Resident

(--- *addr*)

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

DPL Resident

(--- *addr*)

A user variable containing the number of digits to the right of the decimal point on double integer input. It may also be used to hold output column location of a decimal point in user-generated formatting. The default value on single number input is -1 for no decimal point. **DPL** is updated for every double number input.

DRAW Graphics Primitives Library [36]

(---)

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.

DSPLY File I/O Library [47]

(---)

Assigns the attribute DISPLAY to the file pointed to by **PAB-ADDR** .

DSRLNK Resident

(---)

Links an **fbForth** 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 **8356h**. 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 Graphics Primitives Library [36]

(---)

Sets **DMODE** equal to 2. This means that each dot plotted takes on the opposite state as the dot currently at that location.

DUMP Memory Dump Utility [21]

(*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** .

DUP Resident

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

Duplicates the value on top of the stack.

DXY Graphics Primitives Library [36]

($dotcol_1$ $dotrow_1$ $dotcol_2$ $dotrow_2$ --- n_1 n_2)

Places on the stack the square of the ***x*** distance n_1 and the square of the ***y*** distance n_2 between the points ($dotcol_1$,$dotrow_1$) and ($dotcol_2$,$dotrow_2$).

ECOUNT Resident

(--- *addr*)

A user variable that contains an error count. This is used to prevent error recursion.

ED@ (*EDITOR1 Vocabulary*) 40/80 Column Editor [13]

(---)

Brings you back into the 40/80-column editor on the last **fbForth** 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** block you edited. This block is pointed to by **SCR** .

EDIT (*EDITOR1 Vocabulary*) 40/80 Column Editor [13]

(*blk* ---)

Brings you into the 40/80-column editor on the specified **fbForth** 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** block, loading it from the current blocks file if necessary.

ELSE *[immediate word]* Resident

Occurs within a colon-definition in the form:

IF … ELSE … ENDIF

Compile time: ($addr_1$ n_1 --- $addr_2$ n_2)

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

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.

EMIT Resident

(*char* ---)

Transmit 7-bit ASCII character *char* to the current output device. **OUT** , *q.v.*, is incremented for each character output.

EMIT8 Resident

(*char* ---)

Transmit an 8-bit character *char* to the current output device. **OUT** , *q.v.*, is incremented for each character output.

EMPTY-BUFFERS Resident

(---)

Mark all block buffers as empty, not necessarily affecting the contents. Updated blocks are not written to the current blocks file. This is also an initialization procedure executed by **COLD** , *q.v.*, before first use of the default blocks file.

ENCLOSE Resident

($addr_1$ *char* --- $addr_1$ n_1 n_2 n_3)

The text scanning primitive used by **WORD** . From the text address $addr_1$ and an ASCII-delimiting character *char*, is determined the byte offset n_1 to the first non-delimiter 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 17) for the word, which **WORD** copies to **HERE** .

If we let each '{}' represent one character; each character is either a non-delimiter character, 'chr', a delimiter character, 'delim', or the null character, '0', **ENCLOSE** allows three possible parsing scenarios after leading delimiter characters are skipped:

1) $n_1 n_3 \{0\} n_2$
2) $n_1 \{\text{chr}\} \ldots \{\text{chr}\} n_2 n_3 \{0\}$
3) $n_1 \{\text{chr}\} \ldots \{\text{chr}\} n_2 \{\text{delim}\} n_3 \{\text{chr} \mid 0\} \ldots$

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 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.

END *[immediate word]* Resident

Compile time: (*addr* 1 ---) Runtime: (*flag* ---)

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

ENDCASE *[immediate word]* Resident

Occurs in a colon definition as the termination of the **CASE … ENDCASE** construct.

Compile time: (*csp* $addr_1$ … $addr_n$ 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 … ENDOF** clauses for which it must calculate branch distances from the addresses ($addr_1$ … $addr_n$) that each **ENDOF** left on the stack.

Runtime: (*n* ---)

If all **OF … ENDOF** clauses fail, any code after the last **ENDOF** , including **ENDCASE** , will execute. **ENDCASE** will remove the number *n* left on the stack by the failure of the last **OF** .

If you include code between the last **ENDOF** and **ENDCASE** , it must leave at least one number on the stack for **ENDCASE** to consume to prevent stack underflow. See **CASE** .

ENDIF *[immediate word]* Resident

Occurs in a colon-definition in the form:

IF … ENDIF (also **IF … THEN**)
IF … ELSE … ENDIF (also **IF … ELSE … THEN**)

Compile time: (*addr* 2 ---)

ENDIF computes the forward branch offset from *addr* to **HERE** and stores it at the spot reserved for it at *addr*. The value 2 is used for error testing.

Runtime: (---)

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

ENDOF *[immediate word]* Resident

Occurs in a colon definition as the termination of the **OF … ENDOF** construct within the **CASE … ENDCASE** construct.

Compile time: ($addr_1$ 5 --- $addr_2$ 4)

ENDOF checks for a 5 on the stack. It then compiles **BRANCH** , leaves its address $addr_2$ 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 $addr_1$ to **HERE** for its matching **OF** and stores the value at the spot reserved for it at $addr_1$.

Runtime: (---)

ENDOF causes execution to proceed after **ENDCASE** . See **OF** .

ERASE Resident

(*addr n* ---)

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

ERROR Resident

(n_1 --- n_2 n_3)

ERROR processes error notification and restarts the interpreter. **WARNING** is first examined. If **WARNING** < 1, **(ABORT)** is executed. The sole action of **(ABORT)** is to execute **ABORT** . This allows the user to (cautiously!) modify this behavior by redefining **(ABORT)** . **ABORT** clears the stacks and executes **QUIT** , which stops compilation and restarts the interpreter. If **WARNING** ≥ 0, **ERROR** leaves the contents of **IN** n_2 and **BLK** n_3 on the stack to assist in determining the location of the error. If **WARNING** > 0, **ERROR** prints the 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, the system messages are always present in **fbForth**). 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.

EXP Floating Point Math Library [24]

(f_1 --- f_2)

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

EXPECT Resident

(*addr count* ---)

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

F! Floating Point Math Library [24]

(*f addr* ---)

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

F* Floating Point Math Library [24]

($f_1 f_2$ --- f_3)

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

F+ Floating Point Math Library [24]

($f_1 f_2$ --- f_3)

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

F– Floating Point Math Library [24]

($f_1 f_2$ --- f_3)

Subtracts f_2 from f_1 and places the result on the stack ($f_1 - f_2 = f_3$).

F->S Floating Point Math Library [24]

(*f* --- *n*)

Converts a floating point number *f* on the parameter stack into a single precision number *n*.

F-D" *[immediate word]* File I/O Library [47]

(---)

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** .

F. Floating Point Math Library [24]

(*f* ---)

Prints a floating point number in Basic format to the output device.

F.R Floating Point Math Library [24]

(*f n* ---)

Prints the floating point number *f* in Basic format right justified in a field of width *n*.

F/ Floating Point Math Library [24]

($f_1 f_2$ --- f_3)

Divides f_1 by f_2 and leaves the floating point quotient f_3 on the stack. $f_1 / f_2 = f_3$.

F0< Floating Point Math Library [24]

(*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= Floating Point Math Library [24]

(*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.

F< Floating Point Math Library [24]

($f_1 f_2$ --- *flag*)

Leaves a true flag if $f_1 < f_2$, else leaves a false flag.

F= Floating Point Math Library [24]

($f_1 f_2$ --- *flag*)

Leaves a true flag if $f_1 = f_2$, else leaves a false flag.

F> Floating Point Math Library [24]

($f_1 f_2$ --- *flag*)

Leaves a flag if $f_1 > f_2$, else leaves a false flag.

F@ Floating Point Math Library [24]

(*addr* --- *f*)

Retrieves the floating point contents *f* of the given address (4 words) and places it on the stack.

FAC Floating Point Math Library [24]

(--- *addr*)

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

FAC->S Floating Point Math Library [24]

(--- *n*)

Converts a floating point number in **FAC** to a single precision number and places it on the parameter stack.

FAC> Floating Point Math Library [24]

(--- *f*)

Brings a floating point number *f* from **FAC** to the stack.

FAC>ARG Floating Point Math Library [24]

(---)

Moves a floating point number from **FAC** into **ARG** .

FADD Floating Point Math Library [24]

(---)

Adds the floating point number in **FAC** to the floating point number in **ARG** and leaves the result in **FAC** .

FDIV Floating Point Math Library [24]

(---)

Divides the floating point number in **FAC** by the floating point number in **ARG** leaving the quotient in **FAC** .

FDROP Floating Point Math Library [24]

(*f* ---)

Drops the top floating point number *f* from the stack.

FDUP Floating Point Math Library [24]

(*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 **FORGET**ting 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.

FF. Floating Point Math Library [24]

($f\ n_1\ n_2$ ---)

Prints the floating point number *f* with n_2 digits following the decimal point and a maximum of n_1 digits.

FF.R Floating Point Math Library [24]

($f\ \ n_1\ \ n_2\ \ n_3$ ---)

Prints the floating point number *f*, with n_2 digits following the decimal point, right justified in a field of width n_3 with a maximum of n_1 digits.

FILE File I/O Library [47]

(vaddr_1 *addr* vaddr_2 ---)

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*** $_1$ ***addr vaddr*** $_2$ **FILE cccc**

When **cccc** executes, **PAB-ADDR** , **PAB-BUF** and **PAB-VBUF** are set to vaddr_1, *addr* and vaddr_2, respectively.

FILES Resident

(*n* ---)

Change the number of files **fbForth** can have open simultaneously. The number of files can be 1 – 16. Each additional file requires an additional 518 bytes of upper VRAM, reducing the available VRAM for your program. Location **8370h** holds the highest available address in VRAM.

FILL Resident

(*addr count b* ---)

Fill memory beginning at *addr* with *count* bytes of byte *b*.

FIRST Resident

(--- *addr*)

A constant that leaves the address of the first (lowest) block buffer.

FIRST$ Resident

(--- *addr*)

A user variable which contains the first byte of the disk buffer area.

FLD Resident

(--- *addr*)

A user variable for control of number output field width. Presently unused in fig-Forth and **fbForth**.

FLERR Floating Point Math Library [24]

(--- *n*)

Returns on the stack the contents *n* of the floating point status register (**8354h**).

FLUSH Resident

(---)

Writes to disk all disk buffers that have been marked as updated.

FMUL Floating Point Math Library [24]

(---)

Multiplies the floating point number in **FAC** with the floating point number in **ARG** leaving the product in **FAC** .

FORGET Resident

(---)

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 **FORGET**ting 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. Because **FORTH** is immediate, it will execute during the creation of a colon definition to select this vocabulary at compile time.

FOVER Floating Point Math Library [24]

(f_1 f_2 --- f_1 f_2 f_1)

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

FRND Floating Point Math Library [24]

(--- *f*)

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

FSUB Floating Point Math Library [24]

(---)

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

FSWAP Floating Point Math Library [24]

(f_1 f_2 --- f_2 f_1)

Swaps the top two floating point numbers on the stack.

FXD File I/O Library [47]

(---)

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

GCHAR Graphics Primitives Library [36]

(*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.

GET-FLAG File I/O Library [47]

(--- *b*)

Retrieves the flag byte *b* from the current PAB and places it on the stack.

GOTOXY Resident

(*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 Enable GRAPHICS Mode [31]

(---)

Converts from present display screen mode into standard Graphics mode configurations.

GRAPHICS2 Enable GRAPHICS2 (Bitmap) Mode [33]

(---)

Converts from present display screen mode into standard Graphics2 (Bitmap) mode configuration.

HCHAR Graphics Primitives Library [36]

(*col row count char* ---)

Prints a horizontal stream of a specified character *char* beginning at (*col*,*row*) and having a length *char*. *Note:* Rows and columns are numbered from 0.

HERE Resident

(--- *addr*)

Leave the address of the next available dictionary location.

HEX Resident

(---)

Set the numeric conversion base to sixteen (hexadecimal).

HLD Resident

(--- *addr*)

A user variable that holds the address of the latest character of text during numeric output conversion.

HOLD Resident

(*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 Graphics Primitives Library [36]

(---)

Produces the sound associated with incorrect input.

I Resident

(--- *n*)

Used within a **DO** loop to copy the loop index to the stack. **I** is a synonym for **R** .

ID. Resident

(*nfa* ---)

Print a definition's name from its name field address *nfa*.

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 n*)

IF compiles **0BRANCH** and reserves space for an offset at *addr*; *addr* and *n* 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 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 Printing Routines [51]

(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** .

INPT File I/O Library [47]

(---)

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

INT Floating Point Math Library [24]

(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 Resident

(--- *addr*)

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

INTRNL File I/O Library [47]

(---)

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

ISR Resident

(--- *addr*)

A user variable that initially contains the address of the interrupt service linkage code to install an Interrupt Service Routine. The user must modify **ISR** to contain the CFA of the routine to be executed each 1/60 second. Next, the contents of **83C4h** must be modified to point to this address. Note that the interrupt service linkage code address is also available in **INTLNK** .

J Resident

(--- *n*)

Used within an inner **DO** loop to copy the loop index of the next outer **DO** loop to the stack.

JCRU Graphics Primitives Library [36]

(n_1 --- n_2)

Executed by **JOYST** when **JMODE** ≠ 0, **JCRU** allows input from joystick #1 (n_1 = 1) or #2 (n_1 = 2). The value n_2 returned will have 0 or more of the 5 least significant bits set for direction and fire-button status. Bit values are 1 = Fire, 2 = W, 4 = E, 8 = S and 16 = N. Two-bit directional combinations are 18 = NW (N + W or 16 + 2), 20 = NE, 10 = SW and 12 = SE. See § 6.8 “Using Joysticks” for more information.

JKBD Graphics Primitives Library [36]

(n_1 --- *char* n_2 n_3)

Executed by **JOYST** when **JMODE** = 0, **JKBD** allows input from joystick #1 and the left side of the keyboard (n_1 = 1) or from joystick #2 and the right side of the keyboard (n_1 = 2). Values returned are the character code *char* of the key pressed, the ***x*** status n_2 and the ***y*** status n_3. See § 6.8 “Using Joysticks” for more information.

JMODE Graphics Primitives Library [36]

(--- *addr*)

A user variable that uses offset **26h** of the user variable table. It is used by **JOYST** to determine whether to execute **JKBD** (= 0) or **JCRU** (≠ 0). The default value is 0. See **JOYST** , **JKBD** and **JCRU** .

JOYST Graphics Primitives Library [36]

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

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 ***x*** status n_2 and the ***y*** status n_3. If **JMODE** ≠ 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.

KEY Resident

(--- *char*)

Wait for the next terminal keystroke. Leave its ASCII (7-bit) value on the stack.

KEY8 Resident

(--- *char*)

Wait for the next terminal keystroke. Leave its full 8-bit value on the stack.

L/SCR Resident

(--- *n*)

Returns on the stack the number of lines per Forth block.

LATEST Resident

(--- *nfa*)

Leave the name field address *nfa* of the most recently defined word in the **CURRENT** vocabulary. At compile time, this "latest" word will be the most recently compiled word.

LCT Resident

(--- *vaddr*)

Gets the offset in VRAM from the **fbForth** record buffer (in **DISK_BUF**) for the true-lowercase table from user variable **24h**, adds the offset to the contents of **DISK_BUF** and pushes it to the stack.

LD File I/O Library [47]

(*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 [20]

(n_1 n_2 *addr* ---)

Performs a TMS9900 LDCR instruction. The CRU base address *addr* will be shifted left one bit and stored in workspace register R12 prior to executing the TMS9900

LDCR instruction. The 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 CRU documentation in the *Editor/Assembler Manual* for more information.

LEAVE Resident

(---)

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

LFA Resident

(*pfa* --- *lfa*)

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

LIMIT Resident

(--- *addr*)

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

LIMIT$ Resident

(--- *addr*)

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

LINE Graphics Primitives Library [36]

($dotcol_1$ $dotrow_1$ $dotcol_2$ $dotrow_2$ ---)

The high resolution graphics routine which plots a line from ($dotcol_1$,$dotrow_1$) to ($dotcol_2$,$dotrow_2$). **DCOLOR** and **DMODE** must be set before this instruction is used.

LIST Resident

(*blk* ---)

Lists the specified Forth block to the output device. See **PAUSE** .

LIT Resident

(--- *n*)

Within a colon-definition, **LIT** is automatically compiled before each 16-bit literal number encountered in input text. Later execution of **LIT** causes the contents of the next dictionary address to be pushed to the stack.

LITERAL *[immediate word]* Resident

INTERPRETATION: (---)

Interpretation of **LITERAL** does nothing, unlike almost all other compiling words.

COMPILE TIME: (*n* ---)

Compiles the stack value *n* as a 16-bit literal. This will execute during a colon definition. The intended use is:

```
: xxx [ calculation ] LITERAL ;
```

Compilation is suspended for the compile-time calculation of a value. Compilation is resumed and **LITERAL** compiles this value.

RUNTIME: (--- *n*)

Pushes *n* to the stack.

LOAD Resident

(*n* ---)

Begin interpretation of Forth block *n*. Loading will terminate at the end of the Forth block or at **;S** . See **;S** and **-->** .

LOG Floating Point Math Library [24]

(f_1 --- f_2 | 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.

LOOP *[immediate word]* Resident

Occurs in a colon definition in the form:

```
DO … LOOP
```

COMPILE TIME: (*addr* 3 ---)

LOOP compiles **(LOOP)** and uses *addr* to calculate an offset to **DO** . The value 3 is used for compile-time error testing.

RUNTIME: (---)

LOOP selectively controls branching back to the corresponding **DO** based on the loop index and limit. The loop index is incremented by one and compared to the limit. The branch back to **DO** occurs until the index equals or exceeds the limit. At that time, the parameters are discarded and execution continues ahead.

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.

M/ Resident

(*d* n_1 --- n_2 n_3)

A mixed magnitude math operator that leaves the signed remainder n_2 and signed quotient n_3, from a double number dividend *d* and divisor n_1. The remainder takes its sign from the dividend.

M/MOD Resident

(ud_1 u_2 --- u_3 ud_4)

An unsigned mixed magnitude math operation that leaves an unsigned double quotient ud_4 and a single remainder u_3, from a double dividend ud_1 and a single divisor u_2.

MAGNIFY Graphics Primitives Library [36]

(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

(n_1 n_2 --- n_3)

Leave the greater n_3 of the two numbers, n_1 and n_2.

MCHAR Graphics Primitives Library [36]

(*n col row* ---)

Places a square of color *n* at (*col*,*row*). Used in multicolor mode.

MENU Welcome Block [1]

(---)

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 (**msg *#n***). When **WARNING** = 0 in TI Forth, it means the disk unavailable; but, this is not necessary in **fbForth** 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 are now loaded into VRAM by **fbForth** and now use an index table loaded into low RAM as part of **fbForth**'s low-level support. The word **.LINE** , *q.v.*, can be used for this purpose.

MGT Resident

(--- *vaddr*)

Gets the offset in VRAM from the **fbForth** record buffer (in **DISK_BUF**) for the system-messages table from user variable **22h**, adds the offset to the contents of **DISK_BUF** and pushes it to the stack.

MIN Resident

(n_1 n_2 --- n_3)

Leave the smaller n_3 of the two numbers (n_1 and n_2).

MINIT Graphics Primitives Library [36]

(---)

Initializes the monitor screen for use with **MCHAR** .

MINUS Resident

(n_1 --- n_2)

Leave the two's complement n_2 of a number n_1.

MKBFL Resident

(---)

Create a blocks file from the string and number in the input stream. To create a file named MYBLOCKS on DSK1 with room for 80 blocks, type

```
MKBFL DSK1.MYBLOCKS 80
```

MOD Resident

(n_1 n_2 --- *rem*)

Leave the remainder *rem* of n_1/n_2, with the same sign as n_1.

MON Resident

(---)

Exit to the TI 99/4A color bar display screen and the system monitor program.

MOTION Graphics Primitives Library [36]

(n_1 n_2 *spr* ---)

Assigns a horizontal n_1 and vertical n_2 velocity to the specified sprite *spr*.

MOVE Resident

($addr_1$ $addr_2$ *n* ---)

Moves the contents of *n* cells (16-bit contents) beginning at $addr_1$ into *n* cells beginning at $addr_2$. The contents of $addr_1$ is moved first, proceeding toward high memory. This is ***not*** overlap safe for $addr_1 < addr_2$.

MULTI Graphics Primitives Library [36]

(---)

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.

NEXT, Resident

(---)

NEXT, should be paired with **CODE** to surround assembly code or machine code:

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

NEXT, puts **045Fh** (machine code for ALC: **B *R15**) at HERE and advances HERE. See **ASM:** , **NEXT,** , **;ASM** and **CODE** for more information. See also Chapter 9 "The fbForth TMS9900 Assembler".

NFA Resident

(*pfa* --- *nfa*)

Convert the parameter field address *pfa* of a definition to its name field address *nfa*.

NOP Resident

(---)

A do-nothing instruction. **NOP** is useful for patching as in assembly code.

NULL *[Literally* NUL *(ASCII 0)]* *[immediate word]* Resident

(---)

There is actually no word in **fbForth** with the name, ' **NULL** '. The name field for **NULL** contains an ASCII 0. Every **fbForth** buffer, including the terminal input buffer, must end with an ASCII 0. When **INTERPRET** reaches it, it will search for it in the dictionary and will find what we are here calling **NULL** . **NULL** is the only way to exit the endless loop in **INTERPRET** . When **NULL** executes, it drops the top value on the return stack and thus returns, not to **INTERPRET**, but to the word that executed **INTERPRET** (usually **QUIT** or **LOAD**). Here is its definition, keeping in mind that ' **NULL** ' represents an actual NUL (ASCII 0):

```
: NULL BLK @ IF ?EXEC THEN R> DROP ;  IMMEDIATE
```

NUMBER Resident

(*addr* --- *d*)

Convert a packed character string (see footnote 4 on page 17) left at *addr* with the character count in the first byte, to a signed double number *d* , using the current numeric base. If a decimal point is encountered in the text, its position will be given in **DPL** , but no other effect occurs. If numeric conversion is not possible, an error message will be given.

OF *[immediate word]* Resident

Occurs inside a colon definition as part of the **OF … ENDOF** construct inside of the **CASE … ENDCASE** construct.

Compile time: (4 --- *addr* 5)

Checks for the value 4 on the stack left there by **CASE** or a previous **ENDOF** , compiles **(OF)** , leaves its address *addr* for branching resolution by **ENDOF** and leaves a 5 for its matching **ENDOF** to check.

RUNTIME: (*n* --- [ ] | *n*)

The value *n* is compared to the value which was on top of the stack when **CASE** 's runtime action occurred. If the numbers are identical, the words between **OF** and **ENDOF** will be executed. Otherwise, *n* is put back on the stack for execution to continue after **ENDOF** . See **CASE** and **ENDOF** .

OPN File I/O Library [47]

(---)

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

OR Resident

(n_1 n_2 --- n_3)

Leave the bit-wise logical OR n_3 of two 16-bit values, n_1 and n_2.

OUT Resident

(--- *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 File I/O Library [47]

(---)

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

OVER Resident

(n_1 n_2 --- n_1 n_2 n_1)

Copy the second stack value n_1 to the top of the stack.

PAB-ADDR File I/O Library [47]

(--- *addr*)

A variable containing the VDP address of the first byte of the current PAB (Peripheral Access Block).

PAB-BUF File I/O Library [47]

(--- *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 File I/O Library [47]

(--- *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-VBUFF** is also stored in the file's PAB.

PABS Resident

(--- *addr*)

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

PAD Resident

(--- *addr*)

Leave the address of the text output buffer, which is a fixed offset (68 bytes in **fbForth**) above **HERE** . Every time **HERE** changes, **PAD** is updated.

PAUSE Resident

(--- *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>*** .

PDT Graphics Primitives Library [36]

(--- *vaddr*)

A constant which contains the VDP address of the Pattern Descriptor Table. Default value is **800h**.

PFA Resident

(*nfa* --- *pfa*)

Convert the name field address *nfa* of a compiled definition to its parameter field address *pfa*.

PI Floating Point Math Library [24]

(--- *f*)

A floating point approximation of π to 13 significant figures. (3.141592653590)

PREV Resident

(--- *addr*)

A user variable containing the address of the disk buffer most recently referenced. The **UPDATE** command marks this buffer to be later written to disk.

PUT-FLAG File I/O Library [47]

(*b* ---)

Writes the flag byte *b* into the appropriate PAB referenced by **PAB-ADDR** .

QUERY Resident

(---)

Input 80 characters of text (or until ***<ENTER>*** is pressed) from the operator's terminal. Text is positioned at the address contained in **TIB** with **IN** set to 0.

QUIT Resident

(---)

Clear the return stack, stop compilation and return control to the operator's terminal. No message is given, including the usual **ok:*n*** .

R Resident

(--- *n*) (R: *n* --- *n*)

Copy the top of the return stack to the parameter stack.

R# Resident

(--- *addr*)

A user variable which may contain the location of an editing cursor or other file-related function.

R->BASE Resident

(---) (R: *n* ---)

Restore the current base from the return stack. See **BASE->R** .

R/W Resident

(*addr* n_1 *flag* ---)

The fig-Forth standard disk read/write linkage. The only modification to **R/W** for **fbForth** is that it now calls **RBLK** and **WBLK** instead of the replaced **RDISK** and **WDISK** . The source or destination block buffer address is *addr*, n_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> Resident

(--- *n*) (R: *n* ---)

Remove the top value from the return stack and leave it on the parameter stack. See **>R** and **R** .

RANDOMIZE Resident

(---)

Creates an unpredictable seed for the random number generator.

RBLK Resident

(*addr blk* ---)

Read a block from the current blocks file.

RD File I/O Library [47]

(--- *count*)

The file I/O instruction that reads from the current PAB. This instruction uses **PAB-BUF** and **PAB-VBUF** .

REC-LEN File I/O Library [47]

(*b* ---)

Stores the length *b* of the record for the upcoming write into the appropriate byte in the current PAB.

REC-NO File I/O Library [47]

(*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: (*addr* 1 ---)

At compile-time, **REPEAT** compiles **BRANCH** and the offset from **HERE** to *addr*, which it stores at the space reserved for it at *addr* by **BEGIN** , *q.v.* The value 1 is used for error testing.

Runtime: (---)

At runtime, **REPEAT** forces an unconditional branch back to just after the corresponding **BEGIN** . See **WHILE** and **BEGIN** .

RLTV File I/O Library [47]

(---)

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

RND Resident

(n_1 --- n_2)

Generates a positive random integer n_2 greater than or equal to 0 and less than n_1.

RNDW Resident

(--- *n*)

Generates a random word. The value of the word may be positive or negative depending on whether the sign bit is set.

ROA Floating Point Math Library [24]

(--- *addr*)

Variable array (32 bytes) to temporarily hold VDP Rollout area (**3C0h** – **3DFh**).

ROA> Floating Point Math Library [24]

(---)

Restore VDP Rollout Area from **ROA** .

ROT Resident

(n_1 n_2 n_3 --- n_2 n_3 n_1)

Rotate the top three values on the stack, bringing the third n_1 to the top.

RP! Resident

(---)

A procedure to initialize the return stack pointer from user variable **R0** .

RSTR File I/O Library [47]

(*n* ---)

Restores the file whose PAB is pointed to by the current PAB to the specified record number *n*.

S->D Resident

(*n* --- *d*)

Sign-extend a single number *n* to form a double number *d*.

S->F Floating Point Math Library [24]

(*n* --- *f*)

Converts a single-precision number *n* on the stack to a floating point number *f.*

S->FAC Floating Point Math Library [24]

(*n* ---)

Takes a single-precision number *n* from the stack, converts it to floating point, and leaves it in FAC.

S0 Resident

(--- *addr*)

User variable that points to the base of the parameter stack. Pronounced "s zero". See **SP!** .

SATR Graphics Primitives Library [36]

(--- *vaddr*)

A constant whose value *vaddr* is the VDP address of the Sprite Attribute List. Default value is **300h**.

SBO CRU Words [20]

(*addr* ---)

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

SBZ CRU Words [20]

(*addr* ---)

This word expects to find on the stack the CRU address *addr* of the bit to be reset to 0. **SBZ** will put this address into workspace register R12, shift it left (double it) and execute TMS9900 instruction, **0 SBZ**, to effect resetting the bit. See CRU documentation in the *Editor/Assembler Manual* for more information.

SCMP CPYBLK -- Block Copying Utility [19]

(str_1 str_2 --- -1 | 0 | +1)

Compares two strings with leading byte counts pointed to by str_1 and str_2 and leaves the result on the stack: -1, if $str_1 < str_2$; 0, if $str_1 = str_2$; +1, if $str_1 > str_2$.

SCR Resident

(--- *addr*)

A user variable containing the Forth block number most recently referenced by **LIST** or **EDIT** .

SCREEN Resident

(*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 *addr* of the byte immediately following the last byte of the display screen image table to be used as the logical display screen.

SCRN_START Resident

(--- *addr*)

A user variable containing the address *addr* of the first byte of the display screen image table to be used as the logical display screen.

SCRN_WIDTH Resident

(--- *addr*)

A user variable which contains the number of characters that will fit across the display screen. (32 or 40) Used by the display screen scroller.

SEED Resident

(*n* ---)

Places a new seed *n* into the random number generator.

SET-PAB File I/O Library [47]

(---)

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

SETFL Floating Point Math Library [24]

(f_1 f_2 ---)

Performs **>FAC** on f_2 and **>ARG** on f_1.

SGN Resident

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

Returns the sign of *n* or 0.

SIGN Resident

(*n d* --- *d*)

Stores a minus sign (ASCII 45 or **2Dh**) at the current location in a converted numeric output string in the text output buffer if *n* is negative. At the time *n* is evaluated, it is discarded; but, double number *d* is maintained for continued conversion until **#>** removes it from the stack. Must be used between **<#** and **#>** . Using **SIGN** implies that *d* can be negative, which means that *d* should be used to produce *n*. You should then replace *d* with its absolute value (|*d*|) on the stack by using **DABS** . This can be done by pushing *d* to the stack and executing **SWAP OVER DABS** : (*d* --- *n* |*d*|) prior to **<# … SIGN … #>** .

SIN Floating Point Math Library [24]

(f_1 --- f_2)

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 *addr* of the string following it on the stack. It modifies the top of the return stack to point to just after the string.

SMASH 64-Column Editor [6]

($addr_1$ $count_1$ *n* --- $addr_2$ *vaddr* $count_2$)

The assembly code routine that formats a line of tiny characters. It expects the address $addr_1$ 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 character count $count_2$. See **CLIST** and **CLINE** .

SMTN Graphics Primitives Library [36]

(--- *vaddr*)

A constant whose value is the VDP address of the Sprite Motion Table. Default value is **780h**.

SMUDGE Resident

(---)

Used during word definition to toggle the smudge bit in 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 ;
```

SP! Resident

(---)

A procedure to initialize the parameter stack pointer from **S0** , the user variable that points to the base of the parameter stack.

SP@ Resident

(--- *addr*)

This word returns the address of the top of the stack as it was before **SP@** was executed, *e.g.*, **1 2 SP@ @ . . .** would type 2 2 1.

SPACE Resident

(---)

Transmit a blank character (ASCII 32|**20h**) to the output device.

SPACES Resident

(*n* ---)

Transmit *n* blank characters (ASCII 32|**20h**) to the output device.

SPCHAR Graphics Primitives Library [36]

(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.

SPDTAB Graphics Primitives Library [36]

(--- *vaddr*)

A constant whose value is the VDP address of the Sprite Descriptor Table. Default value is `800h`. Notice that this coincides with the Pattern Descriptor Table.

SPLIT Enable SPLIT and SPLIT2 Modes [34]

(---)

Converts from present display screen mode into standard Split mode configuration.

SPLIT2 Enable SPLIT and SPLIT2 Modes [34]

(---)

Converts from present display screen mode into standard Split2 mode configuration.

SPRCOL Graphics Primitives Library [36]

(*n* *spr* ---)

Changes color of the given sprite number *spr* to the color *n* specified.

SPRDIST Graphics Primitives Library [36]

(spr_1 spr_2 --- *n*)

Returns on the stack the square of the distance *n* between two specified sprites, spr_1 and spr_2. Distance is measured in pixels and the maximum distance that can be detected accurately is 181 pixels.

SPRDISTXY Graphics Primitives Library [36]

(*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.

SPRGET Graphics Primitives Library [36]

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

Returns the dot column *dotcol* and dot row *dotrow* position of sprite *spr*.

SPRITE Graphics Primitives Library [36]

(*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 Graphics Primitives Library [36]

(*char spr* ---)

Changes the character pattern of a given sprite *spr* to *char*.

SPRPUT Graphics Primitives Library [36]

(*dotcol dotrow spr* ---)

Places a given sprite *spr* at location (*dotcol*,*dotrow*).

SQNTL File I/O Library [47]

(---)

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

SQR Floating Point Math Library [24]

(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 ;
```

SRC Resident

(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 ;
```

SRL Resident

(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 ;
```

SSDT Graphics Primitives Library [36]

(*vaddr* ---)

Places the Sprite Descriptor Table at the specified VDP address *vaddr* and initializes all sprite tables. The address given must be on an even 2K boundary. This instruction must be executed before sprites can be used.

STAT File I/O Library [47]

(--- *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.

STATE Resident

(--- *addr*)

A user variable containing the compilation state. A non-zero value indicates compilation. The value itself may be implementation dependent.

STCR CRU Words [20]

(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` **`0Fh`**. If $n_1 = 0$, 16 bits will be transferred. For program clarity, you may certainly use $n_1 = 16$ to transfer 16 bits because $n_1 = 0$ will be the value actually used by the final machine code. See CRU documentation in the *Editor/Assembler Manual* for more information.

STR Floating Point Math Library [24]

(---)

Converts the number in the FAC to a string, which is placed in PAD. The string is in Basic format. Used by **F.** and **F.R** .

STR. Floating Point Math Library [24]

(n_1 n_2 n_3 ---)

Converts the number in the FAC to a string which is placed in PAD. The maximum number of output digits is n_1 (**STR.** places n_1 in the byte at FAC+11). Calling **STR.** with $n_1 = 0$ is identical to calling **STR** . The number of significant digits of output is n_2 (**STR.** places n_2 in the byte at FAC+12). The number of digits to be output after the decimal point is n_3 (**STR.** places n_3 in the byte at FAC+13). See the GPL STR routine on page 254 in the *Editor/Assembler Manual* for more detail.

SV File I/O Library [47]

(*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 Resident

(n_1 n_2 --- n_2 n_1)

Exchange the top two values on the stack.

SWCH Printing Routines [51]

(---)

A special purpose word which permits **EMIT** to output characters to an RS232 device rather than to the screen. See **UNSWCH** .

SWPB Resident

(n_1 --- n_2)

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.

SYSTEM Resident

(*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.

TAN Floating Point Math Library [24]

(f_1 --- f_2)

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** 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** 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 **FORGET**ting **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.

TB CRU Words [20]

(*addr* --- *flag*)

TB performs the TMS9900 **TB** instruction. The bit at CRU address *addr* is tested by this instruction. Its value (*flag* = 1|0) is returned to the stack. The CRU base address *addr* will be shifted left one bit and stored in workspace register R12 prior to executing the TMS9900 **TB** instruction. See CRU documentation in the *Editor/Assembler Manual* for more information.

TCHAR 64-Column Editor [6]

(--- *addr*)

Points to the array that holds the tiny character definitions for the 64-column editor. See **CLIST** .

TEXT Enable TEXT & TEXT80 Modes [30]

(---)

Converts from present display screen mode into standard Text mode configuration.

TEXT80 Enable TEXT & TEXT80 Modes [30]

(---)

Converts from present display screen mode into Text80 mode configuration if your computer has that facility.

THEN *[immediate word]* Resident

(---)

An alias for **ENDIF** .

TIB Resident

(--- *addr*)

A user variable containing the address of the terminal input buffer.

TLC Resident

(*vaddr* ---)

Loads true lowercase to *vaddr* in VRAM and patches the ‘0’ pattern to a slashed zero from storage in VRAM.

TOGGLE Resident

(*addr b* ---)

Complement (XOR) the contents of the byte at *addr* by the bit pattern of byte *b*.

TRACE TRACE -- Colon Definition Tracing [23]

(---)

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

(*addr*$_1$ *n* --- *addr*$_2$)

Traverse the name field of a fig-Forth variable-length name field. The starting point *addr*$_1$ 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 *addr*$_2$ points to the other end of the name.

TRIAD Printing Routines [51]

(*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/fig-Forth extension".

TRIADS Printing Routines [51]

(*blk*$_1$ *blk*$_2$ ---)

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.

TROFF TRACE -- Colon Definition Tracing [23]

(---)

Turn off tracing of words compiled with the **TRACE** option. See **TRON** , **TRACE** , **UNTRACE** and § 5.4 .

TRON TRACE -- Colon Definition Tracing [23]

(---)

Turn on tracing of words compiled with the **TRACE** option. See **TROFF** , **TRACE** , **UNTRACE** and § 5.4 .

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* Resident

(u_1 u_2 --- *ud*)

Leave the unsigned double number product *ud* of two unsigned numbers, u_1 and u_2.

U. Resident

(*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_1 --- *rem* *quot*)

Leave the unsigned remainder *rem* and unsigned quotient *quot* from the unsigned double dividend *ud* and unsigned divisor u_1.

U0 Resident

(--- *addr*)

A user variable that points to the base of the user variable area.

U< Resident

(u_1 u_2 --- *flag*)

Leaves a true flag if u_1 is less than u_2, else leaves a false flag.

UCONS$ Resident

(--- *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 *ud* to the output device.

UD.R Resident

(*ud* *n* ---)

Prints an unsigned double number *ud* right justified in a field of length *n*.

UNDRAW Graphics Primitives Library [36]

(---)

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.

UNSWCH Printing Routines [51]

(---)

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 *addr*, which it stores at the space reserved for it at *addr* by **BEGIN** , *q.v*. The value 1 is used for error testing.

Runtime: (*flag* ---)

UNTIL controls the conditional branch back to the corresponding **BEGIN** . If *flag* is *false*, execution returns to just after **BEGIN** ; if *true*, execution continues ahead.

UNTRACE TRACE -- Colon Definition Tracing [23]

(---)

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 File I/O Library [47]

(---)

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

(---)

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**

USER Resident

(*n* ---)

A defining word used in the form:

n **USER cccc**

which creates a user variable **cccc** . The parameter field of **cccc** contains *n* as a fixed offset relative to the user variable base address pointed to by workspace register UP (R8) for this user variable. When **cccc** is later executed, it places the sum of its offset and the user area base address on the stack as the storage address of that particular variable. You should only use the even numbers **66h** – **7Eh** for *n*—enough for 13 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 Floating Point Math Library [24]

(---)

Causes the string at PAD to be converted into a floating point number and put into the FAC. The string must have a leading length byte with no embedded blanks.

VAND Resident

(*b vaddr* ---)

Performs a logical AND on the byte at the specified VDP location *vaddr* and the given byte *b*. The result byte is stored back into the VDP address.

VARIABLE Resident

(*n* ---)

A defining word used in the form:

n **VARIABLE cccc**

When **VARIABLE** is executed, it creates the definition **cccc** with its parameter field initialized to *n*. When **cccc** is later executed, the address of its parameter field (containing *n*) is left on the stack, so that a fetch or store may access this location.

VCHAR Graphics Primitives Library [36]

(*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.

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.

VFILL Resident

(*vaddr count b* ---)

Fills *count* locations beginning at the given VDP address *vaddr* with the specified byte *b*.

VLIST Memory Dump Utility [21]

(---)

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** .

VMBR Resident

(*vaddr addr count* ---)

Reads *count* bytes beginning at the given VDP address *vaddr* and places them at *addr*.

VMBW Resident

(*addr vaddr count* ---)

Writes *count* bytes from *addr* into VDP beginning at the given VDP address *vaddr*.

VMOVE Resident

($vaddr_1$ $vaddr_2$ *n* ---)

Move a block of *n* bytes of VRAM from $vaddr_1$ to $vaddr_2$, 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:

```
VOCABULARY  cccc
```

to create a vocabulary definition **cccc** . Subsequent use of **cccc** will make it the **CONTEXT** vocabulary which is searched first by **INTERPRET** . The sequence **cccc**

DEFINITIONS will also make **cccc** the **CURRENT** vocabulary into which new definitions are placed.

cccc will be so chained as to include all definitions of the vocabulary in which **cccc** is itself defined. All vocabularies ultimately chain to Forth. By convention, vocabulary names are to be declared **IMMEDIATE** . See **VOC-LINK** .

VOR Resident

(*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 File I/O Library [47]

(---)

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

VSBR Resident

(*vaddr* --- *b*)

Reads a single byte from the given VDP address *vaddr* and places its value *b* on the stack.

VSBW Resident

(*b* *vaddr* ---)

Writes a single byte *b* into the given VDP address *vaddr*.

VWTR Resident

(*b* *n* ---)

Writes the given byte *b* into the specified VDP write-only register *n*.

VXOR Resident

(*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*.

WARNING Resident

(--- *addr*)

A user variable initialized by **COLD** at system startup containing a value controlling messages. If **WARNING** > 0, a disk is present (not relevant in **fbForth**). If **WARNING** = 0, no disk is present and messages will be presented by number (**msg** ***#n***). 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** , **ERROR** .

WBLK Resident

(*addr* *blk* ---)

Write a block to the current blocks file.

WHERE (*EDITOR1 Vocabulary*) 40/80 Column Editor [13]

(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*) 64-Column Editor [6]

(n_1 n_2 ---)

When an error occurs on a **LOAD** instruction, typing **WHERE** will bring you into the 64-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.

WHILE *[immediate word]* Resident

Occurs in a colon-definition in the form:

BEGIN … WHILE (true part) **… REPEAT**

Compile time: ($addr_1$ n_1 --- $addr_1$ n_1 $addr_2$ n_2)

WHILE emplaces **(0BRANCH)** and leaves $addr_2$ of the reserved offset. The stack values will be resolved by **REPEAT** .

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.

WLITERAL *[immediate word]* Resident

(---)

A compiling word which compiles **SLIT** and the string which follows **WLITERAL** into the dictionary.

Used in the form: **WLITERAL cccc**

WORD Resident

(*char* ---)

Read the text characters from the input stream being interpreted until a delimiter *char* is found, storing the packed character string (see footnote 4 on page 17) beginning at the dictionary buffer **HERE** . **WORD** leaves the character count in the first byte followed by the input characters and ends with two or more blanks. Leading occurrences of *char* are ignored. If **BLK** is zero, text is taken from the terminal input buffer, otherwise from the disk block stored in **BLK** . See **BLK** , **IN** .

WRT File I/O Library [47]

(*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*.

XOR Resident

(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 **]** .

[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 **xxxx** executes rather than at compile time.

] Resident

(---)

Resume compilation to the completion of a colon definition. See **[** .

^ Floating Point Math Library [24]

(f_1 f_2 --- f_3)

Returns f_3 on the stack f_1 raised to the f_2 power. The operands must be floating point numbers.

Appendix E Differences between fbForth and TI Forth

This appendix will detail **fbForth** changes from TI Forth. This will include words that have been added, removed, re-purposed and deprecated. All of those words, except those removed, will also be discussed elsewhere in the manual where appropriate, including the **fbForth** Glossary. Even some of the removed words will be discussed elsewhere as necessary. Words that have been hoisted into the kernel (resident dictionary) will also be discussed.

E.1 TI Forth Words not in fbForth

Descriptions of words appearing in the comments here that are part of **fbForth** may be found in Appendix D "The fbForth Glossary".

!"	
(!")	
-64SUPPORT	Now type **MENU** for options: **6 LOAD**
-ASSEMBLER	Now type **MENU** for options: **53 LOAD**
-BSAVE	Now type **MENU** for options: **59 LOAD**
-CODE	Words loaded are now part of resident dictionary.
-COPY	**CPYBLK** replaces contents. Now type **MENU** for options: **19 LOAD**
-CRU	Now type **MENU** for options: **20 LOAD**
-DUMP	Now type **MENU** for options: **21 LOAD**
-EDITOR	Now type **MENU** for options: **13 LOAD**
-FILE	Now type **MENU** for options: **47 LOAD**
-FLOAT	Now type **MENU** for options: **24 LOAD**
-GRAPH	Now type **MENU** for options: **36 LOAD**
-GRAPH1	Now type **MENU** for options: **31 LOAD**
-GRAPH2	Now type **MENU** for options: **33 LOAD**
-MULTI	Now type **MENU** for options: **32 LOAD**
-PRINT	Now type **MENU** for options: **51 LOAD**
-SPLIT	Now type **MENU** for options: **34 LOAD**
-SYNONYMS	Words loaded are now part of resident dictionary except **FORMAT-DISK** , which has been removed.
-TEXT	Now type **MENU** for options: **30 LOAD**

-TRACE	Now type **MENU** for options: **23 LOAD**
-VDPMODES	Now type **MENU** for options: **4 LOAD**
B/BUF$	User variable no longer used.
B/SCR$	User variable no longer used.
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	
FORMAT-DISK	
FORTH-COPY	
FORTH_LINK	User variable no longer used. Its function is part of **FORTH** (Forth vocabulary declaration word).
OFFSET	User variable no longer used.
RDISK	Replaced by **RBLK** .
SCOPY	Replaced by **CPYBLK** .
SCRTCH	Never should have been implemented!
SMOVE	Replaced by **CPYBLK** .
WDISK	Replaced by **WBLK** .

E.2 New and Modified Words in fbForth

Descriptions of words listed here may be found in Appendix D "The fbForth Glossary".

(UB) Runtime word for **USEBFL** .

.S Moved to resident dictionary.

;ASM Part of resident dictionary. Synonym for **NEXT,** (TMS9900 Assembly Language equivalent: **B *R15** or **B *NEXT**)

;CODE Moved to resident dictionary.

<CLOAD> Moved to resident dictionary.

>ROA Saves VDP Rollout Area to array **ROA** .

ASM: Part of resident dictionary. Synonym for **CODE** .

B/BUF 1024

B/SCR 1

BFLNAM Part of resident dictionary. Helper routine that gets a blocks filename from the input stream.

BLKRW Part of the resident dictionary. Blocks I/O utility routine called by **DO_BRW** .

BLOCK Modified to accommodate file-based block I/O.

BOOT Modified to accommodate the use of blocks files and to load block 1 of the default blocks file instead of block 3 as in TI Forth.

BPB Part of resident dictionary. Gets address in VRAM of blocks file PABs.

BSAVE Modified to first empty buffers instead of flushing.

CLEAR Modified to accommodate unconditional **B/SCR** = 1.

CLOAD Moved to resident dictionary.

CLR_BLKS Part of resident dictionary. Clears a range of blocks.

CLS Moved to resident dictionary.

CODE Moved to resident dictionary.

COLD Modified to accommodate modified **FORTH** .

CPYBLK Replaces **SCOPY** and **SMOVE** . Copies a range of blocks from one blocks file to the same or a different blocks file.

DBF Part of resident dictionary. Gets address in VRAM of default blocks filename.

DEFBF Part of resident dictionary. Gets default blocks filename to PAD and leaves PAD address.

DEPTH New word to report stack depth. Part of resident dictionary.

DISK_BUF

DKB+ New defining word. Part of resident dictionary. Used to create words that calculate addresses from user variables containing offsets from **fbForth**'s record buffer (see **DISK_BUF**).

DOES>ASM: Part of resident dictionary. Synonym for **;CODE** .

DO_BRW Part of resident dictionary. Helper routine that executes **BLKRW** and processes returned flag.

DSRLNK Moved to resident dictionary.

DUMP Modified cosmetically and to accommodate text80 mode.

FILES Part of resident dictionary. Sets number of simultaneous files allowed open.

FORTH Modified Word. No longer uses removed user variable **FORTH_LINK** .

GPLLNK Moved to resident dictionary.

LCT Part of resident dictionary. Gets address in VRAM of true lowercase table.

MENU Modified word. Displays menu of load options when executed.

MESSAGE Modified word. Displays designated resident system error message. It can no longer be used to display user messages.

MGT Part of resident dictionary. Gets address in VRAM of system error messages table.

MKBFL New word. Part of resident dictionary. Creates a new blocks file to specified size.

MON Modified and moved to resident dictionary.

NEXT, Moved to resident dictionary.

R/W Modified to call **WBLK** and **RBLK** instead of the removed **RDISK** and **WDISK** .

RANDOMIZE Moved to resident dictionary.

RBLK Reads a block from the current blocks file.

RND Moved to resident dictionary.

RNDW Moved to resident dictionary.

ROA 32-byte array for temporarily holding the VDP Rollout Area.

ROA> Restores VDP Rollout Area from array **ROA** .

SCMP Compares 2 strings resulting in -1 | 0 | +1 on the stack.

SCREEN Moved to resident dictionary.

SEED Moved to resident dictionary.

SGN Leaves sign of number on stack: -1 | 0 | +1

SLIT Moved to resident dictionary.

TEXT80 Sets up 80-column text mode on systems so equipped.

TLC New word to load true lowercase to designated VRAM and patch ‘0’ pattern to display it with a slash for clarity.

USEBFL Part of resident dictionary. Changes the current blocks file.

VAND Moved to resident dictionary.

VDPMDE Moved to resident dictionary.

VFILL Moved to resident dictionary.

VLIST Modified to accommodate text80 mode.

VMBR Moved to resident dictionary.

VMBW Moved to resident dictionary.

VMOVE Part of resident dictionary. Moves a block of VRAM from one place in VRAM to another.

VOR Moved to resident dictionary.

VSBR Moved to resident dictionary.

VSBW Moved to resident dictionary.

VWTR Moved to resident dictionary.

VXOR Moved to resident dictionary.

WBLK Writes a block to the current blocks file.

WLITERAL Moved to resident dictionary.

XMLLNK Moved to resident dictionary.

Appendix F User Variables in fbForth

The purpose of this appendix is to detail the User Variables in **fbForth** 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 **66h** through **7Eh** to create his/her own user variables. See the definition of **USER** in Appendix D .

F.1 **fbForth** *User Variables (Address Offset Order)*

Name	Offset	Initial Value	Description
UCONS$	6h	3944h	Base of User Var initial value table
S0	8h	FFA0h	Base of Stack
R0	Ah	3FFEh	Base of Return Stack
U0	Ch	3980h	Base of User Variables
TIB	Eh	FFA0h	Terminal Input Buffer address
WIDTH	10h	31	Name length in dictionary
DP	12h	BC80h	Dictionary Pointer
SYS$	14h	348Eh	Address of System Support
CURPOS	16h	0	Cursor location in VDP RAM
INTLNK	18h	3424h	Pointer to Interrupt Service Linkage
WARNING	1Ah	1	Message Control
C/L$	1Ch	64	Characters per Line
FIRST$	1Eh	2010h	Beginning of Disk Buffers
LIMIT$	20h	3424h	End of Disk Buffers
[no name]	22h	80h	Sys. Msg Table offset from FRB. **MGT** gets address.
[no name]	24h	19Ch	Lowercase Table offset from FRB. **LCT** gets address.
JMODE	26h		Used after graphics primitives are loaded for whether **JOYST** executes **JKBD** or **JCRU**
[available]	28h		—available for storage—
[no name]	2Ah	324h	Def. Blocks Filename offset from FRB. **DBF** gets address.
DISK_BUF	2Ch	1000h	VDP location of 128B Forth Record Buffer (FRB)
PABS	2Eh	460h	VDP location for PABs
SCRN_WIDTH	30h	40	Display Screen Width in Characters
SCRN_START	32h	0	Display Screen Image Start in VDP
SCRN_END	34h	960	Display Screen Image End in VDP
ISR	36h	3424h	Interrupt Service Pointer
ALTIN	38h	0	Alternate Input Pointer
ALTOUT	3Ah	0	Alternate Output Pointer
VDPMDE	3Ch	1	VDP Mode
[no name]	3Eh	298h	Blocks PABs offset from FRB. **BPB** gets address.
BPOFF	40h	0	Current Blocks file offset from **BPB**. (**0** or **70h**)
FENCE	42h		Dictionary Fence
BLK	44h		Block being interpreted

Name	Offset	Initial Value	Description
IN	**46h**		Byte offset in text buffer
OUT	**48h**		Incremented by **EMIT**
SCR	**4Ah**		Last Forth Block (Screen) referenced
CONTEXT	**4Ch**		Pointer to Context Vocabulary
CURRENT	**4Eh**		Pointer to Current Vocabulary
STATE	**50h**		Compilation State
BASE	**52h**		Number Base for Conversions
DPL	**54h**		Decimal Point Location
FLD	**56h**		Field Width (unused)
CSP	**58h**		Stack Pointer for error checking
R#	**5Ah**		Editing Cursor location
HLD	**5Ch**		Holds address during numeric conversion
USE	**5Eh**		Next Block Buffer to Use
PREV	**60h**		Most recently accessed disk buffer
ECOUNT	**62h**		Error control
VOC-LINK	**64h**		Vocabulary linkage
[*user to define*]	**66h**		—*available to user*—
[*user to define*]	**68h**	1	—*available to user*—
[*user to define*]	**6Ah**		—*available to user*—
[*user to define*]	**6Ch**		—*available to user*—
[*user to define*]	**6Eh**		—*available to user*—
[*user to define*]	**70h**		—*available to user*—
[*user to define*]	**72h**		—*available to user*—
[*user to define*]	**74h**		—*available to user*—
[*user to define*]	**76h**		—*available to user*—
[*user to define*]	**78h**		—*available to user*—
[*user to define*]	**7Ah**		—*available to user*—
[*user to define*]	**7Ch**		—*available to user*—
[*user to define*]	**7Eh**		—*available to user*—

F.2 **fbForth** *User Variables (Variable Name Order)*

Name	Offset	Initial Value	Description
ALTIN	38h	0	Alternate Input Pointer
ALTOUT	3Ah	0	Alternate Output Pointer
BASE	52h		Number Base for Conversions
BLK	44h		Block being interpreted
BPOFF	40h	0	Current Blocks file offset from **BPB**. (0 or 70h)
C/L$	1Ch	64	Characters per Line
CONTEXT	4Ch		Pointer to Context Vocabulary
CSP	58h		Stack Pointer for error checking
CURPOS	16h	0	Cursor location in VDP RAM
CURRENT	4Eh		Pointer to Current Vocabulary
DISK_BUF	2Ch	1000h	VDP location of 128B Forth Record Buffer (FRB)
DP	12h	BC80h	Dictionary Pointer
DPL	54h		Decimal Point Location
ECOUNT	62h		Error control
FENCE	42h		Dictionary Fence
FIRST$	1Eh	2010h	Beginning of Disk Buffers
FLD	56h		Field Width (unused)
HLD	5Ch		Holds address during numeric conversion
IN	46h		Byte offset in text buffer
INTLNK	18h	3424h	Pointer to Interrupt Service Linkage
ISR	36h	3424h	Interrupt Service Pointer
JMODE	26h		Used after graphics primitives are loaded for whether **JOYST** executes **JKBD** or **JCRU**
LIMIT$	20h	3424h	End of Disk Buffers
OUT	48h		Incremented by **EMIT**
PABS	2Eh	460h	VDP location for PABs
PREV	60h		Most recently accessed disk buffer
R#	5Ah		Editing Cursor location
R0	Ah	3FFEh	Base of Return Stack
S0	8h	FFA0h	Base of Stack
SCR	4Ah		Last Forth Block (Screen) referenced
SCRN_END	34h	960	Display Screen Image End in VDP
SCRN_START	32h	0	Display Screen Image Start in VDP
SCRN_WIDTH	30h	40	Display Screen Width in Characters
STATE	50h		Compilation State
SYS$	14h	348Eh	Address of System Support
TIB	Eh	FFA0h	Terminal Input Buffer address
U0	Ch	3980h	Base of User Variables
UCONS$	6h	3944h	Base of User Var initial value table
USE	5Eh		Next Block Buffer to Use
VDPMDE	3Ch	1	VDP Mode
VOC-LINK	64h		Vocabulary linkage
WARNING	1Ah	1	Message Control

Name	Offset	Initial Value	Description
WIDTH	**10h**	**31**	Name length in dictionary
[available]	**28h**		—available for storage—
[no name]	**22h**	**80h**	Sys. Msg Table offset from FRB. **MGT** gets address.
[no name]	**24h**	**19Ch**	Lowercase Table offset from FRB. **LCT** gets address.
[no name]	**2Ah**	**324h**	Def. Blocks Filename offset from FRB. **DBF** gets address.
[no name]	**3Eh**	**298h**	Blocks PABs offset from FRB. **BPB** gets address.
[*user to define*]	**66h**		—*available to user*—
[*user to define*]	**68h**	**1**	—*available to user*—
[*user to define*]	**6Ah**		—*available to user*—
[*user to define*]	**6Ch**		—*available to user*—
[*user to define*]	**6Eh**		—*available to user*—
[*user to define*]	**70h**		—*available to user*—
[*user to define*]	**72h**		—*available to user*—
[*user to define*]	**74h**		—*available to user*—
[*user to define*]	**76h**		—*available to user*—
[*user to define*]	**78h**		—*available to user*—
[*user to define*]	**7Ah**		—*available to user*—
[*user to define*]	**7Ch**		—*available to user*—
[*user to define*]	**7Eh**		—*available to user*—

Appendix G fbForth 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, TRACE -- Colon Definition Tracing, depends on the words loaded by option, Memory Dump Utility, If, by chance, the prerequisite words were already in the dictionary at the time you type **23 LOAD** , they would not be loaded again. This is called a conditional load.

G.1 Option: 40/80 Column Editor

Starting screen: 13

Prerequisite options loaded: Must manually load block 30 ("Enable TEXT & TEXT80 Modes") and execute **TEXT80** to operate editor in 80-column mode.

Words loaded: **EDIT** **ED@** **WHERE**

G.2 Option: 64-Column Editor

Starting screen: 6

Prerequisite options loaded:
Graphics Primitives Library
Enable TEXT & TEXT80 Modes
Enable GRAPHICS2 (Bitmap) Mode
Enable SPLIT and SPLIT2 Modes

Words loaded: **EDIT** **ED@** **WHERE**
CLIST **CLINE**

G.3 Option: CPYBLK -- Block Copying Utility

Starting screen: 19

Words loaded: **SCMP** **CPYBLK**

G.4 Option: Memory Dump Utility

Starting screen: 21

Words loaded: **DUMP** **.S** **VLIST**

G.5 Option: TRACE -- Colon Definition Tracing

Starting screen: 23

Prerequisite options loaded: Memory Dump Utility

Words loaded:

```
TRACE       UNTRACE     TRON
TROFF       : (alternate)
```

G.6 Option: Floating Point Math Library

Starting screen: 24

Words loaded:

```
FDUP       FDROP      FOVER
FSWAP      F!         F@
>FAC       SETFL      FADD
FMUL       F+         F-
F*         F/         S->FAC
FAC->S     FAC>ARG    F->S
S->F       FRND       STR
STR.       VAL        F$
>F         F.R        F.
FF.R.      FF.        F0<
F0=        F>         F=
F<         FLERR      ?FLERR
INT        ^          SQR
EXP        LOG        COS
SIN        TAN        ATN
PI         ROA        >ROA
ROA>
```

G.7 Option: File I/O Library

Starting screen: 47

Words loaded:

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

G.8 Option: Printing Routines

Starting screen: 51

Prerequisite options loaded: File I/O Library

Words loaded: **SWCH** **UNSWCH** **?ASCII**
TRIAD **TRIADS** **INDEX**

G.9 Option: TMS9900 Assembler

Starting screen: 53

Words loaded: Entire Assembler vocabulary. See Chapter 9.

G.10 Option: BSAVE -- Binary Save Routine

Starting screen: 59

Words loaded: **BSAVE**

G.11 Option: CRU Words

Starting screen: 20

Words loaded: **SBO** **SBZ** **TB**
LDCR **STCR**

G.12 Option: Enable TEXT & TEXT80 Modes

Starting screen: 30

Words loaded: **TEXT** **TEXT80**

G.13 Option: Enable GRAPHICS Mode

Starting screen: 31

Words loaded: **GRAPHICS**

G.14 Option: Enable MULTIcolor Mode

Starting screen: 32

Words loaded: **MULTI**

G.15 Option: Enable GRAPHICS2 (Bitmap) Mode

Starting screen: 33

Words loaded: **GRAPHICS2**

G.16 Option: Enable SPLIT and SPLIT2 Modes

Starting screen: 34

Prerequisite options loaded: Enable GRAPHICS2 (Bitmap) Mode

Words loaded: **SPLIT** **SPLIT2**

G.17 Option: Enable all of the above VDP Modes

Starting screen: 4

Prerequisite options loaded: Enable TEXT & TEXT80 Modes
Enable GRAPHICS Mode
Enable MULTIcolor Mode
Enable GRAPHICS2 (Bitmap) Mode
Enable SPLIT and SPLIT2 Modes

G.18 Option: Graphics Primitives Library

Starting screen: 36

Words loaded:

CHAR **CHARPAT** **VCHAR**
HCHAR **COLOR** **SCREEN**
GCHAR **SSDT** **SPCHAR**
SPRCOL **SPRPAT** **SPRPUT**
SPRITE **MOTION** **#MOTION**
SPRGET **DXY** **SPRDIST**
SPRDISTXY **MAGNIFY** **JOYST**
COINC **COINCXY** **COINCALL**
DELSPR **DELALL** **MINIT**
MCHAR **DRAW** **UNDRAW**
DTOG **DOT** **LINE**

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:

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

These words have been coded in hexadecimal in FBLOCKS, thus they do not require that the **fbForth** Assembler be in memory before they can be loaded. Their Assembly source code (written in **fbForth** TMS9900 Assembler) is listed on the following pages.

Block 45 needs a little clarification:

1. It should be noted that the definition of **TCHAR** in block 45 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 46 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 46 of FBLOCKS. The following code will do the trick:

USEBFL DSK1.MYBLOCKS	<== Make MYBLOCKS current
45 LOAD	<== Load **TCHAR**
USEBFL DSK1.FBLOCKS	<== Make FBLOCKS current
TCHAR 46 BLOCK 192 + 194 MOVE	<== Copy **TCHAR** to block 46, line 3
FLUSH	<== Flush block to FBLOCKS
FORGET TCHAR	<== Recover space in dictionary used by **TCHAR**

2. The comment, **(^0)** (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 ( Source for CRU words...R12 is CRU register)   BASE->R  HEX
  1 ASM: SBO  ( addr --- )
  2    *SP+ R12 MOV,
       R12 R12 A,
  3    0 SBO,
    ;ASM

  4 ASM: SBZ  ( addr --- )
  5    *SP+ R12 MOV,
       R12 R12 A,
  6    0 SBZ,
    ;ASM

  7 ASM: TB  ( addr --- flag )
  8    *SP R12 MOV,
       R12 R12 A,
  9    *SP CLR,
       0 TB,
 10    EQ IF,
 11       *SP INC,
 12    THEN,
 13 ;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,
  4    NE IF,
  5       R1 0008 CI,
  6       LTE IF,
  7          R0 SWPB,
  8       THEN,
  9    THEN,
 10    R1 06 SLA,
       R1 3000 ORI,
       R1 X,
 11 ;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,
  5    R2 R2 MOV,
  6       NE IF,
  7          R02 0008 CI,
  8          LTE IF,
  9             R0 SWPB,
 10          THEN,
 11       THEN,
 12    R0 *SP MOV,
 13 ;ASM
 14
 15 R->BASE

BLOCK #43
  0 ( Source for DDOT )                              BASE->R HEX
  1  8040 VARIABLE DTAB 2010 , 0804 , 0201 , 7FBF , DFEF ,
  2                     F7FB , 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 ( Source for SMASH )                          BASE->R HEX
  1 0 VARIABLE TCHAR 17E ALLOT 43 BLOCK TCHAR 180 CMOVE
  2 TCHAR 7C - CONSTANT TC 0 VARIABLE LB FE ALLOT
  3 ASM: SMASH ( addr #char line# --- lb vaddr cnt )
  4    *SP+ R1 MOV,
       *SP+ R2 MOV,
       *SP R3 MOV,
       R4 LB LI,
       R4 *SP MOV,
  5    SP DECT,
       R1 SWPB,
       R1 2000 AI,
       R1 *SP MOV,
       R2 R1 MOV,
       R1 INC,
  6    R1 FFFE ANDI,
       SP DECT,
       R1 2 SLA,
       R1 *SP MOV,
       R3 R2 A,
  7    BEGIN,
          R2 R3 C,
  8    GT WHILE,
          R5 CLR,
          R6 CLR,
          *R3+ R5 MOVB,
  9       *R3+ R6 MOVB,
          R5 6 SRL,
          R6 6 SRL,
 10       BEGIN,
             TC @(R5) R0 MOV,
             TC @(R6) R1 MOV,
             R1 4 SRC,
             R12 4 LI,
 11          BEGIN,
                R0 R11 MOV,
                R11 F000 ANDI,
                R1 R7 MOV,
                R7 F00 ANDI,
 12             R11 R7 SOC,
                R7 *R4+ MOVB,
                R0 C SRC,
                R1 C SRC,
                R12 DEC,
 13          EQ UNTIL,
             R5 INCT,
             R6 INCT,
             R5 R12 MOV,
             R12 2 ANDI,
 14       EQ UNTIL,
 15    REPEAT,
    ;ASM                                      R->BASE
```

```
BLOCK #45
( definitions of tiny chars with true lowercase) BASE->R HEX
0EEE  VARIABLE TCHAR EEEE ,
0000 , 0000 , (  )  0444 , 4404 , ( !)  0AA0 , 0000 , ( ")
08AE , AEA2 , ( #)  04EC , 46E4 , ( $)  0A24 , 448A , ( %)
06AC , 4A86 , ( &)  0480 , 0000 , ( ')  0248 , 8842 , ( ()
0842 , 2248 , ( ^0) 04EE , 4000 , ( *)  0044 , E440 , ( +)
0000 , 0048 , ( ,)  0000 , E000 , ( -)  0000 , 0004 , ( .)
0224 , 4488 , ( /)  04AA , EAA4 , ( 0)  04C4 , 4444 , ( 1)
04A2 , 488E , ( 2)  0C22 , C22C , ( 3)  02AA , AE22 , ( 4)
0E8C , 222C , ( 5)  0688 , CAA4 , ( 6)  0E22 , 4488 , ( 7)
04AA , 4AA4 , ( 8)  04AA , 622C , ( 9)  0004 , 0040 , ( :)
0004 , 0048 , ( ;)  0024 , 8420 , ( <)  000E , 0E00 , ( =)
0084 , 2480 , ( >)  04A2 , 4404 , ( ?)  04AE , AE86 , ( @)
04AA , EAAA , ( A)  0CAA , CAAC , ( B)  0688 , 8886 , ( C)
0CAA , AAAC , ( D)  0E88 , C88E , ( E)  0E88 , C888 , ( F)
-->

BLOCK #46
( definitions of tiny chars with true lowercase continued)
04A8 , 8AA6 , ( G)  0AAA , EAAA , ( H)  0E44 , 444E , ( I)
0222 , 22A4 , ( J)  0AAC , CAAA , ( K)  0888 , 888E , ( E)
0AEE , AAAA , ( M)  0AAE , EEAA , ( N)  0EAA , AAAE , ( 0)
0CAA , C888 , ( P)  0EAA , AAEC , ( Q)  0CAA , CAAA , ( R)
0688 , 422C , ( S)  0E44 , 4444 , ( T)  0AAA , AAAE , ( U)
0AAA , AA44 , ( V)  0AAA , AEEA , ( W)  0AA4 , 44AA , ( X)
0AAA , E444 , ( Y)  0E24 , 488E , ( Z)  0644 , 4446 , ( [)
0884 , 4422 , ( \)  0C44 , 444C , ( ])  044A , A000 , ( $)
0000 , 000F , ( _)  0420 , 0000 , ( `)  000E , 2EAE , ( a)
088C , AAAC , ( b)  0006 , 8886 , ( c)  0226 , AAA6 , ( d)
0004 , AE86 , ( e)  0688 , E888 , ( f)  0006 , A62C , ( g)
088C , AAAA , ( h)  0404 , 4442 , ( i)  0202 , 22A4 , ( j)
088A , ACAA , ( k)  0444 , 4444 , ( l)  000A , EEAA , ( m)
0008 , EAAA , ( n)  0004 , AAA4 , ( o)  000C , AC88 , ( p)
-->

BLOCK #47
( definitions of tiny chars with true lowercase concluded)
0006 , A622 , ( q)  0008 , E888 , ( r)  0006 , 842C , ( s)
044E , 4442 , ( t)  000A , AAA6 , ( u)  000A , AAA4 , ( v)
000A , AEEA , ( w)  000A , A4AA , ( x)  000A , A62C , ( y)
000E , 248E , ( z)  0644 , 8446 , ( {)  0444 , 0444 , ( |)
0C44 , 244C , ( })  02E8 , 0000 , ( ~)  0EEE , EEEE , ( DEL)
R->BASE ;S
```

BLOCK #48

```
( Source for JCRU used by JOYST for CRU access to joysticks)
BASE->R   HEX
ASM: JCRU  ( joystick# --- value )
   *SP R1 MOV,     ( get unit number)
   R1 5 AI,        ( use keyboard select 6 for #1, 7 for #2)
   R1 SWPB,
   R12 24 LI,
   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                                       R->BASE
```

Appendix I Error Messages

Error#	Message	Probable Causes
1	empty stack	Procedure being executed attempts to pop a number off the parameter stack when there is no number on the parameter stack. The error may have occurred long before it is detected because Forth checks for this condition only when control returns to the outer interpreter.
2	dictionary full	The user dictionary space is full. Too many definitions have been compiled.
4	isn't unique	This message is more a warning than an error. It informs the user that a word with the same name as the one just compiled is already in the **CURRENT** or **CONTEXT** vocabulary.
5	FBLOCKS not current	This message is displayed when **fbForth** needs to read from the system blocks file, FBLOCKS, and the user has made another blocks file current with **USEBFL** . This is likely the result of executing **MENU** without FBLOCKS current.
6	disk error	This has several possible causes: No disk in disk drive, disk not initialized, disk drive or controller not connected properly, disk drive or controller not plugged in. The diskette may be damaged with some sector having a hard error.
7	full stack	The procedure being executed is leaving extra unwanted numbers on the parameter stack resulting in a stack overflow.
8	block # out of range	A block # has been requested from the current blocks file that is less than 1 or greater than the number of blocks in the file.
9	file I/O error	Any file I/O operation which results in an error will return this message. The **GET-FLAG** instruction will fetch the flag/status byte (PAB + 1). The high-order 3 bits contain the error code, which can be obtained with **HEX GET-FLAG 0E0 AND 5 SRA** . An error code of 0 indicates no error only if the COND bit (bit 2) of the GPL status byte located at **837Ch** is ***not*** set.

code	meaning
00	Bad device name
01	Device is write protected
02	Bad open attribute

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. **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 **: xx 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** TMS9900 Assembler.

Appendix J Contents of FBLOCKS

The contents of the **fbForth** 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**, as well. (*cf.* Appendix E “Differences between fbForth and TI Forth”)

Note that blocks number from 1 in **fbForth** rather than 0 as in TI Forth. There are also five blank blocks (blocks 5, 60 – 63), which you can use as you wish.

Note, also, that the following file is dated 22DEC2013.

```
BLOCK #1  ( old TIF #3)
( fbForth WELCOME SCREEN---LES 22DEC2013)
BASE->R HEX  04F 7 VWTR
CLS  0 0 GOTOXY  ." Booting fbForth..." CR
10 83C2 C! ( QUIT OFF! )

: MENU 1 BLOCK 2+ @ 6662 - 5 ?ERROR 2 LOAD ;
CLS   0 0 GOTOXY
." fbForth 1.0         (c) 2013 Lee Stewart"
."  ...a file-based TI Forth implementation"
CR ." FBLOCKS mod: 22DEC2013"
CR CR ." Type MENU for load options." CR CR  R->BASE  ;S






BLOCK #2
CLS 0 0 GOTOXY ." Load Options--Page 1:       fbForth 1.0" CR CR
 ." Description                  Load Block" CR
 ." ---------------------------------------" CR
 ." 40/80 Column Editor..................13" CR
 ." 64-Column Editor......................6" CR
 ." CPYBLK -- Block Copying Utility......19" CR
 ." Memory Dump Utility..................21" CR
 ." TRACE -- Colon Definition Tracing....23" CR
 ." Floating Point Math Library..........24" CR
 ." File I/O Library.....................47" CR
 ." Printing Routines....................51" CR
 ." TMS9900 Assembler....................53" CR
 ." BSAVE -- Binary Save Routine.........59" CR
 ." CRU Words............................20" CR CR
 ." Type <block> LOAD to load." CR
 CR ."      Tap any key for next page..." KEY DROP -->
```

BLOCK #3

```


CLS 0 0 GOTOXY ." Load Options--Page 2:       fbForth 1.0" CR CR
 ." Description                  Load Block" CR
 ." ---------------------------------------" CR
 ." Enable TEXT & TEXT80 Modes...........30" CR
 ." Enable GRAPHICS Mode.................31" CR
 ." Enable MULTIcolor Mode...............32" CR
 ." Enable GRAPHICS2 (Bitmap) Mode.......33" CR
 ." Enable SPLIT and SPLIT2 Modes........34" CR
 ." Enable all of the above VDP Modes.....4" CR
 ." Graphics Primitives Library..........36" CR CR
 ." Type <block> LOAD to load." CR CR    ;S



```

BLOCK #4

```
( All VDP modes )
BASE->R  DECIMAL  CR ." loading all VDP modes"
30 LOAD 31 LOAD 32 LOAD 33 LOAD 34 LOAD
R->BASE    ;S
```

BLOCK #5

```

```

```
BLOCK #6  ( old TIF #22)
( 64 COLUMN EDITOR )   0 CLOAD ED@             BASE->R
DECIMAL 36 R->BASE CLOAD LINE BASE->R DECIMAL 30 R->BASE
CLOAD TEXT BASE->R DECIMAL 33 R->BASE CLOAD GRAPHICS2 BASE->R
DECIMAL 34 R->BASE CLOAD SPLIT
BASE->R DECIMAL 44 R->BASE CLOAD CLIST
BASE->R HEX      CR ." loading 64-column editor"
VOCABULARY EDITOR2 IMMEDIATE EDITOR2 DEFINITIONS
  0 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 ;
: BCK 0 0 GOTOXY QUIT ;   ( <--This line can be removed)
: PTR CUR @ SCR @ BLOCK + ;
: R/C CUR @ C/L /MOD ; ( --- col row )    R->BASE  -->

```

```
BLOCK #7  ( old TIF #23)
( 64 COLUMN EDITOR )  BASE->R HEX                 ." ."

: CINIT 3800 DUP ' SPDTAB ! 800 / 6 VWTR 1B00 ' SATR !
  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 ;

R->BASE -->
```

```
BLOCK #8  ( old TIF #24)
( 64 COLUMN EDITOR ) BASE->R                      ." ."

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)
( 64 COLUMN EDITOR )  BASE->R  DECIMAL            ." ."
: -TAB PTR DUP C@ BL >
  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)
( 64 COLUMN EDITOR )  BASE->R                              ." ."
 DECIMAL
: !BLK PTR  C! UPDATE  ;
: BLNKS PTR R/C DROP C/L SWAP - 32 FILL ;
: HOME  0 0 CGOTOXY ;
: REDRAW SCR @ CLIST UPDATE .CUR ;
: SCRNO CLS 0 0 GOTOXY ." BLOCK #" SCR @ BASE->R DECIMAL U.
  R->BASE CR ;
: +SCR SCR @ 1+ DUP SCR ! SCRNO CLIST  ;
: -SCR SCR @ 1- 0 MAX DUP SCR ! SCRNO  CLIST ;
: DEL PTR DUP 1+ SWAP R/C DROP C/L SWAP - CMOVE 32
  PTR R/C DROP - C/L + 1- C! ;
: INS 32 PTR DUP R/C DROP C/L SWAP - + SWAP DO
  I C@ LOOP DROP PTR DUP R/C DROP C/L SWAP - + 1- SWAP 1- SWAP
  DO I C! -1 +LOOP ;   R->BASE  -->


BLOCK #11  ( old TIF #27)
( 64 COLUMN EDITOR  15JUL82 LAO )       BASE->R DECIMAL  ." ."
0 VARIABLE BLINK  0 VARIABLE OKEY
10 CONSTANT RL  150 CONSTANT RH  0 VARIABLE KC RH VARIABLE RLOG
: RKEY BEGIN ?KEY -DUP 1 BLINK +! BLINK @ DUP 60 < IF 6 0 SPRPAT
 ELSE 5 0 SPRPAT ENDIF    120 = IF 0 BLINK ! ENDIF
        IF ( SOME KEY IS PRESSED )    KC @   1 KC +!  0 BLINK !
          IF ( WAITING TO REPEAT )   RLOG @  KC @  <
             IF ( LONG ENOUGH ) RL RLOG ! 1 KC ! 1 ( FORCE EXT)
             ELSE OKEY @ OVER =
               IF DROP 0   ( NEED TO WAIT MORE )
               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
   UNTIL DUP OKEY !      ;                 R->BASE -->
```

```
BLOCK #12  ( old TIF #28 & 29)
  0 ( 64 COLUMN EDITOR )  BASE->R HEX                   ." ."
  1 : EDT  VDPMDE @ 5 = 0= IF SPLIT ENDIF CINIT !CUR R/C CGOTOXY
  2   DUP DUP SCR ! SCRNO CLIST BEGIN RKEY  CASE 08 OF LEFT ENDOF
  3   0C OF -SCR ENDOF  0A OF DOWN ENDOF  03 OF DEL RELINE ENDOF
  4   0B OF UP ENDOF  04 OF INS RELINE ENDOF  09 OF RIGHT ENDOF
  5   07 OF DELLIN REDRAW ENDOF  06 OF INSLIN REDRAW ENDOF
  6   0E OF HOME  ENDOF  02 OF +SCR  ENDOF  16 OF TAB ENDOF
  7   0D OF 1 +LIN .CUR PLACE ENDOF 1E OF INSLIN BLNKS REDRAW ENDOF
  8   01 OF DELHALF BLNKS RELINE ENDOF  7F OF -TAB ENDOF
  9   0F OF 5 0 SPRPAT CLS SCRNO DROP 300 ' SATR ! QUIT ENDOF
 10  DUP 1F > OVER 7F < AND IF DUP !BLK R/C SWAP DROP DUP SCR @
 11  (LINE) ROT CLINE 1 +.CUR ELSE  7 EMIT  ENDIF ENDCASE AGAIN ;
 12 FORTH DEFINITIONS     : EDIT EDITOR2 0 EDT ;
 13 : WHERE EDITOR2 SWAP 2- EDT ;
 14 : ED@ EDITOR2 SCR @ SCRNO EDIT ;
 15 CR CR ." See Manual for usage." CR   R->BASE

BLOCK #13  ( old TIF #34)
  0 ( SCREEN EDITOR 09JUL82 LCT---mod 27OCT2013 LES) 0 CLOAD ED@
  1                       CR ." loading 40/80-column editor"
  2 BASE->R   HEX VOCABULARY EDITOR1 IMMEDIATE EDITOR1 DEFINITIONS
  3   0 VARIABLE OLDCUR 6 ALLOT  : VM VDPMDE @ ;
  4   : GETCUR 8F0 OLDCUR 8 VMBR ; : PUTCUR OLDCUR 8F0 8 VMBW ;
  5   : BOX 8F7 8F1 DO 84 I VSBW LOOP 0FC 8F0 VSBW 0FC 8F7 VSBW ;
  6   : CUR R# ; : !CUR 0 MAX 3FF MIN CUR ! ;
  7   : +CUR CUR @ + !CUR ; : +LIN CUR @ C/L / + C/L * !CUR ;
  8   0 VARIABLE S_H     DECIMAL
  9   : FTYPE 3 + 40 VM 0= IF 2 * THEN * 3 + SWAP VMBW ;
 10   : LISTA BASE->R DECIMAL 0 0 GOTOXY DUP SCR ! ."  BLOCK #"
 11      . CR CR CR 16 0 DO I 2 .R ." |" CR LOOP R->BASE ;
 12   : ROWCAL S_H @ IF 29 + ENDIF ;  ( only called in 40-col mode)
 13   : LINE. DO I SCR @ (LINE) VM IF DROP ROWCAL 35 THEN I FTYPE
 14   LOOP ;   : LISTB L/SCR 0 LINE. ;
 15 R->BASE -->

BLOCK #14  ( old TIF #35)
  0 ( SCRN ED 09JUL82 LCT) : XY CURPOS @ SCRN_WIDTH @ /MOD ; ." ."
  1 BASE->R  DECIMAL  : VL 19 3 DO OVER OVER I GOTOXY EMIT LOOP
  2 DROP DROP ;  : HEADR VM IF 3 ELSE 32 THEN >R R 1 GOTOXY
  3 ."  3         4         5         6   " R 2 GOTOXY
  4 ." -0----+----0----+----0----+----0---+" 124 XY DROP 1- VL
  5 VM IF 60 2 VL THEN R 19 GOTOXY CURPOS @ 35 45 VFILL R> 35 + 19
  6 GOTOXY ." +" ;        : HEADL LISTA 2 1  GOTOXY
  7 ."  0         1         2         3     " 2 2 GOTOXY
  8 ." +0----+----0----+----0----+----0----+"  2 19 GOTOXY ." +"
  9  CURPOS @ 35 45 VFILL VM IF 62 38 VL ELSE HEADR THEN ;
 10 : LISTL HEADL 0 S_H ! LISTB ; : LISTR DROP HEADR 1 S_H !
 11  LISTB ;  : BCK  0 L/SCR 4 + GOTOXY PUTCUR QUIT ;
 12 : PTR CUR @ SCR @ BLOCK + ;   ( --- addr )
 13 : R/C CUR @ C/L /MOD ;  ( --- col row )
 14 : DELHALF PAD 64 BLANKS PTR PAD C/L R/C DROP - CMOVE ;
 15 R->BASE  -->
```

BLOCK #15 (old TIF #36)

```
( SCREEN EDITOR 12JUL82 LCT) BASE->R  DECIMAL      ." ."
  : .CUR R/C 3 + SWAP 3 + VM IF DUP S_H @
    IF 31 > IF 29 - ELSE SCR @ LISTL THEN
    ELSE 38 < 0= IF SCR @ LISTR 29 - THEN
    THEN THEN SWAP GOTOXY ;
  : 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 BL 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 13 EMIT LINE. UPDATE .CUR ;
: +.CUR +CUR .CUR ;    : ~CUR1023? CUR @ 1023 < ;
: TAB ~CUR1023? IF PTR DUP C@ BL > IF BEGIN ~CUR1023? IF 1+ DUP
1 +CUR C@ BL = ELSE 1 THEN UNTIL ENDIF BEGIN ~CUR1023? IF 1+ DUP
 1 +CUR C@ BL > ELSE 1 THEN UNTIL DROP .CUR THEN ; R->BASE -->
```

BLOCK #16 (old TIF #37)

```
( SCREEN EDITOR 12JUL82 LCT)  BASE->R  DECIMAL         ." ."
: -TAB CUR @ IF PTR DUP C@ BL > IF BEGIN CUR @ IF 1- DUP -1
 +CUR C@ BL = ELSE 1 THEN UNTIL ENDIF BEGIN CUR @ IF 1- DUP -1
 +CUR C@ BL > ELSE 1 ENDIF UNTIL BEGIN CUR @ IF 1- DUP -1 +CUR
C@ BL = DUP IF 1 +.CUR ENDIF ELSE 1 ENDIF UNTIL DROP .CUR THEN ;
 : !BLK PTR C! UPDATE 1 +.CUR ;
: BLNKS PTR R/C DROP C/L SWAP - BL FILL ;      : FLIP VM IF
   S_H @ IF -29 ELSE 29 THEN +.CUR ELSE 0 CUR ! .CUR THEN ;
: REDRAW SCR @ S_H @ IF LISTR ELSE LISTL ENDIF UPDATE .CUR ;
: NEWSCR 0 SWAP LISTL !CUR .CUR ;
: +SCR SCR @ 1+ NEWSCR ;     : -SCR SCR @ 1- 0 MAX NEWSCR ;
: DEL PTR DUP 1+ SWAP R/C DROP C/L SWAP - CMOVE BL
  PTR R/C DROP - C/L + 1- C! ;
: INS BL PTR DUP R/C DROP C/L SWAP - + SWAP DO
  I C@ LOOP DROP PTR DUP R/C DROP C/L SWAP - + 1- SWAP 1- SWAP
  DO I C! -1 +LOOP ;     R->BASE -->
```

BLOCK #17 (new block)

```
( 40 COLUMN EDITOR  19OCT2013 LES mod)   BASE->R DECIMAL  ." ."
0 VARIABLE BLINK  0 VARIABLE OKEY  0 VARIABLE CURCHR
: GCH CURPOS @ VSBR CURCHR ! ;   : PCH CURCHR @ CURPOS @ VSBW ;
: PCUR 30 CURPOS @ VSBW ;
10 CONSTANT RL  150 CONSTANT RH  0 VARIABLE KC RH VARIABLE RLOG
: RKEY BEGIN ?KEY -DUP 1 BLINK +! BLINK @ DUP 60 < IF
   PCUR ELSE PCH ENDIF    120 = IF 0 BLINK ! ENDIF
        IF ( SOME KEY IS PRESSED )    KC @   1 KC +!   0 BLINK !
          IF ( WAITING TO REPEAT )   RLOG @  KC @  <
             IF ( LONG ENOUGH ) RL RLOG ! 1 KC ! 1 ( FORCE EXT)
             ELSE OKEY @ OVER = IF DROP 0 ( NEED TO WAIT MORE )
               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
   UNTIL DUP OKEY ! PCH     ;                R->BASE -->
```

```
BLOCK #18  ( old TIF #38)
( SCREEN EDITOR 12JUL82 LCT) BASE->R HEX          ." ."
: VED GETCUR BOX SWAP CLS LISTL !CUR .CUR BEGIN GCH RKEY CASE
  0F OF BCK             ENDOF  01 OF DELHALF BLNKS RELINE ENDOF
  08 OF -1 +.CUR        ENDOF  02 OF +SCR                 ENDOF
  0A OF C/L +.CUR       ENDOF  0C OF -SCR                 ENDOF
  0B OF C/L MINUS +.CUR ENDOF  03 OF DEL RELINE           ENDOF
  09 OF 1 +.CUR         ENDOF  04 OF INS RELINE           ENDOF
  0D OF 1 +LIN .CUR     ENDOF  07 OF DELLIN REDRAW        ENDOF
  0E OF FLIP            ENDOF  06 OF INSLIN REDRAW        ENDOF
  1E OF INSLIN BLNKS REDRAW ENDOF 16 OF TAB             ENDOF
  7F OF -TAB ENDOF
  DUP 1F > OVER 7F < AND IF DUP EMIT DUP !BLK ELSE 7 EMIT ENDIF
  ENDCASE AGAIN ; FORTH DEFINITIONS
: WHERE EDITOR1 SWAP 2- VED ;
: EDIT EDITOR1 0 VED ;  : ED@ EDITOR1 SCR @ EDIT ;
   CR CR ." See Manual for usage." CR   R->BASE

BLOCK #19  ( old TIF #39)
( Block Copy   10NOV2013 LES )         CR CR
." CPYBLK copies a range of blocks to the  same or another file,
 e.g.,"  CR CR ."    CPYBLK 5 8 DSK1.F1 9 DSK2.F2" CR CR ." will
 copy blocks 5-8 from DSK1.F1 to" CR ." DSK2.F2 starting at bloc
k 9." CR CR   BASE->R  DECIMAL 0 CLOAD CPYBLK 0 VARIABLE SFL
0 VARIABLE DFL  0 CONSTANT XD : SCMP  OVER C@ OVER C@ OVER OVER
 - SGN >R MIN 1+ 0 SWAP 1 DO DROP OVER I + C@ OVER I + C@ - SGN
 DUP IF LEAVE THEN LOOP R> OVER 0= IF OR ELSE DROP THEN SWAP
 DROP SWAP DROP ; : GNUM BL WORD HERE NUMBER DROP ; : GBFL HERE
 0 BFLNAM SWAP ! ; : CPYBLK 1 ' XD ! HERE BPB BPOFF @ + 9 + DUP
 VSBR 1+ HERE SWAP DUP =CELLS ALLOT VMBR GNUM GNUM OVER OVER >
 IF SWAP THEN OVER - 1+ >R SFL GBFL GNUM DFL GBFL SFL @ DFL @
 SCMP 0= IF OVER OVER - DUP 0< SWAP R MINUS > + 2 = IF SWAP R +
 1- SWAP R + 1- -1 ' XD ! THEN THEN CR R> 0 DO OVER DUP . OVER
 SFL @ (UB) SWAP BLOCK 2- ! DFL @ (UB) UPDATE FLUSH XD + SWAP
 XD + SWAP LOOP DROP DROP DUP (UB) DP ! ;  R->BASE

BLOCK #20  ( old TIF #88)
( CRU WORDS   12OCT82 LAO )   0 CLOAD STCR
                           CR ." loading CRU words"
BASE->R  HEX
CODE SBO  C339 , A30C , 1D00 , NEXT,
CODE SBZ  C339 , A30C , 1E00 , NEXT,
CODE TB   C319 , A30C , 04D9 , 1F00 , 1601 , 0599 , NEXT,

CODE LDCR C339 , A30C , C079 , C039 , 0241 , 000F , 1304 ,
          0281 , 0008 , 1501 , 06C0 , 0A61 , 0261 , 3000 ,
          0481 , NEXT,

CODE STCR C339 , A30C , C059 , 04C0 , 0241 , 000F , C081 ,
          0A61 , 0261 , 3400 , 0481 , C082 , 1304 , 0282 ,
          0008 , 1501 , 06C0 , C640 , NEXT,

  CR ." See Manual for usage." CR R->BASE
```

```
BLOCK #21  ( old TIF #42)
( DUMP ROUTINES 12JUL82 LCT...18NOV2013 LES mod)
0 CLOAD VLIST   BASE->R HEX  CR ." loading memory dump utility"
: VM+ VDPMDE @ 0= IF + ELSE DROP THEN ;
: 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 #22  ( old TIF #43)
( DUMP ROUTINES 12JUL82 LCT...18NOV2013 LES mod)        ." ."
: 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
  ENDIF SWAP DUMP8 DROP ;
( .S has been put in resident dictionary)
( maybe should put VLIST there, as well)
: VLIST 80 OUT ! CONTEXT @ @ 0 SWAP ( start counter)
    BEGIN DUP C@ 3F AND OUT @ + SCRN_WIDTH @ 3 - >
      IF CR 0 OUT ! ENDIF
      DUP ID.
      SWAP 1+ SWAP    ( increment counter)
      PFA LFA @ SPACE DUP 0= PAUSE OR
    UNTIL DROP CR . ." words listed" ;  R->BASE


```

```
BLOCK #23  ( old TIF #44 )
( TRACE COLON WORDS-FORTH DIMENSIONS III/2 P.58 26OCT82 LCT)
0 CLOAD (TRACE)      CR ." loading colon definition tracing"
FORTH DEFINITIONS
0 VARIABLE TRACF   ( CONTROLS INSERTION OF TRACE ROUTINE )
0 VARIABLE TFLAG   ( CONTROLS TRACE OUTPUT )
: TRACE 1 TRACF ! ;
: UNTRACE 0 TRACF ! ;
: TRON 1 TFLAG ! ;
: TROFF 0 TFLAG ! ;
: (TRACE) TFLAG @       ( GIVE TRACE OUTPUT? )
  IF  CR R 2- NFA ID.  ( BACK TO PFA NFA FOR NAME )
      .S ENDIF ;        ( PRINT STACK CONTENTS )
: :  ( 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 #24  ( old TIF #45)
( FLOATING POINT <4 WORD> STACK ROUTINES 12JUL82 LCT)
0 CLOAD PI     CR ." loading floating point library"
BASE->R HEX   0 VARIABLE ROA 1E ALLOT ( rollout temp storage)
: FDUP SP@ DUP 2- SWAP 6 + DO I @ -2 +LOOP ;
: FDROP DROP DROP DROP DROP ;  : >ROA 3C0 ROA 20 VMBR ;
: FOVER SP@ DUP 6 + SWAP E + DO I @ -2 +LOOP ;
: FSWAP FOVER >R >R >R >R >R >R >R >R
        FDROP R> R> R> R> R> R> R> R> ;
: F! 4 0 DO DUP >R ! R> 2+ LOOP DROP ;
: F@ 6 + 4 0 DO DUP >R @ R> 2- LOOP DROP ;
834A CONSTANT FAC  835C CONSTANT ARG
: >FAC FAC F! ;   : >ARG ARG F! ;  : FAC> FAC F@ ;
: SETFL >FAC >ARG ;    : ROA> ROA 3C0 20 VMBW ;
: FADD 0600 C SYSTEM ;  : FSUB 0700 C SYSTEM ;
: FMUL 0800 C SYSTEM ;  : FDIV 0900 C SYSTEM ;
R->BASE  -->
```

```
BLOCK #25  ( old TIF #46)
( FLOATING POINT ARITHMETIC ROUTINES 12JUL82 LCT)
BASE->R HEX                               ." ."
: F+ SETFL FADD FAC> ;
: F- SETFL FSUB FAC> ;
: F* SETFL FMUL FAC> ;
: F/ SETFL FDIV FAC> ;
: S->FAC FAC ! 2300 C SYSTEM ;
: FAC->S 1200 C SYSTEM FAC @ ;
: FAC>ARG FAC ARG 8 CMOVE ;
: F->S >FAC FAC->S ;
: S->F S->FAC FAC> ;
DECIMAL
: FRND 3 0 DO 100 RND 100 RND 256 * + LOOP
  100 RND 16128 + ;

R->BASE -->
```

```
BLOCK #26  ( old TIF #47)
( FLOATING POINT CONVERSION ROUTINES CONTINUED 12JUL82 LCT)
BASE->R HEX                                      ." ."
: DOSTR FAC B + C! >ROA 14 GPLLNK ROA>
  FAC B + C@ 8300 + FAC C + C@ DUP PAD C!
  PAD 1+ SWAP CMOVE ;

( NUMBER IN FAC CONVERTED TO BASIC STRING AND PLACED AT PAD)
: STR 0 DOSTR ;

( NUMBER IN FAC CONVERTED TO FIXED STRING AND PLACED AT PAD)
: STR. FAC D + C! FAC C + C! DOSTR ;

( STRING AT PAD CONVERTED TO NUMBER IN FAC)
: VAL PAD 1+ DISK_BUF @ DUP FAC C + ! PAD C@ OVER OVER + 20
  SWAP VSBW VMBW 1000 XMLLNK ;
R->BASE -->
```

```
BLOCK #27  ( old TIF #48)
( FLOATING POINT - COMPILE NO TO STACK 12JUL82 LCT) BASE->R HEX
                                                  ." ."
: F$ PAD 1+ SWAP >R R CMOVE R> PAD C! VAL FAC> ;
: (>F) R COUNT DUP 1+ =CELLS R> + >R F$ ;
: >F 20 STATE @
    IF   COMPILE (>F) WORD HERE C@
         1+ =CELLS ALLOT
    ELSE WORD HERE COUNT F$
    ENDIF ; IMMEDIATE
( FLOATING POINT OUTPUT ROUTINES )
: JST PAD C@ - SPACES PAD COUNT TYPE ;
: F.R >R >FAC STR R> JST ;
: F. 0 F.R ;
: FF.R >R >R >R >FAC R> 0 R> STR. R> JST ;
: FF. 0 FF.R ;
R->BASE -->

BLOCK #28  ( old TIF #49)
( FLOATING POINT COMPARE ROUTINES 12JUL82 LCT)
BASE->R HEX                               ." ."
: FCLEAN >R DROP DROP DROP R> ;

: F0< 0< FCLEAN ;

: F0= 0= FCLEAN ;

: FCOM SETFL 0A00 C SYSTEM 837C C@ ;
: F> FCOM 40 AND MINUS 0< ;
: F= FCOM 20 AND MINUS 0< ;
: F< FCOM 60 AND 0= ;
: FLERR 8354 C@ ;
: ?FLERR FLERR A ?ERROR ;

R->BASE -->

BLOCK #29  ( old TIF #50)
( FLOATING POINT TRANSCENDENTAL FUNCTIONS 12JUL82 LCT)
BASE->R HEX                                  ." ."
0 VARIABLE LNKSAV
: GLNK 83C4 @ LNKSAV ! GPLLNK LNKSAV @ 83C4 ! ;
: INT >FAC 22 GLNK FAC> ;
: ^   SETFL ARG 836E @ 8 VMBW 24 GLNK FAC> 8 836E +! ;
: SQR >FAC 26 GLNK FAC> ;
: EXP >FAC 28 GLNK FAC> ;
: LOG >FAC 2A GLNK FAC> ;
: COS >FAC 2C GLNK FAC> ;
: SIN >FAC 2E GLNK FAC> ;
: TAN >FAC 30 GLNK FAC> ;
: ATN >FAC 32 GLNK FAC> ;
: PI  >F 3.141592653590 ;

R->BASE
```

```
BLOCK #30  ( old TIF #51)
( CONVERT TO TEXT MODE CONFIGURATION 14SEP82 LAO)
0 CLOAD TEXT BASE->R DECIMAL 35 R->BASE CLOAD SETVDP2
BASE->R  HEX                     CR ." loading text modes"

: TEXT   0 3C0 20 VFILL ( BLANKS TO SCREEN IMAGE AREA )
  28 SCRN_WIDTH ! 0 SCRN_START ! 3C0 SCRN_END ! 460 PABS !
  SETVDP1     1 VDPMDE !   ( NOW SET VDP REGISTERS -->)
  1 6 VWTR   04F 7 VWTR  0F0 SETVDP2 ;

  04B0 VARIABLE TXT8 03E8 , 0106 , 014F , 8800 ,
   0000 , 4F10 , 0000 ,
: TEXT80    TEXT ( temporary)   0 780 20 VFILL
  TXT8 F 0 DO DUP I + C@ I VWTR LOOP DROP
  00 SCRN_START !   50 SCRN_WIDTH !  780 SCRN_END !
    135E PABS !  780 836E !  0 VDPMDE !
  0 0 GOTOXY 0F0 DUP 83D4 C! 1 VWTR   ;     R->BASE
```

```
BLOCK #31  ( old TIF #52)
( CONVERT TO GRAPHICS MODE CONFIG 14SEP82 LAO)
0 CLOAD GRAPHICS  BASE->R DECIMAL 35 R->BASE CLOAD SETVDP2
BASE->R  HEX                       CR ." loading graphics mode"

: GRAPHICS
0 300 20 VFILL ( BLANKS TO SCREEN IMAGE AREA ) 300 80 0 VFILL
380 20 F4 VFILL
20 SCRN_WIDTH ! 0 SCRN_START ! 300 SCRN_END !
SETVDP1   2 VDPMDE !
( NOW SET VDP REGISTERS )
   1 6 VWTR   0F4 7 VWTR
E0 SETVDP2 ;          R->BASE      ;S




```

```
BLOCK #32  ( old TIF #53)
( CONVERT TO MULTI-COLOR MODE CONFIG 14SEP82 LAO)
0 CLOAD MULTI  BASE->R DECIMAL 35 R->BASE CLOAD SETVDP2
BASE->R  HEX                    CR ." loading multicolor mode"

: MULTI      0B0 1 VWTR ( BLANK THE SCREEN )
-1 18 0 DO I 4 / 0FF SWAP DO 1+ I OVER VSBW 8 +LOOP LOOP DROP
800 800 0 VFILL         ( INIT 256 CHAR PATTERNS TO 0 )
 300 80 0 VFILL 380 20 0F4 VFILL
20 SCRN_WIDTH ! 0 SCRN_START ! 300 SCRN_END !
( 460 PABS !  1000 DISK_BUF ! <--SETVDP2 does this!)
3 VDPMDE !
( NOW SET VDP REGISTERS )
   4 6 VWTR   11 7 VWTR
0EB SETVDP2 ;

R->BASE
```

```
BLOCK #33  ( old TIF #54)
  0 ( CONVERT TO GRAPHICS2 MODE CONFIG 14SEP82 LAO)
  1 0 CLOAD GRAPHICS2 BASE->R DECIMAL 35 R->BASE CLOAD SETVDP2
  2                   CR ." loading graphics2 (bitmap) mode"
  3 BASE->R HEX : GRAPHICS2  0A0 1 VWTR
  4 1C62 1CA2 1B80 SETVARS  ( reset user vars, etc.)
  5 -1 1B00 1800 DO 1+ DUP 0FF AND I VSBW LOOP DROP
  6 2 FILES ( # of files = 2)   ( check 8370 for high VRAM????)
  7 0 1800 010 VFILL ( INIT COLOR TABLE )
  8 2000 1800 0 VFILL ( INIT BIT MAP )
  9 20 SCRN_WIDTH ! 1800 SCRN_START ! 1B00 SCRN_END !
 10  2 0 VWTR      6 2 VWTR  ( SET VDP REGISTERS )
 11 07F 3 VWTR     0FF 4 VWTR    36 5 VWTR    7 6 VWTR
 12 0FE 7 VWTR     0E0 DUP 83D4 C! 1 VWTR
 13 1B00 80 0 VFILL ( zero sprite attribute table)
 14 0 0 GOTOXY 4 VDPMDE !  0 837A C!  ;
 15 R->BASE

BLOCK #34  ( old TIF #55)
  0 ( CONVERT TO SPLIT MODE CONFIG 14SEP82 LAO)
  1 0 CLOAD SPLIT  BASE->R DECIMAL 35 R->BASE CLOAD SETVDP2
  2 BASE->R DECIMAL 33 R->BASE CLOAD GRAPHICS2
  3 BASE->R  HEX          CR ." loading split & split2 modes"
  4 : SPLIT GRAPHICS2 1A00 SCRN_START ! 0A0 1 VWTR 3000 800 0FF
  5   VFILL 3100 834A ! 18 GPLLNK   3300 TLC
  6   1A00 100 20 VFILL 1000 800 0F4 VFILL 0 0 GOTOXY 0E0 1 VWTR
  7   5 VDPMDE !   0 837A C!  ;
  8
  9 : SPLIT2 GRAPHICS2 1880 SCRN_END ! 2000 400 0FF VFILL
 10   2100 834A ! 18 GPLLNK    2300 TLC
 11   1800 80 20 VFILL 0 400 0F4 VFILL 0 0 GOTOXY 6 VDPMDE !
 12   0 837A C! ;
 13
 14
 15 R->BASE

BLOCK #35  ( old TIF #56)
  0 ( VDPMODES 14SEP82 LAO ) CR ." loading vdp initializing words"
  1 0 CLOAD SETVDP2  BASE->R  HEX
  2  : SETVDP1 0B0 1 VWTR     ( BLANK THE SCREEN )
  3     800 800 0FF VFILL     ( INIT 256 CHAR PATTERNS TO FF )
  4     900 834A ! 18 GPLLNK  ( LOAD CAPITAL LETTERS )
  5     B00 TLC   ( load true lowercase -ON 99/4A ONLY ) ;
  6 : SETVARS ( vsptr_addr disk_buf_addr pabs_addr --- )
  7    PABS ! DUP DISK_BUF @ = IF DROP ELSE MGT ( old sys loc)
  8    SWAP DISK_BUF ! ( new disk buf) MGT ( new sys loc)  2DE
  9    VMOVE ( move 734B sys area) THEN 836E !  ( VSPTR )  ;
 10 : SETVDP2 ( vreg#1 --- )  ( reset user vars, etc.)
 11     3E0 1000 460 SETVARS
 12     ( SET VDP REGISTERS ) 0 0 VWTR  0 2 VWTR  0E 3 VWTR
 13     1 4 VWTR  6 5 VWTR
 14     3 FILES  ( # of files = 3)
 15     0 0 GOTOXY  0 837A C!  DUP 83D4 C!  1 VWTR ;     R->BASE
```

```
BLOCK #36  ( old TIF #57)
  0 ( GRAPHICS PRIMITIVES 12JUL82 LCT)
  1 0 CLOAD LINE  BASE->R HEX CR ." loading graphics primitives"
  2
  3 380 CONSTANT COLTAB 300 CONSTANT SATR 780 CONSTANT SMTN
  4 800 CONSTANT PDT  800 CONSTANT SPDTAB
  5 : CHAR ( W1 W2 W3 W4 CH ---  )
  6   8 * PDT + >R -2 6 DO PAD I + ! -2 +LOOP PAD R> 8 VMBW ;
  7 : CHARPAT ( CH --- W1 W2 W3 W4 )
  8   8 * PDT + PAD 8 VMBR 8 0 DO PAD I + @ 2 +LOOP ;
  9 : VCHAR ( X Y CNT CH ---  )
 10   >R >R SCRN_WIDTH @ * + SCRN_END @ SCRN_START @ - SWAP
 11   R> R> SWAP 0 DO SWAP OVER OVER SCRN_START @ + VSBW SCRN_WIDTH
 12   @ + ROT OVER OVER /MOD IF 1+ SCRN_WIDTH @ OVER OVER = IF -
 13   ELSE DROP ENDIF ENDIF ROT DROP ROT LOOP DROP DROP DROP ;
 14 R->BASE -->
 15

BLOCK #37  ( old TIF #58)
  0 ( GRAPHICS PRIMITIVES 20OCT83 LAO)   BASE->R HEX   ." ."
  1 : HCHAR ( X Y CNT CH ---  )
  2   >R >R SCRN_WIDTH @ * + SCRN_START @ + R> R> VFILL ;
  3 : COLOR ( FG BG CHSET  ---  ) >R SWAP 10 * + R> COLTAB + VSBW ;
  4 ( : SCREEN { COLOR  --- }  7 VWTR ; ) ( <--now in kernel)
  5 : GCHAR ( X Y --- ASCII ) ( COLUMNS AND ROWS NUMBERED FROM 0 )
  6   SCRN_WIDTH @ * + SCRN_START @ + VSBR ;
  7 : SSDT  ( ADDR --- ) ( SET SPRITE DESCRIPTOR TABLE ADDRESS )
  8   DUP ' SPDTAB ! 800 / 6 VWTR ( RESET VDP REG 6 )
  9   VDPMDE @ 4 < IF SMTN 80 0 VFILL 300 ' SATR ! ENDIF
 10   SATR 20 0 DO DUP >R D000 SP@ R> 2 VMBW DROP 4 + LOOP DROP
 11   ( INIT ALL SPRITES ) ;
 12 : SPCHAR  ( W1 W2 W3 W4 CH#  ---  )
 13   8 * SPDTAB + >R -2 6 DO PAD I + ! -2 +LOOP PAD R> 8 VMBW ;
 14 : SPRCOL ( COL #  ---  )  4 * SATR 3 + + DUP >R VSBR 0F0 AND OR
 15   R> VSBW ;             R->BASE -->

BLOCK #38  ( old TIF #59)
  0 ( GRAPHICS PRIMITIVES 20OCT83 LCT--LES 20DEC2013)   ." ."
  1 BASE->R  HEX    : SPRPAT ( ch # --- )  4 * SATR 2+ + VSBW ;
  2 : SPRPUT ( dx dy # --- ) 4 * SATR + >R 1- 100 U* DROP + SP@
  3   R> 2 VMBW DROP ;   : SPRITE ( dx dy col ch # --- )
  4   DUP 4 * SATR + >R DUP >R SPRPAT R SPRCOL R> SPRPUT R> 4 +
  5   SATR DO I VSBR D0 = IF C001 SP@ I 2 VMBW DROP ENDIF 4 +LOOP ;
  6 : MOTION ( spx spy # --- )
  7   4 * SMTN + >R 8 SLA SWAP 00FE AND OR SP@ R> 2 VMBW DROP ;
  8 : #MOTION ( NO ---  )   837A C! ;  : SPRGET ( # --- DX DY )
  9   4 * SATR + DUP VSBR 1+ 0FF AND SWAP 1+ VSBR SWAP ;
 10 : DXY ( x1 y1 x2 y2 --- x^2 y^2 )  ROT - ABS ROT ROT - ABS DUP
 11   * SWAP DUP * ;   : BEEP 34 GPLLNK ;   : HONK 36 GPLLNK ;
 12 26 USER JMODE ( 0=TI Forth; ~0=CRU)
 13 : SPRDIST ( #1 #2 --- dist^2 ) SPRGET ROT SPRGET DXY OVER OVER
 14   + DUP >R OR OR 8000 AND IF R> DROP 7FFF ELSE R> ENDIF ;
 15                                            R->BASE -->
```

```
BLOCK #39  ( old TIF #60)
( GRAPHICS PRIMITIVES 12JUL82 LCT--LES 22DEC2013) BASE->R HEX
." ."   : SPRDISTXY ( x y # --- dist^2 ) SPRGET DXY OVER OVER
  + DUP >R OR OR 8000 AND IF R> DROP 7FFF ELSE R> ENDIF ;
: MAGNIFY ( mag --- ) 83D4 C@ 0FC AND + DUP 83D4 C! 1 VWTR ;
: JKBD ( kbd --- chr xstat ystat )  8374 C!
  ?KEY DROP 8375 C@ DUP 12 = OVER 0FF = OR
  IF 8377 C@ 8376 C@ ELSE DUP
  CASE 4 OF 0FC 4  ENDOF  5 OF 0 4 ENDOF  6 OF 4   4 ENDOF
       2 OF 0FC 0  ENDOF  3 OF 4 0 ENDOF  0 OF 0 0FC ENDOF
      0F OF 0FC 0FC ENDOF 0E OF 4 0FC ENDOF DROP DROP 0 0 0 0
  ENDCASE THEN  0 8374 C! ;
CODE JCRU  ( joyst# --- n ) C059 , 0221 , 0005 , 06C1 , 020C ,
  0024 , 30C1 , 020C , 0006 , 3541 , 06C1 , 0541 , 0241 , 001F ,
  C641 , 04E0 , 83D6 , NEXT,
: JOYST ( kbd|joyst --- [chr xst yst]|n )
  JMODE @ IF JCRU ELSE JKBD THEN ;              R->BASE -->

BLOCK #40  ( old TIF #61)
( GRAPHICS PRIMITIVES 12JUL82 LCT)  BASE->R HEX    ." ."
: COINC ( #1 #2 tol  --- f ) ( 0= no coinc  1= coinc )
  DUP * DUP + >R SPRDIST R> > 0= ;
: COINCXY ( DX DY # TOL --- F )
  DUP * DUP + >R SPRDISTXY R>  > 0= ;
: COINCALL  ( --- F ) ( BIT SET IF ANY TWO SPRITES OVERLAP )
  837B C@ 20 AND 20 = ;  ( <--may work better than 8802)
: DELSPR ( # --- )
  4 * DUP SATR + >R 0 C001 SP@ R> 4 VMBW DROP DROP
  SMTN + >R 0 0 SP@ R> 4 VMBW DROP DROP ;
: DELALL ( --- )
  0 #MOTION SATR 20 0 DO DUP D0 SWAP VSBW 4 + LOOP DROP
  SMTN 80 0 VFILL ;


  R->BASE -->

BLOCK #41  ( old TIF #62)
( GRAPHICS PRIMITIVES 24NOV82 LAO) BASE->R  HEX  0 VARIABLE ADR
: MINIT 18 0 DO 0 I 4 / 20 * DUP 20 + SWAP
             DO DUP J 1 I HCHAR 1+ LOOP  DROP LOOP ;    ." ."
: MCHAR ( COLOR C R --- ) DUP >R 2 / SWAP DUP >R 2 / SWAP
  DUP >R GCHAR  DUP 20 /  100 U* DROP 800 + >R 20 MOD
  8 * R> + R>  4 MOD 2 * + ADR ! R> 2 MOD R> 2 MOD SWAP
  IF  IF 3 ELSE 1 ENDIF  ELSE  IF 2 ELSE 0 ENDIF   ENDIF
  DUP 2 MOD 0= IF SWAP 10 * SWAP ENDIF
  CASE 0 OF ADR @ VSBR 0F ENDOF  1 OF ADR @ VSBR F0 ENDOF
  2 OF 1 ADR +! ADR @ VSBR 0F ENDOF
  3 OF 1 ADR +! ADR @ VSBR F0 ENDOF
  ENDCASE AND +   ADR @  VSBW ;
0 VARIABLE DMODE   -1 VARIABLE DCOLOR
: DRAW 0 DMODE ! ;  : UNDRAW 1 DMODE ! ;   : DTOG 2 DMODE ! ;
8040 VARIABLE DTAB 2010 , 804 , 201 , 7FBF , DFEF , F7FB ,
FDFE , 8040 , 2010 , 804 , 201 ,  R->BASE -->
```

```
BLOCK #42  ( old TIF #63)
( GRAPHICS PRIMITIVES ) BASE->R  HEX               ." ."
CODE DDOT   C079 ,
     C0D9 , C081 , C103 , 0241 ,
     0007 , 0243 , 0007 , 0242 ,
     00F8 , 0244 , 00F8 , 0A52 ,
     A042 , A044 , 0221 , 2000 ,
     04C4 , D123 , DTAB , 06C4 ,
     C644 , 0649 , C641 , NEXT,
: DOT ( X Y --- )
  DDOT DUP 2000 - >R DMODE @
  CASE  0 OF VOR  ENDOF  ( DRAW )
        1 OF SWAP FF XOR SWAP VAND ENDOF  ( UNDRAW )
        2 OF VXOR ENDOF  ( TOGGLE )
  DROP DROP ENDCASE R>
  DCOLOR @ 0 < IF DROP ELSE DCOLOR @ SWAP VSBW ENDIF ;
 R->BASE  -->

BLOCK #43  ( old TIF #64)
( GRAPHICS PRIMITIVES 12JUL82 LCT) BASE->R HEX       ." ."
: SNW DUP SGN + ;
: LINE >R R ROT >R R - SNW SWAP >R R ROT >R R - SNW OVER ABS
  OVER ABS < >R R 0= IF SWAP ENDIF 100 ROT ROT */ R>
  IF ( X AXIS ) R> R> OVER OVER >
     IF ( MAKE L TO R ) SWAP R> DROP R>
     ELSE R> R> DROP
     ENDIF 100 * ROT ROT 1+ SWAP
     DO I OVER 0 100 M/ SWAP DROP DOT OVER + LOOP
  ELSE ( Y AXIS ) R> R> R> R> ROT >R ROT >R OVER OVER >
     IF ( MAKE T TO B ) SWAP R> DROP R>
     ELSE R> R> DROP
     ENDIF 100 * ROT ROT 1+ SWAP
     DO DUP 0 100 M/ SWAP DROP I DOT OVER + LOOP
  ENDIF DROP DROP ;
R->BASE

BLOCK #44  ( old TIF #65)
( COMPACT LIST )
0 CLOAD CLIST  BASE->R   CR ." loading compact list words"
DECIMAL  0 VARIABLE TCHAR 382 ALLOT
46 BLOCK 192 + TCHAR 384 CMOVE  HEX
TCHAR 7C - CONSTANT TC  0 VARIABLE BADDR  0 VARIABLE INDX
( SMASH EXPECTS ADDR #CHAR LINE# --- LB VADDR CNT )
0 VARIABLE LB FE ALLOT
CODE SMASH
  C079 , C0B9 , C0D9 , 0204 , LB   , C644 , 0649 , 06C1 ,
  0221 , 2000 , C641 , C042 , 0581 , 0241 , FFFE , 0649 ,
  0A21 , C641 , A083 , 80C2 , 1501 , 1020 , 04C5 , 04C6 ,
  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 , NEXT,
R->BASE -->
```

```
BLOCK #45  ( old TIF #66)
  0 ( COMPACT LIST ) BASE->R DECIMAL          ." ."
  1 : CLINE LB 100 ERASE  SMASH   VMBW    ;
  2 : CLOOP  DO I 64 * OVER + 64 I CLINE LOOP DROP ;
  3
  4 : CLIST BLOCK 16 0 CLOOP ;     R->BASE      ;S
  5
  6
  7
  8
  9
 10
 11
 12
 13
 14
 15

BLOCK #46  ( old TIF #67)
  0 ( Tiny character patterns for TCHAR array---compact list for
  1   64-column editor---388 bytes, lines 3:0-9:0 below )
  2
  3
  4
  5
  6
  7
  8
  9
 10
 11
 12
 13
 14
 15

BLOCK #47  ( old TIF #68)
  0 ( FILE I/O ROUTINES 12JUL82 LCT)
  1 0 CLOAD STAT   BASE->R HEX     CR ." loading file I/O library"
  2
  3 0 VARIABLE PAB-ADDR
  4 0 VARIABLE PAB-BUF
  5 0 VARIABLE PAB-VBUF
  6 : FILE <BUILDS , , , DOES> DUP @ PAB-VBUF ! 2+ DUP @ PAB-BUF !
  7     2+ @ PAB-ADDR ! ;
  8 : GET-FLAG PAB-ADDR @ 1+ VSBR ;
  9 : PUT-FLAG PAB-ADDR @ 1+ VSBW ;
 10 : SET-PAB PAB-ADDR @ DUP 0A 0 VFILL 2+ PAB-VBUF SWAP 2 VMBW ;
 11 : CLR-STAT GET-FLAG 1F AND PUT-FLAG ;
 12 : CHK-STAT GET-FLAG 0E0 AND
 13      837C C@ 20 AND OR 9 ?ERROR ;
 14 : FXD GET-FLAG 0EF AND PUT-FLAG ;
 15 : VRBL GET-FLAG 10 OR PUT-FLAG ;   R->BASE -->
```

```
BLOCK #48  ( old TIF #69)
( FILE I/O ROUTINES 12JUL82 LCT) BASE->R HEX     ." ."
: DSPLY GET-FLAG 0F7 AND PUT-FLAG ;
: INTRNL GET-FLAG 8 OR PUT-FLAG ;
: I/OMD GET-FLAG 0F9 AND ;
: INPT I/OMD 4 OR PUT-FLAG ;
: OUTPT I/OMD 2 OR PUT-FLAG ;
: UPDT I/OMD PUT-FLAG ;
: APPND I/OMD 6 OR PUT-FLAG ;
: SQNTL GET-FLAG 0FE AND PUT-FLAG ;
: RLTV GET-FLAG 1 OR PUT-FLAG ;
: REC-LEN PAB-ADDR @ 4 + VSBW ;
: CHAR-CNT! PAB-ADDR @ 5 + VSBW ;
: CHAR-CNT@ PAB-ADDR @ 5 + VSBR ;
: REC-NO DUP SWPB PAB-ADDR @ 6 + VSBW PAB-ADDR @ 7 + VSBW ;
: N-LEN! PAB-ADDR @ 9 + VSBW ;
R->BASE  -->

BLOCK #49  ( old TIF #70)
( FILE I/O ROUTINES 12JUL82 LCT) BASE->R HEX      ." ."
( COMPILE A STRING WHICH IS MOVED TO VDP-ADDR AT EXECUTION)

: (F-D")
    PAB-ADDR @ 0A + R COUNT DUP 1+ =CELLS R> +
    >R >R SWAP R VMBW R> N-LEN! ;
: F-D" 22 STATE @
    IF
      COMPILE (F-D") WORD HERE C@
      1+ =CELLS ALLOT
    ELSE
      PAB-ADDR @ 0A + SWAP WORD HERE COUNT >R SWAP R
      VMBW R> N-LEN!
    ENDIF ; IMMEDIATE

R->BASE  -->

BLOCK #50  ( old TIF #71)
( FILE I/O ROUTINES 12JUL82 LCT)
BASE->R HEX                              ." ."
: DOI/O CLR-STAT PAB-ADDR @ VSBW PAB-ADDR @ 9 + 8356 !
  0 837C C! DSRLNK CHK-STAT ;
: OPN 0 DOI/O ;
: CLSE 1 DOI/O ;
: RD 2 DOI/O PAB-VBUF @ PAB-BUF @ CHAR-CNT@ VMBR CHAR-CNT@ ;
: WRT >R PAB-BUF @ PAB-VBUF @ R VMBW R> CHAR-CNT! 3 DOI/O ;
: RSTR REC-NO 4 DOI/O ;
: LD REC-NO 5 DOI/O ;
: SV REC-NO 6 DOI/O ;
: DLT 7 DOI/O ;

: STAT 9 DOI/O PAB-ADDR @ 8 + VSBR ;

R->BASE
```

```
BLOCK #51  ( old TIF #72)
  0 ( ALTERNATE I/O SUPPORT FOR RS232 PNTR 12JUL82 LCT...mod LES)
  1 0 CLOAD INDEX     BASE->R DECIMAL 47 R->BASE CLOAD STAT
  2 0 0 0 FILE >RS232  BASE->R HEX CR ." loading printing routines"
  3 : SWCH >RS232 PABS @ 10 + DUP PAB-ADDR ! 1- PAB-VBUF !
  4   SET-PAB OUTPT F-D" RS232.BA=9600"              OPN 3
  5   PAB-ADDR @ VSBW 1 PAB-ADDR @ 5 + VSBW  PAB-ADDR @ ALTOUT ! ;
  6 : UNSWCH 0 ALTOUT ! CLSE ;
  7 : ?ASCII ( BLOCK# --- FLAG )
  8        BLOCK 0 SWAP DUP 400 + SWAP
  9        DO I C@ 20 > + I C@ DUP 20 < SWAP 7F > OR
 10           IF DROP 0 LEAVE ENDIF LOOP  ;
 11 : TRIAD 0 SWAP SWCH 3 / 3 * 1+ DUP 3 + SWAP
 12   DO I ?ASCII IF 1+ I LIST CR ENDIF LOOP
 13   -DUP IF 3 SWAP - 14 * 0 DO CR LOOP
 14   ." fbForth --- a TI-Forth/fig-Forth extension" 0C EMIT
 15   ENDIF UNSWCH  ;                           R->BASE   -->

BLOCK #52  ( old TIF #73)
  0 ( SMART TRIADS AND INDEX 15SEP82 LAO ) BASE->R DECIMAL   ." ."
  1 : TRIADS ( FROM TO --- )
  2   3 / 3 * 2+ SWAP 3 / 3 * 1+ DO I TRIAD 3 +LOOP ;
  3 : INDEX  ( FROM TO --- )  1+ SWAP
  4   DO I DUP ?ASCII IF CR 4 .R 2 SPACES I BLOCK 64 TYPE ELSE DROP
  5      ENDIF PAUSE IF LEAVE ENDIF LOOP  ;    R->BASE    ;S
  6
  7
  8
  9
 10
 11
 12
 13
 14
 15

BLOCK #53  ( old TIF #75)
  0 ( ASSEMBLER 12JUL82 LCT-LES12DEC2013) 0 CLOAD A$$M BASE->R HEX
  1 ASSEMBLER DEFINITIONS CR ." loading TMS9900 assembler" CR ."  "
  2 : GOP' OVER DUP 1F > SWAP 30 < AND IF + , , ELSE + , ENDIF ;
  3 : GOP <BUILDS , DOES> @ GOP' ;
  4 0440 GOP B,     0680 GOP BL,     0400 GOP BLWP,
  5 04C0 GOP CLR,   0700 GOP SETO,   0540 GOP INV,
  6 0500 GOP NEG,   0740 GOP ABS,    06C0 GOP SWPB,
  7 0580 GOP INC,   05C0 GOP INCT,   0600 GOP DEC,
  8 0640 GOP DECT,  0480 GOP X,
  9 : 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 GGOP C,   9000 GGOP CB,
 15 6000 GGOP S,  7000 GGOP SB,  E000 GGOP SOC, F000 GGOP SOCB, -->
```

```
BLOCK #54  ( old TIF #76)
  0 ( ASSEMBLER 12JUL82 LCT)                        ." ."
  1 4000 GGOP SZC,  5000 GGOP SZCB,  C000 GGOP MOV,  D000 GGOP MOVB,
  2  : 0OP <BUILDS , DOES> @ , ;
  3 0340 0OP IDLE,  0360 0OP RSET, 03C0 0OP CKOF,
  4 03A0 0OP CKON,  03E0 0OP LREX, 0380 0OP RTWP,
  5 : ROP <BUILDS , DOES> @ + , ;  02C0 ROP STST,  02A0 ROP STWP,
  6 : IOP <BUILDS , DOES> @ , , ;  02E0 IOP LWPI,  0300 IOP LIMI,
  7 : RIOP <BUILDS , DOES> @ ROT + , , ;   0220 RIOP AI,
  8 0240 RIOP ANDI,  0280 RIOP CI,  0200 RIOP LI,  0260 RIOP ORI,
  9 : RCOP <BUILDS , DOES> @ SWAP 10 * + + , ;
 10 0A00 RCOP SLA,  0800 RCOP SRA,  0B00 RCOP SRC,  0900 RCOP SRL,
 11 : DOP <BUILDS , DOES> @ SWAP 00FF AND OR , ;
 12 1300 DOP JEQ,  1500 DOP JGT,  1B00 DOP JH,  1400 DOP JHE,
 13 1A00 DOP JL,  1200 DOP JLE,  1100 DOP JLT,  1000 DOP JMP,
 14 1700 DOP JNC,  1600 DOP JNE,  1900 DOP JNO,  1800 DOP JOC,
 15 1C00 DOP JOP,  1D00 DOP SBO,  1E00 DOP SBZ,  1F00 DOP TB, -->

BLOCK #55  ( old TIF #77)
  0 ( ASSEMBLER 12JUL82 LCT)                  ." ." CR ."  "
  1 : GCOP <BUILDS , DOES> @ SWAP 000F AND 040 * + GOP' ;
  2 3000 GCOP LDCR,  3400 GCOP STCR,
  3 00 CONSTANT R0  01 CONSTANT R1  02 CONSTANT R2  03 CONSTANT R3
  4 04 CONSTANT R4  05 CONSTANT R5  06 CONSTANT R6  07 CONSTANT R7
  5 08 CONSTANT R8  09 CONSTANT R9  0A CONSTANT R10  0B CONSTANT R11
  6  0C CONSTANT R12  0D CONSTANT R13  0E CONSTANT R14
  7 0F CONSTANT R15  08 CONSTANT UP  09 CONSTANT SP  0A CONSTANT W
  8 0D CONSTANT IP  0E CONSTANT RP  0F CONSTANT NEXT
  9 : @() 020 ;  : *? 010 + ;  : *?+ 030 + ;  : @(?) 020 + ;
 10 : @(R0) R0 @(?) ;  : *R0 R0 *? ;  : *R0+ R0 *?+ ;
 11 : @(R1) R1 @(?) ;  : *R1 R1 *? ;  : *R1+ R1 *?+ ;
 12 : @(R2) R2 @(?) ;  : *R2 R2 *? ;  : *R2+ R2 *?+ ;
 13 : @(R3) R3 @(?) ;  : *R3 R3 *? ;  : *R3+ R3 *?+ ;
 14 : @(R4) R4 @(?) ;  : *R4 R4 *? ;  : *R4+ R4 *?+ ;
 15 : @(R5) R5 @(?) ;  : *R5 R5 *? ;  : *R5+ R5 *?+ ;      -->

BLOCK #56  ( 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 #57  ( old TIF #79)
( ASSEMBLER 12JUL82 LCT)                           ." ."
: @(RP) RP @(?) ;  : *RP RP *? ;  : *RP+ RP *?+ ;
: *NEXT+ NEXT *?+ ;  : *NEXT NEXT *? ;  : @(NEXT) NEXT @(?) ;
: @@ @() ; : ** *? ; : *+ *?+ ; : () @(?) ;  ( Wycove syntax)

( DEFINE JUMP TOKENS )
: GTE 1 ;  : H 2 ;  : NE 3 ;  : L 4 ;  : LTE 5 ;  : EQ 6 ;
: OC 7 ;  : NC  8 ;  : OO 9 ;  : HE 0A ;  : LE 0B ;  : NP 0C ;
: LT 0D ;  : GT 0E ;  : NO 0F ;  : OP 10 ;
: CJMP    ?EXEC
     CASE LT OF 1101 , 0 ENDOF    GT OF 1501 , 0 ENDOF
          NO OF 1901 , 0 ENDOF    OP OF 1C01 , 0 ENDOF
          DUP 0< OVER 10 > OR IF 19 ERROR ENDIF DUP
     ENDCASE 100 * 1000 + , ;
: IF,     ?EXEC [COMPILE] CJMP HERE 2- 42 ; IMMEDIATE
-->

BLOCK #58  ( old TIF #80)
( ASSEMBLER 12JUL82 LCT)                         ." ."
: 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 )
: RT, R11 ** B, ;  ( RT pseudo-instruction )
: THEN, [COMPILE] ENDIF, ; IMMEDIATE ( ENDIF, synonym )
 FORTH DEFINITIONS  : A$$M ;   R->BASE

BLOCK #59  ( old TIF #83)
( BSAVE -- BINARY SAVER FOR FORTH OVERLAYS     LCT 14SEP82 )
0 CLOAD BSAVE    BASE->R DECIMAL  CR ." loading BSAVE utility"
: BSAVE ( addr  strt_block --- nxt_block)  EMPTY-BUFFERS
    BEGIN
      SWAP >R DUP 1+ SWAP
      BUFFER UPDATE DUP B/BUF ERASE
      R OVER ! 2+ HERE OVER ! 2+
      CURRENT @ OVER ! 2+           LATEST      OVER ! 2+
      CONTEXT @ OVER ! 2+           CONTEXT @ @ OVER ! 2+
      VOC-LINK @ OVER ! 2 +   29801 OVER ! 10 +
      HERE R -
      R> DUP 1000 + >R SWAP >R SWAP R>
      1000 MIN CMOVE
      R SWAP HERE R> <
    UNTIL
    SWAP DROP FLUSH   ;  R->BASE
```

Appendix K Diskette Format Details

The information in this section is based on TI’s *Software Specifications for the 99/4 Disk Peripheral (March 28, 1983)*.

The original disk drives supplied by TI supported only single-sided, single-density (SSSD), 90-KB diskettes. The original TI Forth system was designed around and supplied in this disk format. Though the TI Forth system could not readily be moved to a disk of another size, **fbForth** consists of only two files, which can easily be moved a disk of any size. Different disk formats are possible; however, we will consider the usual format of 256 bytes per sector and 40 tracks per side. The following table shows possible formats with 256 bytes/sector and 40 tracks/side:

Disk Type	Sides	Density	Sectors/ Track	Total Sectors	Capacity
SSSD	1	single	9	360	90 KB
DSSD	2	single	9	720	180 KB
SSDD	1	double	18	720	180 KB
DSDD	2	double	18	1440	360 KB
Compact Flash[16]	2	double	20	1600[17]	400 KB

The information in the following sections accrues to all the above formats:

K.1 Volume Information Block (VIB)

Byte #	1st Byte	2nd Byte	Byte #
0 8	Disk Volume Name (10 characters padded on the right with blanks)		1 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 54	Reserved		21 55
56 254	Allocation Bitmap (room for 1600 sectors)		57 255

[16] This is a third-party peripheral expansion device with 400 KB virtual disks using Compact Flash memory on devices named nanoPEB and CF7+ (see website: *http://webpages.charter.net/nanopeb/*)

[17] 1600 sectors is the maximum possible number of sectors that can be managed by the current specification.

Sector 0 contains the volume information block (VIB). The layout is shown in the above table.

K.2 File Descriptor Index Record (FDIR)

Sector 1 contains the file descriptor index record (FDIR). It can hold up to 127 2-byte entries, each pointing to a file descriptor record (FDR—see next section). These pointers are alphabetically sorted by the file names to which they point. This list of pointers starts at the beginning of sector 1 and ends with a pointer value of 0.

K.3 File Descriptor Record (FDR)

Byte #	1st Byte	2nd Byte	Byte #
0 8	File Name (10 characters padded on the right with blanks)		1 9
10	Reserved		11
12	File Status Flags	# of Records/Sector (0 for program)	13
14	# of Sectors currently allocated (not counting this FDR)		15
16	EOF Offset (bytes in last Sector)	Bytes/Record	17
18	# of Records (Fixed) or # of Sectors (Variable)—bytes are in reverse order		19
20 26	Reserved		21 27
28 254	Data Chain Pointer Blocks (3 bytes/block encoding two 12-bit numbers that indicate cluster start and highest, cumulative sector offset)		29 255

There can be as many as 127 file descriptor records (FDRs) laid out as in the above table. There are no subdirectories. FDRs will start in sector 2 and continue, at least, until sector 33, unless a file allocation requires more space than is available in sectors 34 – end-of-disk, in which case the system will begin allocating space for the file in the first available sector in sectors 3 – 33. This is done "to obtain faster directory search response times"[18]. Each FDR beyond 32 files will be placed in the first available sector.

Byte 12 contains file status flags defined as follows, with bit 0 as the least significant bit:

Bit #	Description
0	Program or Data file (0 = Data; 1 = Program)
1	Binary or ASCII data (0 = ASCII, DISPLAY file; 1 = Binary, INTERNAL or program file)
2	Reserved
3	PROTECT flag (0 = not protected; 1 = protected)
4–6	Reserved
7	FIXED/VARIABLE flag (0 = fixed-length records; 1 = variable-length records)

[18] *Software Specifications for the 99/4 Disk Peripheral (March 28, 1983)*, p. 19.

The cluster blocks listed in bytes 28 – 255 of the FDR each contain 2 12-bit (3-nybble[19]) numbers. The first points to the beginning sector of that cluster of contiguous sectors and the second is the sector offset reached by that cluster. If we label the 3 nybbles of the cluster pointer as $n_1 - n_3$ and the 3 nybbles of the cumulative sector offset as $m_1 - m_3$, with the subscripts indicating the significance of the nybble, then the 3 bytes are laid out as follows:

Byte 1: n_2n_1 Byte 2: m_1n_3 Byte 3: m_3m_2

The actual 12-bit numbers, then, are

Cluster Pointer: $n_3n_2n_1$ Sector Offset: $m_3m_2m_1$

For example, the following represents 2 blocks in the FDR for a file with 2 clusters allocated:

Actual layout in the FDR: **4D20h 5F05h F060h**

1st Cluster Pointer: **04Dh** (77_{10})[20] Record Offset: **5F2h** (1522_{10})

2nd Cluster Pointer: **005h** (5_{10}) Record Offset: **60Fh** (1551_{10})

The above example represents a file, the data for which occupies 1552 sectors on the disk. If we assume that no files have been deleted in this case, you should also be able to deduce that there are only 3 files on the disk because the second cluster starts in sector 5 and occupies all sectors from 5 – 33, which should tell you there are 3 FDRs before this cluster was allocated: Sector 0 (VIB), sector 1 (FDIR), sector 2 (FDR of first file), sector 3 (FDR of second file), sector 4 (FDR of third file and sector 5 (second cluster start of the third file, the first two occupying sectors 34 – 76 by inference). Furthermore, the disk contains 1600 sectors because that is the maximum and the first cluster ended in the 1600th sector of the disk (1st cluster starts in sector 77 and ends 1522 sectors later in sector 1599).[21]

[19] 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.

[20] The subscript, 10, indicates base 10 (decimal).

[21] This example is taken from one of my (Lee Stewart’s) Compact Flash volumes.

Appendix L Notes on Radix-100 Notation

fbForth floating-point math routines use radix-100 format for floating-point numbers. The term “radix” is used in mathematics to mean “number base”. We will use “radix 100” to describe the base-100 or centimal number system and “radix 10” to describe the base-10 or decimal number system. Radix-100 format is the same format used by the XML and GPL routines in the TI-99/4A console. Each floating-point number is stored in 8 bytes (4 cells) with a sign bit, a 7-bit, excess-64 (64-biased) integer exponent of the radix (100) and a normalized, 7-digit (1 radix-100 digit/byte) significand for a total of 8 bytes per floating point number. The signed, radix-100 exponent can be -64 to +63. (Keep in mind that the exponent is for radix-100 notation. Those same exponents radix 10 would be -128 to +126.) The exponent is stored in the most significant byte (MSB) biased by 64, *i.e.*, 64 is added to the actual exponent prior to storing, *i.e.*, -64 to +63 is stored as 0 to 127.

The significand (significant digits of the number) must be normalized, *i.e.*, if the number being represented is not zero, the MSB of the significand must always contain the first non-zero (significant) radix-100 digit, with the radix exponent of such a value that the radix point immediately follows the first digit. This is essentially scientific notation for radix 100. Each byte contains one radix-100 digit of the number, which, of course, means that each byte can have a value from 0 to 99 (`0` to `63h`) except for the first byte of a non-zero number, which must be 1 to 99. It is easy to view a radix-100 number as a radix-10 number by representing the radix-100 digits as pairs of radix-10 digits because radix 100 is the square of radix 10. In the following list of largest and smallest possible 8-byte floating point numbers, the radix-100 representation is on the left with spaces between pairs of radix-100 digits. The radix-16 (hexadecimal) internal representation of each byte of the number is also shown:

- Largest positive floating point number [hexadecimal: `7F 63 63 63 63 63 63 63`]:

  $$99\ .\ 99\ 99\ 99\ 99\ 99\ 99 \times 100^{63} = 99.999999999999 \times 10^{126}$$
  $$= 9.9999999999999 \times 10^{127}$$

- Largest negative floating point number [hexadecimal: `80 9D 63 63 63 63 63 63`]:

  $$-99\ .\ 99\ 99\ 99\ 99\ 99\ 99 \times 100^{63} = -99.999999999999 \times 10^{126}$$
  $$= -9.9999999999999 \times 10^{127}$$

- Smallest positive floating point number [hexadecimal: `00 01 00 00 00 00 00 00`]:

  $$01\ .\ 00\ 00\ 00\ 00\ 00\ 00 \times 100^{-64} = 1.000000000000 \times 10^{-128}$$

- Smallest negative floating point number [hexadecimal: `FF FF 00 00 00 00 00 00`]:

  $$-01\ .\ 00\ 00\ 00\ 00\ 00\ 00 \times 100^{-64} = -1.000000000000 \times 10^{-128}$$

The only difference in the internal storage of positive and negative floating point numbers is that only the first word (2 bytes) of negative numbers is negated or complemented (two’s complement).

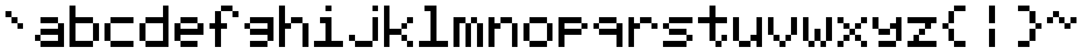
A floating point zero is represented by zeroing only the first word. The remainder of the floating point number does not need to be zeroed for the number to be treated as zero for all floating point calculations.

Appendix M Changing the True Lowercase Character Sets

This appendix explains how to change the true lowercase character sets for the text, text80 graphics modes as well as the 64-column editor of **fbForth**.

M.1 True Lowercase for Text, Text80 and Graphics Modes

The following graphic shows the true lowercase character set the author designed for text, text80 and graphics modes:



To change it to a character set of your own design will require copying the requisite 31 8-byte character patterns to the VRAM location reserved by **fbForth** for this purpose. Unless you want to rewrite the Assembly language code for **fbForth**, you will need to copy the patterns after **fbForth** has booted. You can set up FBLOCKS to do this automatically every time you boot up **fbForth** or do it manually. What follows details how to do it automatically. We will make use of the unused blocks at the end of FBLOCKS and modify block 1 to load our new character set.

Each character pattern requires 8 bytes. You can use the following blocks as a guide to designing your own patterns. Remember that only the 6 leftmost bits of each byte are used in text and text80 modes and that graphics mode uses all 8 bits:

1. Block 60 – block 62, line 2 define a temporary array, **`TRUE_LC`** .
2. Block 62, line 4 copies the **`TRUE_LC`** array to block 63 and then **`FORGET`**s **`TRUE_LC`** to recover the memory used by the array. The **`;S`** on line 5 will stop interpretation of the block.
3. Copy block 62, line 7 to block 1, line 4 so that it will load your new lowercase patterns to the VRAM address retrieved by **`LCT`** that is expected by **fbForth** and, subsequently, copy them (**`2816 TLC`**) to their proper place in the pattern descriptor table (2816 or **`0B00h`**) for text, text80 and graphics modes. *Note:* If you decide to put this line on or after block 1, line 11, be sure to move the **`;S`** after it or remove it altogether.

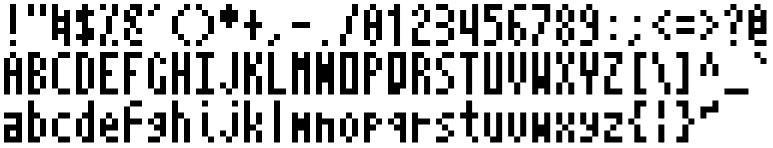
```
BLOCK #60
  0 ( True lowercase characters for TEXT mode)
  1 BASE->R HEX   0000 VARIABLE TRUE_LC
  2        2010 , 0800 , 0000 , ( `)
  3 0000 , 0038 , 043C , 443C , ( a)
  4 0040 , 4078 , 4444 , 4478 , ( b)
  5 0000 , 003C , 4040 , 403C , ( c)
  6 0004 , 043C , 4444 , 443C , ( d)
  7 0000 , 0038 , 447C , 4038 , ( e)
  8 0018 , 2420 , 7020 , 2020 , ( f)
  9 0000 , 003C , 443C , 0438 , ( g)
 10 0040 , 4058 , 6444 , 4444 , ( h)
 11 0010 , 0030 , 1010 , 107C , ( i)
 12 0004 , 0004 , 0404 , 4438 , ( j)
 13 0040 , 4044 , 4870 , 4844 , ( k)
 14 0030 , 1010 , 1010 , 107C , ( l)
 15 0000 , 0068 , 5454 , 5454 , ( m)   -->

BLOCK #61
  0 ( True lowercase characters for TEXT mode continued)
  1 0000 , 0058 , 6444 , 4444 , ( n)
  2 0000 , 0038 , 4444 , 4438 , ( o)
  3 0000 , 0078 , 4478 , 4040 , ( p)
  4 0000 , 0038 , 443C , 0404 , ( q)
  5 0000 , 0058 , 6440 , 4040 , ( r)
  6 0000 , 003C , 4038 , 0478 , ( s)
  7 0010 , 107C , 1010 , 1408 , ( t)
  8 0000 , 0044 , 4444 , 4C34 , ( u)
  9 0000 , 0044 , 4444 , 2810 , ( v)
 10 0000 , 0044 , 4454 , 5428 , ( w)
 11 0000 , 0044 , 2810 , 2844 , ( x)
 12 0000 , 0044 , 4C34 , 0438 , ( y)
 13 0000 , 007C , 0810 , 207C , ( z)
 14 0018 , 2020 , 4020 , 2018 , ( {)
 15 0010 , 1010 , 0010 , 1010 , ( |)   -->

BLOCK #62
  0 ( True lowercase characters for TEXT mode concluded)
  1 0030 , 0808 , 0408 , 0830 , ( })
  2 0000 , 2054 , 0800 , 0000 , ( ~)   R->BASE
  3
  4 BASE->R DECIMAL TRUE_LC 63 BLOCK 124 MOVE FLUSH FORGET TRUE_LC
  5 R->BASE   ;S
  6
  7 BASE->R DECIMAL 63 BLOCK LCT 248 VMBW 2816 TLC R->BASE
  8
  9
 10
 11
 12
 13
 14
 15
```

M.2 True Lowercase for Bitmap mode

The following graphic shows the complete character set, with the true lowercase letters and the '@' designed by the author, for bitmap mode:


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

This character set is used principally by the 64-column editor via the word **SMASH** defined in block 44 of FBLOCKS. Designing the characters for a 3×7 matrix was quite a challenge. The '&' should probably be re-designed.

The only change necessary here is to overwrite the character codes for the tiny character set in block 45, lines 3 – 9 of FBLOCKS. Loading the following blocks from a blocks file of your design and contiguous block numbers will accomplish this:

```
BLOCK #10
    ( DEFINITIONS FOR true lowercase TINY CHARACTERS) BASE->R HEX
    0EEE  VARIABLE TCHAR EEEE ,
    0000 , 0000 , (  )  0444 , 4404 , ( !)  0AA0 , 0000 , ( ")
    08AE , AEA2 , ( #)  04EC , 46E4 , ( $)  0A24 , 448A , ( %)
    06AC , 4A86 , ( &)  0480 , 0000 , ( ')  0248 , 8842 , ( ()
    0842 , 2248 , ( ^0) 04EE , 4000 , ( *)  0044 , E440 , ( +)
    0000 , 0048 , ( ,)  0000 , E000 , ( -)  0000 , 0004 , ( .)
    0224 , 4488 , ( /)  04AA , AAA4 , ( 0)  04C4 , 4444 , ( 1)
    04A2 , 488E , ( 2)  0C22 , C22C , ( 3)  02AA , AE22 , ( 4)
    0E8C , 222C , ( 5)  0688 , CAA4 , ( 6)  0E22 , 4488 , ( 7)
    04AA , 4AA4 , ( 8)  04AA , 622C , ( 9)  0004 , 0040 , ( :)
    0004 , 0048 , ( ;)  0024 , 8420 , ( <)  000E , 0E00 , ( =)
    0084 , 2480 , ( >)  04A2 , 4404 , ( ?)  04AE , AE86 , ( @)
    04AA , EAAA , ( A)  0CAA , CAAC , ( B)  0688 , 8886 , ( C)
    0CAA , AAAC , ( D)  0E88 , C88E , ( E)  0E88 , C888 , ( F)
                                           -->
```

```
BLOCK #11
( DEFINITIONS FOR true lowercase TINY CHARACTERS continued)
04A8 , 8AA6 , ( G)  0AAA , EAAA , ( H)  0E44 , 444E , ( I)
0222 , 22A4 , ( J)  0AAC , CAAA , ( K)  0888 , 888E , ( E)
0AEE , AAAA , ( M)  0AAE , EEAA , ( N)  0EAA , AAAE , ( 0)
0CAA , C888 , ( P)  0EAA , AAEC , ( Q)  0CAA , CAAA , ( R)
0688 , 422C , ( S)  0E44 , 4444 , ( T)  0AAA , AAAE , ( U)
0AAA , AA44 , ( V)  0AAA , AEEA , ( W)  0AA4 , 44AA , ( X)
0AAA , E444 , ( Y)  0E24 , 488E , ( Z)  0644 , 4446 , ( [)
0884 , 4422 , ( \)  0C44 , 444C , ( ])  044A , A000 , ( $)
0000 , 000F , ( _)  0420 , 0000 , ( `)  000E , 2EAE , ( a)
088C , AAAC , ( b)  0006 , 8886 , ( c)  0226 , AAA6 , ( d)
0004 , AE86 , ( e)  0688 , E888 , ( f)  0006 , A62C , ( g)
088C , AAAA , ( h)  0404 , 4442 , ( i)  0202 , 22A4 , ( j)
088A , ACAA , ( k)  0444 , 4444 , ( l)  000A , EEAA , ( m)
0008 , EAAA , ( n)  0004 , AAA4 , ( o)  000C , AC88 , ( p)
                                      -->
```

```
BLOCK #12
( DEFINITIONS FOR true lowercase TINY CHARACTERS concluded)
0006 , A622 , ( q)  0008 , E888 , ( r)  0006 , 842C , ( s)
044E , 4442 , ( t)  000A , AAA6 , ( u)  000A , AAA4 , ( v)
000A , AEEA , ( w)  000A , A4AA , ( x)  000A , A62C , ( y)
000E , 248E , ( z)  0644 , 8446 , ( {)  0444 , 0444 , ( |)
0C44 , 244C , ( })  02E8 , 0000 , ( ~)  0EEE , EEEE , ( DEL)
TCHAR 43 BLOCK C2 MOVE FLUSH FORGET TCHAR    R->BASE
```

Appendix N TMS9900 Assembly Source Code for fbForth

This appendix includes the Assembly source for **fbForth**. It is based on the original TMS9900 Assembly source code from the two 90-KB diskettes made available to user groups when TI Forth was released into the Public Domain. The Assembly source code is in this **`BOLD, MONO-SPACED FONT`**. Most of the author's comments are in lowercase.

There are three source files:

1. fbForth_boot.a99 (based on TI Forth source file, BOOT)
2. fbForth_low-level-support.a99 (based on TI Forth source files, DRIVER, UTILEQU, UTILROM and UTILRAM)
3. fbForth_dictionary.a99 (based on TI Forth source files, ASMSRC1, ASMSRC2 and ASMSRC3)

The author assembled these in the order listed to a compressed object file, FBFORTH, with Cory Burr's PC-based Asm994a Assembler V3.010 (WinAsm99.exe). The compressed object file, FBFORTH, and the system blocks file, FBLOCKS, fit on a single, SSSD (90KB) diskette with only 23 sectors (5888 bytes) to spare. They can be copied to and run from any size media. FBFORTH, however, expects FBLOCKS to be on DSK1.

N.1 fbForth_boot.a99

The file fbForth_boot.a99 contains the startup routines for **fbForth**. It ends just after label, **`FOUR`**, by branching to the **fbForth** cold-start label, **`FF9900`**, of fbForth_dictionary.a99. It contains the system error messages, true lowercase table, patch code for slashed zero, space for blocks file PABs and default blocks file pathname (DSK1.FBLOCKS), all of which are copied to VRAM just before the **fbForth** cold start.

Prior to copying the above-mentioned tables to VRAM, the true lowercase patterns are loaded and the pattern for zero is patched. Once the cold-start branch occurs, the code in fbForth_boot.a99 is abandoned.

```
*    _____   ____         __  __     ___ ___
*   / _/ /  / __/__  ____/ /_/ /    <  // _ \
*  / _/ _ \/ _// _ \/ __/ __/ _ \  / // // /
* /_//_.__/_/  \___/_/  \__/_//_/  /_(_)___/
*
*              ____ __        __                   __
*             / _(_) /__ ____/ /  ___ ____ ___ ___/ /
*      _ _ _ / _/ / / -_)___/ _ \/ _ `(_-</ -_) _  /
*     (_|_|_)_//_/_/\__/   /_.__/\_,_/___/\__/\_,_/
*
*             __________  ____         __  __
*            /_  __/  _/ / __/__  ____/ /_/ /
*             / / _/ /  / _// _ \/ __/ __/ _ \
*            /_/ /___/ /_/  \___/_/  \__/_//_/

* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*                                                                   *
* fbForth --- a file-based TI Forth                                 *
* (C) Lee Stewart 2013                                              *
* ---written in TMS9900 Assembler for the TI-99/4A and based on     *
*    the the original public domain code written by Leon Tietz,     *
*    Leslie O'Hagan and Edward E. Ferguson                          *
*                                                                   *
* Filename: fbForth_boot.a99                                        *
*                                                                   *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

*    ______           __              ___            __  _
*   / __/ /____ _____/ /___ _____    / _ \___  __ __/ /_(_)__  ___ ___
*  _\ \/ __/ _ `/ __/ __/ // / _ \  / , _/ _ \/ // / __/ / _ \/ -_|_-<
* /___/\__/\_,_/_/  \__/\_,_/ .__/ /_/|_|\___/\_,_/\__/_/_//_/\__/___/
*                          /_/

* fbForth workspace registers
*
TEMP0  EQU  0
TEMP1  EQU  1
TEMP2  EQU  2
TEMP3  EQU  3
TEMP4  EQU  4
TEMP5  EQU  5
TEMP6  EQU  6
TEMP7  EQU  7
U      EQU  8
SP     EQU  9
W      EQU  10
LINK   EQU  11
CRU    EQU  12
IP     EQU  13
R      EQU  14
NEXT   EQU  15
************************************************************
*++ fbForth's workspace is 32 bytes at start of PAD (>8300->831F)
*
MAINWS EQU  >8300        IN CONSOLE CPU RAM
SUBPTR EQU  >8356        POINTS TO SUBROUTINE NAMES IN VDP
DSKERR EQU  >8350        DISK DSR ERROR CODES HERE
************************************************************
       DEF  BOOT
*
VSPTR  EQU  >836E        *++ pointer to value stack in VDP RAM
KYSTAT EQU  >837C        *++ GPL status byte
FAC    EQU  >834A        *++ Floating Point Accumulator
GRMWA  EQU  >9C02        *++ GROM write address register
GRMRA  EQU  >9802        *++ GROM read address register
GRMRD  EQU  >9800        *++ GROM read data register
XMLTAB EQU  >0CFA        *++ XML table in console ROM
```

```
************************************************************
************************************************************
       AORG >D000       <---workaround!!!

*++ location of fbForth's inner interpreter in PAD RAM

       DORG >832E        *++ code at FMOVE will be moved here
DODOES DECT SP           DUMMY COPY TO GET ADRESSES
       MOV  W,*SP
       MOV  LINK,W
DOCOL  DECT R
       MOV  IP,*R
       MOV  W,IP
$NEXT  MOV  *IP+,W
DOEXEC MOV  *W+,TEMP1
       B    *TEMP1
$SEMIS MOV  *R+,IP
       MOV  *IP+,W
       MOV  *W+,TEMP1
       B    *TEMP1
*
*
       AORG >D000

*++ this is the guts of fbForth's inner interpreter that gets moved to DODOES in PAD RAM

FMOVE  DECT SP           COPY TO MOVE TO CONSOLE RAM
       MOV  W,*SP
       MOV  LINK,W
       DECT R
       MOV  IP,*R
       MOV  W,IP
       MOV  *IP+,W
       MOV  *W+,TEMP1
       B    *TEMP1
       MOV  *R+,IP
       MOV  *IP+,W
       MOV  *W+,TEMP1
       B    *TEMP1
*
*++ program start

BOOT   LWPI MAINWS          *++ set up fbForth workspace in PAD
*
* set up GPL stuff
*
*++ finding and saving GROM address (682Dh) of XML instruction in E/A cartridge that got
*++ us here so we can use it to "return" from GPL to execute assembly code.  The GPL code
*++ in question ("XML >22") starts the E/A loader that loaded this fbForth BOOT program,
*++ which means the loader address is stored at CPU RAM address 2004h. until we change it
*++ in later code

       MOVB @GRMRA,TEMP1
       SWPB TEMP1
       MOVB @GRMRA,TEMP1
       SWPB TEMP1
       AI   TEMP1,-3
       MOV  TEMP1,@GRMSAV

*++ get object of GPL XML instruction, ">22" of "XML >22", into high byte of TEMP1

       INC  TEMP1
       MOVB TEMP1,@GRMWA
       SWPB TEMP1
       MOVB TEMP1,@GRMWA
       NOP
       MOVB @GRMRD,TEMP1

*++ calculate the XML vector by using first nybble of ">22" = 2 to look up the table's
*++ address in the console ROM's XML table at 0CFAh + (2 x 2) = 0CFEh (which contains
*++ 2000h) and adding the table's offset (the second nybble (2) x 2 = 4) to get 2004h,
*++ which is then stored in TEMP2

       MOV  TEMP1,TEMP2
       SRL  TEMP1,12
       SLA  TEMP1,1
       SLA  TEMP2,4
       SRL  TEMP2,11
       A    @XMLTAB(TEMP1),TEMP2

*++ save E/A loader's return address to the GPL interpreter in console ROM = 061Ch, which
*++ is by a "JMP >05E4" followed by a "B @>0070"

       MOV  @>2030,@SVGPRT       >2030 IS SVGPRT USED BY E/A LOADER

*++ move the address of our return from GPL (RTFGPL=3AB0h) to 2004h, the object of the
*++ GROM "XML >22" instruction noted above, which will be executed every time we return
*++ from GPL

       LI   TEMP1,RTFGPL
       MOV  TEMP1,*TEMP2

*++ copy fbForth's inner interpreter code (26 bytes) from FMOVE to where it will execute
*++ in PAD at 832Eh

       LI   TEMP1,BOOT-FMOVE
       LI   TEMP2,FMOVE
       LI   TEMP3,DODOES
MLOOP  MOV  *TEMP2+,*TEMP3+
       DECT TEMP1
       JNE  MLOOP
*
*** INITIALIZE VDP STUFF
*
       LI   TEMP0,>01B0  BLANK SCREEN
       BLWP @VWTR
       LI   TEMP0,>030E  SET COLOR TABLE AT >0380
       BLWP @VWTR
       LI   TEMP0,>0401  SET PATTERN DESCRIPTOR TABLE >0800
       BLWP @VWTR
       LI   TEMP0,>0506  SET SPRITE ATTRIBUTE TABLE >0300
       BLWP @VWTR
       LI   TEMP0,>0601  SET SPRITE DESCRIPTOR TABLE >0800
       BLWP @VWTR
       LI   TEMP0,>07F4  SET TEXTMODE COLORS
       BLWP @VWTR
       LI   TEMP0,>2000  BLANK
       LI   TEMP1,>960   TEXT-MODE SCREEN SIZE  <--this should be either 960 or >3C0
       LI   TEMP2,>0     SCREEN STARTS AT 0
       BL   @FILLER      CLEAR SCREEN
       LI   TEMP0,>FF00  CHAR FF
       LI   TEMP1,>2048  BLOCK SIZE             <--this should be either 2048 or >800
       LI   TEMP2,>800   STARTING LOCATION IN VDP
       BL   @FILLER      FILL AREA WITH FF'S

*++ force text mode

       LI   TEMP0,>81F0
       SWPB TEMP0
       MOVB TEMP0,@>83D4 USED TO UPDATE VDP REG EACH KEYSTROKE
       MOVB TEMP0,@VDPWA FORCE TEXT MODE
       SWPB TEMP0
       MOVB TEMP0,@VDPWA

*++ load capital letters

       LI   TEMP0,>900   VDP LOCATION
       MOV  TEMP0,@FAC   FAC must contain VDP start address
       CLR  TEMP1        CLEAR GPL STATUS
       MOVB TEMP1,@KYSTAT
       LI   TEMP7,>3E0
       MOV  TEMP7,@VSPTR
       BLWP @GPLLNK      LOAD CAPITAL LETTER SHAPES
       DATA >0018
*
* patch zero pattern
*
       LI   TEMP0,>983
       LI   TEMP1,ZPATCH
       LI   TEMP2,3
       BLWP @VMBW
*
* patch in XML 23 location of our CIF (Convert Integer to Floating point)
*
       LI   TEMP2,CIF
       MOV  TEMP2,@>2006
*
* load system messages, true lowercase and blocks PABs to VRAM
*
       MOV  @UBASE0+$DKBUF,TEMP2    initial VRAM location of Forth disk buffer
       A    @UBASE0+$VFSM,TEMP2     calculate VRAM location of system messages, etc.
       ORI  TEMP2,>4000             set bit for VDP write
       SWPB TEMP2
       MOVB TEMP2,@VDPWA        LS byte first
       SWPB TEMP2
       MOVB TEMP2,@VDPWA        then MS byte
       NOP                      kill time
       LI   TEMP1,PBEND-MSGS    block size
       LI   TEMP0,MSGS          initial RAM location of system messages, etc.
MSLOOP MOVB *TEMP0+,@VDPWD      write a byte
       DEC  TEMP1
       JNE  MSLOOP              not done; write another byte
*
       LI   TEMP2,>1200
       CB   TEMP2,@3            if byte @3 in the console is >12 it's a 99/4
       JEQ  FOUR                don't load lower case in a 99/4
*
* load lower case in 99/4a
*
       LI   TEMP2,>0B00         VRAM location of lowercase patterns
       ORI  TEMP2,>4000         set bit for VDP write
       SWPB TEMP2
       MOVB TEMP2,@VDPWA        LS byte first
       SWPB TEMP2
       MOVB TEMP2,@VDPWA        then MS byte
       NOP                      kill time
       LI   TEMP1,TLCEND-TLCTAB block size
       LI   TEMP0,TLCTAB        initial RAM location of true lowercase patterns
LCLOOP MOVB *TEMP0+,@VDPWD      write a byte
       DEC  TEMP1
       JNE  LCLOOP              not done; write another byte

*++ finally, we cold start fbForth
```

```
FOUR   LI   TEMP1,UBASE0
       B    @FF9900      BRANCH TO COLD start
*
FILLER ORI  TEMP2,>4000  SET BIT FOR VDP WRITE
       SWPB TEMP2
       MOVB TEMP2,@VDPWA LS BYTE FIRST
       SWPB TEMP2
       MOVB TEMP2,@VDPWA THEN MS BYTE
       NOP               KILL TIME
FLLOOP MOVB TEMP0,@VDPWD WRITE A BYTE
       DEC  TEMP1
       JNE  FLLOOP       NOT DONE, FILL ANOTHER
       B    *LINK

*    __  ___
*   /  |/  /__ ___ ___ ___ ____ ____ ___
*  / /|_/ / -_|_-<(_-</ _ `/ _ `/ -_|_-<
* /_/  /_/\__/___/___/\_,_/\_, /\__/___/
*                         /___/
*
* Messages to be loaded into VRAM
*
MSGS   DATA 25          ...highest message number in table
MSG01  BYTE 11                                      msg  1
       TEXT "empty stack"
MSG02  BYTE 15                                      msg  2
       TEXT "dictionary full"
MSG04  BYTE 13                                      msg  4
       TEXT "isn't unique."
MSG05  BYTE 19                                      msg  5
       TEXT "FBLOCKS not current"
MSG06  BYTE 10                                      msg  6
       TEXT "disk error"
MSG07  BYTE 10                                      msg  7
       TEXT "full stack"
MSG08  BYTE 20                                      msg  8
       TEXT "block # out of range"
MSG09  BYTE 14                                      msg  9
       TEXT "file I/O error"
MSG10  BYTE 20                                      msg 10
       TEXT "floating point error"
MSG17  BYTE 16                                      msg 17
       TEXT "compilation only"
MSG18  BYTE 14                                      msg 18
       TEXT "execution only"
MSG19  BYTE 23                                      msg 19
       TEXT "conditionals not paired"
MSG20  BYTE 23                                      msg 20
       TEXT "definition not finished"
MSG21  BYTE 23                                      msg 21
       TEXT "in protected dictionary"
MSG22  BYTE 21                                      msg 22
       TEXT "use only when loading"
MSG25  BYTE 14                                      msg 25
       TEXT "bad jump token"

*  ______               __
* /_  __/_____ _____   / /  ___ _    _____ ___________ ____ ___
*  / / / __/ // / -_) / /__/ _ \ |/|/ / -_) __/ __/ _ `(_-</ -_)
* /_/ /_/  \_,_/\__/ /____/\___/__,__/\__/_/  \__/\_,_/___/\__/
*
* True lowercase characters for non-bitmap modes
*   ...to be loaded into VRAM
*
TLCTAB DATA >0000,>2010,>0800,>0000    ( `)
       DATA >0000,>0038,>043C,>443C    ( a)
       DATA >0040,>4078,>4444,>4478    ( b)
       DATA >0000,>003C,>4040,>403C    ( c)
       DATA >0004,>043C,>4444,>443C    ( d)
       DATA >0000,>0038,>447C,>4038    ( e)
       DATA >0018,>2420,>7020,>2020    ( f)
       DATA >0000,>003C,>443C,>0438    ( g)
       DATA >0040,>4058,>6444,>4444    ( h)
       DATA >0010,>0030,>1010,>107C    ( i)
       DATA >0004,>0004,>0404,>4438    ( j)
       DATA >0040,>4044,>4870,>4844    ( k)
       DATA >0030,>1010,>1010,>107C    ( l)
       DATA >0000,>0068,>5454,>5454    ( m)
       DATA >0000,>0058,>6444,>4444    ( n)
       DATA >0000,>0038,>4444,>4438    ( o)
       DATA >0000,>0078,>4478,>4040    ( p)
       DATA >0000,>0038,>443C,>0404    ( q)
       DATA >0000,>0058,>6440,>4040    ( r)
       DATA >0000,>003C,>4038,>0478    ( s)
       DATA >0010,>107C,>1010,>1408    ( t)
       DATA >0000,>0044,>4444,>4C34    ( u)
       DATA >0000,>0044,>4444,>2810    ( v)
       DATA >0000,>0044,>4454,>5428    ( w)
       DATA >0000,>0044,>2810,>2844    ( x)
       DATA >0000,>0044,>4C34,>0438    ( y)
       DATA >0000,>007C,>0810,>207C    ( z)
       DATA >0018,>2020,>4020,>2018    ( {)
       DATA >0010,>1010,>0010,>1010    ( |)
       DATA >0030,>0808,>0408,>0830    ( })
       DATA >0000,>2054,>0800,>0000    ( ~)
TLCEND
ZPATCH DATA >4C54,>6400
*
* blocks file PABs and default blocks file pathname
*
BPAB$  BSS  70
       BSS  70
DEFNAM BYTE 12
       TEXT "DSK1.FBLOCKS                "
       TEXT "                            "
PBEND  DATA 0       dummy
```

N.2 fbForth_low-level-support.a99

The file fbForth_low-level-support.a99 contains the low-level system support for **fbForth**, which is virtually all of the functionality that the Editor/Assembler cartridge loads into low memory expansion RAM, except that it is unique to **fbForth**'s needs and not at the same locations. This is due to the fact that the area from `2010h` – `3423h` is reserved for the five **fbForth** block buffers. If you are comparing this file with TI Forth's DRIVER, you will notice that all of the extraneous startup code has been removed.

```
*
*    _____   ____         __  __     ___ ___
*   / _/ /  / __/__  ____/ /_/ /    <  // _ \
*  / _/ _ \/ _// _ \/ __/ __/ _ \   / // // /
* /_//_.__/_/  \___/_/  \__/_//_/  /_(_)___/
*
*              ____ __        __                   __
*             / _(_) /__ ____/ /  ___ ____ ___ ___/ /
*      _ _ _ / _/ / / -_)___/ _ \/ _ `(_-</ -_) _  /
*     (_|_|_)_//_/_/\__/   /_.__/\_,_/___/\__/\_,_/
*
*             __________  ____         __  __
*            /_  __/  _/ / __/__  ____/ /_/ /
*             / / _/ /  / _// _ \/ __/ __/ _ \
*            /_/ /___/ /_/  \___/_/  \__/_//_/

* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*                                                                   *
* fbForth --- a file-based TI Forth                                 *
* (C) Lee Stewart 2013                                              *
* ---written in TMS9900 Assembler for the TI-99/4A and based on     *
*    the the original public domain code written by Leon Tietz,     *
*    Leslie O'Hagan and Edward E. Ferguson                          *
*                                                                   *
* Filename: fbForth_low-level-support.a99                           *
*                                                                   *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

*    __                  __             __  ____                        __
*   / /  ___ _    ______/ /__ _  _____ / / / __/_ _____  ___  ___  ____/ /_
*  / /__/ _ \ |/|/ /___/ / -_) |/ / -_) / _\ \/ // / _ \/ _ \/ _ \/ __/ __/
* /____/\___/__,__/   /_/\__/|___/\__/_/ /___/\_,_/ .__/ .__/\___/_/  \__/
*                                                /_/  /_/

* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*                                                                   *
* fbForth---                                                        *
*                                                                   *
*       Low-level support routines                                  *
*                                                                   *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*

MAINWS EQU  >8300        IN CONSOLE CPU RAM
KYCHAR EQU  >8375
VDPSTA EQU  >8802

       AORG >2002       <---workaround!!!

*++ fbForth's inner interpreter uses 26 bytes in PAD

       DORG >832E
DODOES DECT SP           DUMMY COPY TO GET ADRESSES
       MOV  W,*SP
       MOV  LINK,W
DOCOL  DECT R
       MOV  IP,*R
       MOV  W,IP
$NEXT  MOV  *IP+,W
DOEXEC MOV  *W+,TEMP1
       B    *TEMP1
$SEMIS MOV  *R+,IP
       MOV  *IP+,W
       MOV  *W+,TEMP1
       B    *TEMP1
       AORG >2002       <---workaround!!!
       DORG 0
UBASE  BSS  6             BASE OF USER VARIABLES
```

```
$UCONS BSS  2            06 USER UCONS
$S0    BSS  2            08 USER S0
$R0    BSS  2            0A USER R0 { R0$
$U0    BSS  2            0C USER U0
       BSS  2            0E USER TIB
       BSS  2            10 USER WIDTH
       BSS  2            12 USER DP
$SYS   BSS  2            14 USER SYS$
CURPO$ BSS  2            16 USER CURPOS
       BSS  2            18 USER INTLNK
       BSS  2            1A USER WARNING
       BSS  2            1C USER C/L$ { CL$
       BSS  2            1E USER FIRST$
       BSS  2            20 USER LIMIT$
$VFSM  BSS  2            22 USER MGT  <--offset from DISK_BUF
$VTLC  BSS  2            24 USER LCT  <--offset from DISK_BUF
       BSS  2            26 USER JMODE    loaded by graphics primitives; used by JOYST
       BSS  2            28 USER      <---available
$DEFBF BSS  2            2A USER DBF  <--offset from DISK_BUF of default blocks filename
$DKBUF BSS  2            2C USER DISK_BUF  (BUFFER LOC IN VDP. SIZE=128)  128)
$PABS  BSS  2            2E USER PABS      (AREA FOR PABS ETC.)
$SWDTH BSS  2            30 USER SCRN_WIDTH
$SSTRT BSS  2            32 USER SCRN_START
$SEND  BSS  2            34 USER SCRN_END
$ISR   BSS  2            36 USER ISR
$ALTI  BSS  2            38 USER ALTIN
$ALTO  BSS  2            3A USER ALTOUT
       BSS  2            3C USER VDPMDE    permanent location for VDPMDE
$BPABS BSS  2            3E USER BPB   <--offset from DISK_BUF of PABs for blocks files
$BPOFF BSS  2            40 USER BPOFF <--offset into BPABS for cur. blocks file's PAB
*                                      ...always toggled between 0 and 70
       BSS  2            42 USER FENCE
       BSS  2            44 USER BLK
       BSS  2            46 USER IN
$OUT   BSS  2            48 USER OUT
       BSS  2            4A USER SCR
       BSS  2            4C USER CONTEXT
       BSS  2            4E USER CURRENT
       BSS  2            50 USER STATE
       BSS  2            52 USER BASE
       BSS  2            54 USER DPL
       BSS  2            56 USER FLD
       BSS  2            58 USER CSP
       BSS  2            5A USER R# <--RNUM
       BSS  2            5C USER HLD
       BSS  2            5E USER USE
       BSS  2            60 USER PREV
       BSS  2            62 USER ECOUNT
       BSS  2            64 USER VOC-LINK
UMAX   BSS  0
*
*++ parameter stack grows down toward HERE
*++ TIB (text input buffer) starts at same address, but goes the other way
*++ TIB is 82 bytes long

SPBASE EQU  >FFA0       BASE OF PARAMETER STACK & TIB

*++ return stack grows down toward system support

RBASE  EQU  >3FFE       BASE OF RETURN STACK
*
*** UTILITY EQUATES ******
*
SCNKEY EQU  >000E
FLAG2  EQU  >8349
SCLEN  EQU  >8355
```

```
SCNAME EQU  >8356
SUBSTK EQU  >8373
CRULST EQU  >83D0
SADDR  EQU  >83D2
GPLWS  EQU  >83E0        GPL/EXTENDED BASIC WORKSPACE
SCRPAD EQU  >8300
VDPRD  EQU  >8800        VDP read data address
VDPWD  EQU  >8C00        VDP write data address
VDPWA  EQU  >8C02        VDP write address address
R0LB   EQU  >83E1
R1LB   EQU  >83E3
R3LB   EQU  >83E7
*
*
       AORG >2010
$BUFF  BSS  5*>404        fbForth block I/O buffers
*
$LO    BSS  0       start of low-level routines (>3424)
*
*
*** INTERRUPT SERVICE
*
INT1   LI   TEMP1,INT2   FIX 'NEXT' SO THAT INTERRUPT IS
       MOV  TEMP1,@2*NEXT+MAINWS  PROCESSED AT END OF
       LWPI >83C0        NEXT 'CODE' WORD
       RTWP
*
INT2   LIMI 0
       MOVB @>83D4,TEMP0
       SRL  TEMP0,8
       ORI  TEMP0,>100
       ANDI TEMP0,>FFDF
       BLWP @VWTR        TURN OFF VDP INTERRUPTS
       LI   NEXT,$NEXT   RESTORE 'NEXT'
       SETO @INTACT
       DECT R            SET UP RETURN LINKAGE
       MOV  IP,*R
       LI   IP,INT3
       MOV  @$ISR(U),W   DO THE FORTH ROUTINE
       B    @DOEXEC
INT3   DATA $+2
       DATA $+2
       MOV  *R+,IP
       CLR  @INTACT
       MOVB @>83D4,TEMP0
       SRL  TEMP0,8
       AI   TEMP0,>100
       MOVB @VDPSTA,TEMP1 REMOVE PENDING INTERRUPT
       BLWP @VWTR
       LIMI 2
       B    *NEXT        CONTINUE NORMAL TASK
*===========================================================
BKLINK MOV  @INTACT,TEMP7
       JNE  BKLIN1
       LIMI 2
BKLIN1 B    *LINK
*===========================================================
*
$SYS$  LIMI 0
       MOV  @SYSTAB(TEMP1),TEMP0
       B    *TEMP0
       DATA BRW          CODE=-20 write block to blocks file
       DATA BRW          CODE=-18 read block from blocks file
       DATA BRW          CODE=-16 create blocks file
       DATA BRW          CODE=-14 use blocks file
       DATA GXY          CODE=-12 GOTOXY
       DATA QKY          CODE=-10 ?KEY
       DATA QTM          CODE=-8  ?TERMINAL
       DATA CLF          CODE=-6  CRLF
       DATA EMT          CODE=-4  EMIT
       DATA KY           CODE=-2  KEY
SYSTAB DATA SBW          CODE=0   VSBW
       DATA MBW          CODE=2   VMBW
       DATA SBR          CODE=4   VSBR
       DATA MBR          CODE=6   VMBR
       DATA WTR          CODE=8   VWTR
       DATA GPL          CODE=10  GPLLNK
       DATA XML          CODE=12  XMLLNK
       DATA DSR          CODE=14  DSRLNK
       DATA CLS$         CODE=16  CLS
       DATA FMT          CODE=18  FORMAT-DISK   <--to be removed or re-purposed
       DATA FILL$        CODE=20  VFILL
       DATA AOX          CODE=22  VAND
       DATA AOX          CODE=24  VOR
       DATA AOX          CODE=26  VXOR
*
*== THIS IS A VDP SINGLE BYTE WRITE.  CODE=0  ==============
*
SBW    MOV  *SP+,TEMP0   VDP ADDRESS (DESTINATION)
       MOV  *SP+,TEMP1   CHARACTER TO WRITE
       SWPB TEMP1        GET IN LEFT BYTE
       BLWP @VSBW
       B    @BKLINK
*
*== THIS IS A VDP MULTI BYTE WRITE.  CODE=2   ==============
*
MBW    MOV  *SP+,TEMP2   NUMBER OF BYTES TO MOVE
       MOV  *SP+,TEMP0   VDP ADDRESS (DESTINATION)
       MOV  *SP+,TEMP1   RAM ADDRESS (SOURCE)
       BLWP @VMBW
       B    @BKLINK
*
*== THIS IS A VDP SINGLE BYTE READ.  CODE=4   ==============
*
SBR    MOV  *SP,TEMP0    VDP ADDRESS (SOURCE)
       BLWP @VSBR
       SRL  TEMP1,8      CHARACTER TO RIGHT HALF FOR FORTH
       MOV  TEMP1,*SP    STACK IT
       B    @BKLINK
*
*== THIS IS A VDP MULTI BYTE READ.  CODE=6    ==============
*
MBR    MOV  *SP+,TEMP2   NUMBER OF BYTES TO READ
       MOV  *SP+,TEMP1   RAM ADDRESS (DESTINATION)
       MOV  *SP+,TEMP0   VDP ADDRESS (SOURCE)
       BLWP @VMBR
       B    @BKLINK
*
*== VDP REGISTER WRITE.  CODE=8               ==============
*
WTR    MOV  *SP+,TEMP1   VDP REGISTER NUMBER
       MOV  *SP+,TEMP0   DATA FOR REGISTER
       SWPB TEMP1        GET REGISTER TO LEFT BYTE
       MOVB TEMP1,TEMP0  PLACE WITH DATA
       BLWP @VWTR
       B    @BKLINK
*
*== THIS IS THE GPL LINK UTILITY.  CODE=10    ==============
*
GPL    CLR  TEMP0
       MOVB TEMP0,@KYSTAT
       LI   TEMP0,>0420       CONSTRUCT THE BLWP INSTRUCTION
       LI   TEMP1,GPLLNK      TO THE GPLLNK UTILITY
       MOV  *SP+,TEMP2        WITH THIS DATA IDENTIFYING THE ROUTINE
       LI   TEMP3,>045B       CONSTRUCT THE B *LINK INSTRUCTION
       MOV  LINK,TEMP4        SAVE LINK ADDRESS
       BL   @2*TEMP0+MAINWS   EXECUTE THE ABOVE INSTRUCTIONS
       MOV  TEMP4,LINK        AND RECONSTRUCT LINK
       B    @BKLINK
*
*== THIS IS THE XML LINK UTILITY.  CODE=12    ==============
*
XML    LI   TEMP0,>0420       CONSTRUCT THE BLWP INSTRUCTION
       LI   TEMP1,XMLLNK      TO THE XMLLNK UTILITY
       MOV  *SP+,TEMP2        WITH THIS DATA IDENTIFYING THE ROUTINE
       LI   TEMP3,>045B       CONSTRUCT THE B *LINK INSTRUCTION
       MOV  LINK,TEMP4        SAVE LINK ADDRESS
       BL   @2*TEMP0+MAINWS   EXECUTE THE ABOVE INSTRUCTIONS
       MOV  TEMP4,LINK        AND RECONSTRUCT LINK
       B    @BKLINK
*
*== THIS IS THE DSR LINK UTILITY.  CODE=14    ==============
*
DSR    LI   TEMP0,>0420       CONSTRUCT THE BLWP INSTRUCTION
       LI   TEMP1,DSRLNK      TO THE DSRLNK UTILITY
       MOV  *SP+,TEMP2        THIS DATUM SELECTS DSR OR SUBROUTINE
       LI   TEMP3,>045B       CONSTRUCT THE B *LINK INSTRUCTION
       MOV  LINK,TEMP4        SAVE LINK ADDRESS
       BL   @2*TEMP0+MAINWS   EXECUTE THE ABOVE INSTRUCTIONS
       MOV  TEMP4,LINK        AND RECONSTRUCT LINK
       B    @BKLINK
*
*== THIS IS THE SCREEN CLEARING UTILITY.  CODE=16 ==========
*
CLS$   MOV  @$SSTRT(U),TEMP2     BEGINNING OF SCREEN IN VDP
       MOV  @$SEND(U),TEMP1      END OF SCREEN IN VDP
       S    TEMP2,TEMP1       SCREEN SIZE
       LI   TEMP0,>2000       BLANK CHARACTER
       MOV  LINK,TEMP7
       BL   @FILL1
       MOV  TEMP7,LINK
FMT    B    @BKLINK         <---the label insures FMT does nothing
*
*== THIS IS THE DISK FORMATTER.  CODE=18   <<<< === disabled === >>>>
*
* FMT  <---this label moved to branch above to make sure it only returns
*
*== THIS IS THE VDP FILL ROUTINE.  CODE=20
*
FILL$  MOV  *SP+,TEMP0   FILL CHARACTER
       SWPB TEMP0        TO LEFT BYTE
       MOV  *SP+,TEMP1   FILL COUNT
       MOV  *SP+,TEMP2   ADDRESS TO START VDP FILL
       MOV  LINK,TEMP7
       BL   @FILL1
       MOV  TEMP7,LINK
       B    @BKLINK
*===========================================================
FILL1  ORI  TEMP2,>4000  SET BIT FOR VDP WRITE
       SWPB TEMP2
       MOVB TEMP2,@VDPWA LS BYTE FIRST
       SWPB TEMP2
       MOVB TEMP2,@VDPWA THEN MS BYTE
       NOP               KILL TIME
FLOOP  MOVB TEMP0,@VDPWD WRITE A BYTE
       DEC  TEMP1
       JNE  FLOOP        NOT DONE, FILL ANOTHER
       B    *LINK
```

```
*============================================================
*
*== VDP BYTE 'AND' 'OR' 'XOR' ROUTINES.  CODE=22,24,26   ===
*
AOX    MOV  *SP+,TEMP2   VDP ADDRESS
       SWPB TEMP2
       MOVB TEMP2,@VDPWA LS BYTE FIRST
       SWPB TEMP2
       MOVB TEMP2,@VDPWA THEN MS BYTE
       NOP               KILL TIME
       MOVB @VDPRD,TEMP3 READ BYTE
       MOV  *SP+,TEMP0   GET DATA TO OPERATE WITH
       SWPB TEMP0        TO LEFT BYTE
*** NOW DO REQUESTED OPERATION *****************
       CI   TEMP1,24
       JEQ  DOOR
       JGT  DOXOR
       INV  TEMP3        THESE TWO INSTRUCTIONS
       SZC  TEMP3,TEMP0  PERFORM AN 'AND'
       JMP  FINAOX
DOOR   SOC  TEMP3,TEMP0  PERFORM OR
       JMP  FINAOX
DOXOR  XOR  TEMP3,TEMP0  PERFORM XOR
FINAOX LI   TEMP1,1
       MOV  LINK,TEMP7
       BL   @FILL1
       MOV  TEMP7,LINK
       B    @BKLINK
*
*============================================================
*
*== KEY ROUTINE  CODE= -2  =================================
*
KY     MOV  @$ALTI(U),TEMP0
       JEQ  KEY0
       CLR  TEMP7
       MOVB TEMP7,@KYSTAT
       INC  TEMP0
       BLWP @VSBR
       ANDI TEMP1,>1F00
       BLWP @VSBW
       MOV  TEMP0,TEMP1
       AI   TEMP1,8
       MOV  TEMP1,@SUBPTR
       BLWP @DSRLNK
       DATA >8
       DECT TEMP0
       BLWP @VSBR
       SRL  TEMP1,8
       MOV  TEMP1,TEMP0
       B    @BKLINK
KEY0   MOV  @KEYCNT,TEMP7
       INC  TEMP7
       JNE  KEY1
       MOV  @CURPO$(U),TEMP0
       BLWP @VSBR             READ CHARACTER AT CURSOR POSITION
       MOVB TEMP1,@CURCHR     AND SAVE IT
       LI   TEMP1,>1E00       PLACE CURSOR CHARACTER ON SCREEN
       BLWP @VSBW
*
KEY1   LI   TEMP4,>2000       MASK TO CHECK STATUS
       BLWP @KSCAN
       MOVB @KYSTAT,TEMP0
       COC  TEMP4,TEMP0
       JEQ  KEY2              JMP IF KEY WAS PRESSED
*
```

```
       CI   TEMP7,100         NO KEY PRESSED
       JNE  KEY3
       MOVB @CURCHR,TEMP1
       JMP  KEY5
*
KEY3   CI   TEMP7,200
       JNE  KEY4
       CLR  TEMP7
       LI   TEMP1,>1E00       CURSOR CHAR
KEY5   MOV  @CURPO$(U),TEMP0
       BLWP @VSBW
KEY4   MOV  TEMP7,@KEYCNT
       MOV  @INTACT,TEMP7
       JNE  KEY6
       LIMI 2
KEY6   DECT IP                THIS WILL RE-EXECUTE KEY
       B    *NEXT
*KE     DATA KEY,SEMIS
*
*
KEY2   SETO @KEYCNT           KEY WAS PRESSED
       MOV  @CURPO$(U),TEMP0  RESTORE CHARACTER AT CURSOR LOCATION
       MOVB @CURCHR,TEMP1
       BLWP @VSBW
       MOVB @KYCHAR,TEMP0     PUT CHAR IN RIGHT HALF OF TEMP0
       SRL  TEMP0,8
       B    @BKLINK
*
*== EMIT ROUTINE  CODE= -4  ================================
*
EMT    MOV  TEMP2,TEMP1
       MOV  @$ALTO(U),TEMP0
       JEQ  EMIT0
       CLR  TEMP7             ALTOUT ACTIVE
       MOVB TEMP7,@KYSTAT
       DEC  TEMP0
       SWPB TEMP1
       BLWP @VSBW
       INCT TEMP0
       BLWP @VSBR
       ANDI TEMP1,>1F00
       BLWP @VSBW
       AI   TEMP0,8
       MOV  TEMP0,@SUBPTR
       BLWP @DSRLNK
       DATA >8
       B    @BKLINK
*
EMIT0  CI   TEMP1,7      IS IT A BELL?
       JNE  NOTBEL
       CLR  TEMP2
       MOVB TEMP2,@KYSTAT
       MOVB @GRMSAV,@GRMWA    RESTORE GROM ADDRESS
       NOP
       MOVB @GRMSAV+1,@GRMWA
       BLWP @GPLLNK
       DATA >0036        EMIT ERROR TONE
       JMP  EMEXIT
*
NOTBEL CI   TEMP1,8      IS IT A BACKSPACE?
       JNE  NOTBS
       LI   TEMP1,>2000
       MOV  @CURPO$(U),TEMP0
       BLWP @VSBW
       JGT  DECCUR
       JMP  EMEXIT
```

```
DECCUR DEC  @CURPO$(U)
       JMP  EMEXIT
*
NOTBS  CI   TEMP1,>A     IS IT A LINE FEED?
       JNE  NOTLF
       MOV  @$SEND(U),TEMP7
       S    @$SWDTH(U),TEMP7
       C    @CURPO$(U),TEMP7
       JHE  SCRLL
       A    @$SWDTH(U),@CURPO$(U)
       JMP  EMEXIT
SCRLL  MOV  LINK,TEMP7
       BL   @SCROLL
       MOV  TEMP7,LINK
       JMP  EMEXIT
*
*** SCROLLING ROUTINE
*
SCROLL MOV  @$SSTRT(U),TEMP0     VDP ADDR
       LI   TEMP1,LINBUF      BUFFER
       MOV  @$SWDTH(U),TEMP2     COUNT
       A    TEMP2,TEMP0       START AT LINE 2
SCROL1 BLWP @VMBR
       S    TEMP2,TEMP0       ONE LINE BACK TO WRITE
       BLWP @VMBW
       A    TEMP2,TEMP0       TWO LINES AHEAD FOR NEXT READ
       A    TEMP2,TEMP0
       C    TEMP0,@$SEND(U)      END OF SCREEN?
       JL   SCROL1
       MOV  TEMP2,TEMP1       BLANK BOTTOM ROW OF SCREEN
       LI   TEMP0,>2000       BLANK
       S    @$SEND(U),TEMP2
       NEG  TEMP2             NOW CONTAINS ADDRESS OF START OF LAST LINE
       MOV  LINK,TEMP6
       BL   @FILL1            WRITE THE BLANKS
       B    *TEMP6
*
NOTLF  CI   TEMP1,>D     IS IT A CARRIAGE RETURN?
       JNE  NOTCR
       CLR  TEMP0
       MOV  @CURPO$(U),TEMP1
       MOV  TEMP1,TEMP3
       S    @$SSTRT(U),TEMP1  ADJUSTED FOR SCREEN NOT AT 0
       MOV  @$SWDTH(U),TEMP2
       DIV  TEMP2,TEMP0
       S    TEMP1,TEMP3
       MOV  TEMP3,@CURPO$(U)
       JMP  EMEXIT
*
NOTCR  SWPB TEMP1        ASSUME IT IS A PRINTABLE CHARACTER
       MOV  @CURPO$(U),TEMP0
       BLWP @VSBW
       MOV  @$SEND(U),TEMP2
       DEC  TEMP2
       C    TEMP0,TEMP2
       JNE  NOTCR1
       MOV  @$SEND(U),TEMP0
       S    @$SWDTH(U),TEMP0 WAS LAST CHAR ON SCREEN. SCROLL
       MOV  TEMP0,@CURPO$(U)
       JMP  SCRLL
NOTCR1 INC  TEMP0        NO SCROLL NECESSARY
       MOV  TEMP0,@CURPO$(U)
*
EMEXIT B    @BKLINK
*
*== CRLF ROUTINE  CODE= -6  ================================
*
CLF    MOV  LINK,TEMP5
       LI   TEMP2,>000D
       BL   @EMT
       LI   TEMP2,>000A
       LIMI 0            PREVIOUS CALL TO EMT ALTERED INT MASK
       BL   @EMT
       MOV  TEMP5,LINK
       B    @BKLINK
*
*== ?TERMINAL ROUTINE CODE= -8  ============================
*
QTM    BLWP @KSCAN
       MOVB @KYCHAR,TEMP0
       SRL  TEMP0,8
       CI   TEMP0,>2
       JEQ  QTERM1
       CLR  TEMP0
QTERM1 B    @BKLINK
*
*== ?KEY ROUTINE CODE= -10  ================================
*
QKY    BLWP @KSCAN
       MOVB @KYCHAR,TEMP0
       SRL  TEMP0,8
       CI   TEMP0,>00FF
       JNE  QKEY1
       CLR  TEMP0
QKEY1  B    @BKLINK
*
*== GOTOXY ROUTINE CODE= -12  ==============================
*
GXY    MPY  @$SWDTH(U),TEMP3
       A    TEMP2,TEMP4       POSITION WITHIN SCREEN
       A    @$SSTRT(U),TEMP4     ADD VDP OFFSET TO SCREEN TOP
       MOV  TEMP4,@CURPO$(U)
       B    @BKLINK

*    ___  __         __     ____  ______
*   / _ )/ /__  ____/ /__  /  _/_/_/ __ \
*  / _  / / _ \/ __/  '_/ _/ /_/_// /_/ /
* /____/_/\___/\__/_/\_\ /___/_/  \____/

*
* === block file I/O support ===============================
*
* BPTOG utility to toggle one of 2 PABs for block file access
*
BPTOG  MOV  @$BPOFF(U),R0    PAB offset to R0
       LI   R1,70            toggle amount
       XOR  R0,R1            new offset
       MOV  R1,@$BPOFF(U)    update offset
*
**xxx** entry point to insure we have correct PAB address
*
BPSET  MOV  @$DKBUF(U),R0    get DISK_BUF address
       A    @$BPABS(U),R0    get BPABS address
*
       A    @$BPOFF(U),R0    add current offset
       MOV  R0,@BFPAB        update current block file's PAB address
       RT
*
* CLOSE blocks file
*
BKCLOS MOV  @BFPAB,R0
       LI   R1,$FCLS            opcode=CLOSE
       BLWP @VSBW
       AI   R0,9                address of filename's char count
       MOV  R0,@SUBPTR          point to filename's char count
       BLWP @DSRLNK             close the file
       DATA 8
       RT                       deal with error in caller
*
* storage area
*
SVBRET DATA 0          storage for LINK coming into BRW
BFPAB  DATA 0          storage for current blocks file PAB address
*                      ...will have current PAB on entry
* PAB header storage
*
PABHD  BSS  4        BYTE 0: opcode 0=OPEN,1=CLOSE,2=READ,3=WRITE,4=RESTORE
*                    BYTE 1: >05=INPUT mode + clear error,fixed,display,relative
*                            >03=OUTPUT mode + "
*                            >01=UPDATE mode + "
*                    BYTE 2,3: save contents of DISK_BUF here
       BYTE >80      record length
       BYTE >80      character count of transfer
       BSS  2        record number
*
*** file I/O equates
*
$FOPN  EQU  >0000
$FCLS  EQU  >0100
$FRD   EQU  >0200
$FWRT  EQU  >0300
$FRST  EQU  >0400
$FINP  EQU  5
$FOUT  EQU  3
$FUPD  EQU  1
*
*** entry point for block read/write routines
*
BRW    MOV  LINK,@SVBRET      save LINK address
       MOV  TEMP1,TEMP7       save CODE {TEMP1 to TEMP7}
       SRA  TEMP7,1           divide CODE by 2 (now -7,-8,-9,-10)
       AI   TEMP7,12          CODE + 12 (now 5,4,3,2, with OP for output, but not input)
       BL   @BPSET            insure correct PAB address in BFPAB (it may have moved)
       CI   TEMP7,4           USE or CREATE?
       JLT  BRW01             no
       BL   @BPTOG            yes...toggle BPOFF & BFPAB
       MOV  @BFPAB,TEMP0      load PAB address
       AI   TEMP0,9           set to name length byte
       CLR  TEMP2
       MOV  *SP+,TEMP1        pop bfnaddr to TEMP1
       MOVB *TEMP1,@MAINWS+5  copy length byte to low byte of TEMP2
       INC  TEMP2             add 1 to # bytes to copy
       BLWP @VMBW             copy char count & pathname to PAB
*
*** set up PAB for OPEN
*
BRW01  LI   TEMP1,$FUPD             opcode=0,mode=update
       CB   @MAINWS+15,@MAINWS+15   set mode=input (OP)?
       JOP  BRW02                   no
       LI   TEMP1,$FINP             yes...change mode=input
BRW02  MOV  TEMP1,@PABHD            put in PAB header
       MOV  @$DKBUF(U),@PABHD+2     VRAM buffer location to PAB header
       CLR  TEMP0
       MOV  TEMP0,@PABHD+6          set record#=0
       MOV  @BFPAB,TEMP0            VRAM destination
       LI   TEMP1,PABHD             RAM source
       LI   TEMP2,8                 copy first 8 bytes of PAB header
       BLWP @VMBW                   do the copy
```

```
*
*** open new blocks file [CODE = -14, USE; CODE = -16,CREATE]
*
       AI   TEMP0,9             address of filename's char count in PAB
       MOV  TEMP0,@SUBPTR       point to    " "
       BLWP @DSRLNK             open/create the file
       DATA 8
       JEQ  BKERR
       CI   TEMP7,4             READ or WRITE?
       JLT  BRW04               yes
       JGT  BRWDON              no; =USE; we're done
*
*** write blank records to newly created blocks file [CODE = -16,CREATE]
*
       MOV  *SP+,TEMP5          no; =CREATE; pop #blocks from stack
       SLA  TEMP5,3             convert #blocks to #records
       MOV  TEMP5,TEMP3         save
       MOV  TEMP5,TEMP4         set up counter
       LI   TEMP0,$FWRT+$FUPD   set up for WRITE
       MOV  TEMP0,@PABHD        copy to PAB header
BRLOOP S    TEMP4,TEMP5         calculate next record
       MOV  TEMP5,@PABHD+6      copy to PAB header
       MOV  @BFPAB,TEMP0        VRAM destination
       LI   TEMP1,PABHD         RAM source
       LI   TEMP2,8             #bytes of PAB header to copy to PAB
       BLWP @VMBW               do the copy
       AI   TEMP0,9             address of filename's char count
       MOV  TEMP0,@SUBPTR       point to filename's char count
       BLWP @DSRLNK             write one record of blanks
       DATA 8
       JEQ  BKERR
       MOV  TEMP3,TEMP5         get #blocks
       DEC  TEMP4               count down 1 record
       JNE  BRLOOP              write another record if not done
       JMP  BRWDON              we're done
*
*** prepare for read/write block
*
BRW04  MOV  *SP+,TEMP5          pop block# to write
       MOV  *SP+,TEMP6          pop bufaddr
       DEC  TEMP5               block#-1
       SLA  TEMP5,3             convert to starting record#
       LI   TEMP4,8             load counter for 8 records
       LI   TEMP0,$FWRT+$FUPD   set up for WRITE
       LI   TEMP3,VMBW          WRITE vector
       CI   TEMP7,2             are we writing the block?
       JEQ  BRW05               yup
       LI   TEMP0,$FRD+$FINP    nope...set up for READ
       LI   TEMP3,VMBR          READ vector
BRW05  MOV  TEMP0,@PABHD        copy opcode&mode to PAB header
*
* READ/WRITE block routine [CODE = -18/-20]
*
RWLOOP MOV  TEMP5,@PABHD+6      copy record# to PAB header
       MOV  @BFPAB,TEMP0        VRAM destination
       LI   TEMP1,PABHD         RAM source
       LI   TEMP2,8             #bytes of PAB header to copy to PAB
       BLWP @VMBW               do the copy
       MOV  @$DKBUF(U),TEMP0    VRAM buffer address to TEMP0
       MOV  TEMP6,TEMP1         RAM buffer to TEMP1
       LI   TEMP2,128           bytes to copy
       CI   TEMP7,3             READ?
       JEQ  BRW06               yup
       BLWP *TEMP3              nope...copy record to VRAM
*
* temporarily use CRU register---it should be OK
```

```
*
BRW06  MOV  @BFPAB,CRU          PAB address
       AI   CRU,9               address of filename's char count
       MOV  CRU,@SUBPTR         point to filename's char count
       BLWP @DSRLNK             read/write one record
       DATA 8
       JEQ  BKERR
       CI   TEMP7,2             WRITE?
       JEQ  BRW07               yup...next record
       BLWP *TEMP3              nope...copy record to RAM buffer
BRW07  INC  TEMP5               next record in file
       AI   TEMP6,128           next record to/from block RAM buffer
       DEC  TEMP4               count down 1 record
       JNE  RWLOOP              read/write another record if not done
       JMP  BRWDON              we're done
*
*** error handling
*
BKERR  MOVB TEMP0,TEMP0         device error?
       JEQ  BKERR6              yes, exit with disk error
BKERR9 LI   TEMP6,9             no, exit with file error
       JMP  BKCLN
BKERR8 LI   TEMP6,8             block# <=0!  exit with range error
       JMP  BKCLN
BKERR6 LI   TEMP6,6
BKCLN  BL   @BKCLOS             close current blocks file; ignore error
       CI   TEMP7,4             USE or CREATE?
       JLT  BKCLN1              no
       BL   @BPTOG                yes...toggle BPOFF & BFPAB
BKCLN1 MOV  TEMP6,TEMP0         pass error back to caller
       JMP  BKEXIT
BRWDON CLR  TEMP6
       BL   @BKCLOS             close current blocks file
       JNE  BRWDN1              error?
       LI   TEMP6,9             yes...assume it was a file error
BRWDN1 CI   TEMP7,4             (no error)...CREATE?
       JNE  BRWDN2              no...we're done
       BL   @BPTOG              yes...revert to correct blocks file
BRWDN2 MOV  TEMP6,TEMP0         error to TEMP0
BKEXIT MOV  @SVBRET,LINK        restore LINK
       B    @BKLINK

*   __  __              _   __          _       __   __
*  / / / /__ ___ ____  | | / /__ _____(_)__ _/ /  / /__
* / /_/ (_-</ -_) __/  | |/ / _ `/ __/ / _ `/ _ \/ / -_)
* \____/___/\__/_/     |___/\_,_/_/ /_/\_,_/_.__/_/\__/
*            ___      ___         ____
*           / _ \___ / _/__ ___ __/ / /____
*          / // / -_) _/ _ `/ // / / __(_-<
*         /____/\__/_/ \_,_/\_,_/_/\__/___/

*===========================================================
*
*++ initial values for first 30 user variables—COLD resets user variable table to these

UBASE0 BSS  6                BASE OF USER VARIABLES
       DATA UBASE0           06 USER UCONS
       DATA SPBASE           08 USER S0
       DATA RBASE            0A USER R0 { R0$
       DATA $UVAR            0C USER U0
       DATA SPBASE           0E USER TIB
       DATA 31               10 USER WIDTH
       DATA DPBASE           12 USER DP
       DATA $SYS$            14 USER SYS$
       DATA 0                16 USER CURPOS
       DATA INT1             18 USER INTLNK
       DATA 1                1A USER WARNING
       DATA 64               1C USER C/L$ { CL$
       DATA $BUFF            1E USER FIRST$
       DATA $LO              20 USER LIMIT$
       DATA >80              22 USER MGT pushes addr that is relative dist. from DISK_BUF
       DATA TLCTAB-MSGS+>80  24 USER LCT pushes addr that is relative dist. from DISK_BUF
       DATA 0                26 USER JMODE  loaded by graphics primitives; used by JOYST
       DATA 0                28 USER      <---available
       DATA DEFNAM-MSGS+>80  2A USER DBF pushes addr that is relative dist. from DISK_BUF
       DATA >1000            2C USER DISK_BUF  (BUFFER LOC IN VDP. SIZE=1K)  1K)
       DATA >460             2E USER PABS  (AREA FOR PABS ETC.)
       DATA 40               30 USER SCRN_WIDTH
       DATA 0                32 USER SCRN_START
       DATA 960              34 USER SCRN_END
       DATA INT1             36 USER ISR
       DATA 0                38 USER ALTIN
       DATA 0                3A USER ALTOUT
       DATA 1                3C USER VDPMDE    permanent location for VDPMDE
       DATA BPAB$-MSGS+>80   3E USER BPB pushes addr that is relative dist. from DISK_BUF
       DATA >0               40 USER BPOFF  offset into BPABS for cur. blocks file's PAB
*                                           ...always toggled between 0 and 70
*
$UVAR  BSS  >80              USER VARIABLE AREA

*    __  ___                            ______     __   __
*   /  |/  /__ ___ ___ ___ ____ ____   /_  __/__ _/ /  / /__
*  / /|_/ / -_|_-<(_-</ _ `/ _ `/ -_)   / / / _ `/ _ \/ / -_)
* /_/  /_/\__/___/___/\_,_/\_, /\__/   /_/  \_,_/_.__/_/\__/
*                         /___/
*                ____        __
*               /  _/__  ___/ /____ __
*              _/ // _ \/ _  / -_) \ /
*             /___/_//_/\_,_/\__/_\_\

*
* Message Table Index:  Offsets + 1 into VRAM Table of Messages
*   Offsets are incremented by 1 to allow 0 to indicate "no message".
*   First message indexed is 0, not 1, and does not really exist.
*
MTIDX  BYTE 0,MSG01-MSGS-1,MSG02-MSGS-2,0,MSG04-MSGS-4
       BYTE MSG05-MSGS-5,MSG06-MSGS-6,MSG07-MSGS-7
       BYTE MSG08-MSGS-8,MSG09-MSGS-9,MSG10-MSGS-10,0,0,0,0
       BYTE 0,0,MSG17-MSGS-17,MSG18-MSGS-18,MSG19-MSGS-19
       BYTE MSG20-MSGS-20,MSG21-MSGS-21,MSG22-MSGS-22,0,0
       BYTE MSG25-MSGS-25
*
*==================================================================
*
C100   DATA 100
H20    EVEN
H2000  DATA >2000
DECMAL TEXT '.'
HAA    BYTE >AA
       EVEN
*
* Utility Vectors
*
GPLLNK DATA UTILWS,GLENTR     Link to GROM routines
XMLLNK DATA UTILWS,XMLENT     Link to ROM routines
KSCAN  DATA UTILWS,KSENTR     Keyboard scan
VSBW   DATA UTILWS,VSBWEN     VDP single byte write
VMBW   DATA UTILWS,VMBWEN     VDP multiple byte write
VSBR   DATA UTILWS,VSBREN     VDP single byte read
VMBR   DATA UTILWS,VMBREN     VDP multiple byte read
VWTR   DATA UTILWS,VWTREN     VDP write to register
DSRLNK DATA DLNKWS,DLENTR     Link to device service routine
```

```
*
*===========================================================
* This is where ENTLNK used to be
*==========================================================
*      LINK TO SYSTEM XML UTILITIES
*
XMLENT MOV  *R14+,@GPLWS+2    Get argument
       LWPI GPLWS             Select GPL workspace
       MOV  R11,@UTILWS+22    Save GPL return address
       MOV  R1,R2             Make a copy of argument
       CI   R1,>8000          Direct address in ALC?
       JH   XML30             We have the address
       SRL  R1,12
       SLA  R1,1
       SLA  R2,4
       SRL  R2,11
       A    @XMLTAB(R1),R2
       MOV  *R2,R2
XML30  BL   *R2
       LWPI UTILWS            GET BACK TO RIGHT WS
       MOV  R11,@GPLWS+22     Restore GPL return address
       RTWP
*
*===========================================================
*** Link to GPL utilities
*
GLENTR MOVB @SUBSTK,R2        Fetch GPL subroutine stack ptr
       SRL  R2,8              Make it an index
       AI   R2,SCRPAD
       INCT R2
       MOV  @GRMSAV,R1        Push XML address for return
       MOVB R1,*R2
       SWPB R1
       MOVB R1,@1(R2)
       SWPB R2                Adjust stack pointer
       MOVB R2,@SUBSTK
       MOVB *R14+,@GRMWA      Set up address to call
       MOVB *R14+,@GRMWA      and second byte, adjusting return
       LWPI GPLWS
       MOV  @SVGPRT,R11
       RT                     Return to GPL
*
*** Return to assembly language from GPL
*
RTFGPL LWPI UTILWS       Select utility workspace
       RTWP              Return to calling AL routine
*
*==============================================================
*      KEYBOARD SCAN
*
KSENTR LWPI GPLWS
       MOV  R11,@UTILWS+22   Save GPL return address
       BL   @SCNKEY
       LWPI UTILWS
       MOV  R11,@GPLWS+22    Restore GPL return address
       RTWP
*
*==============================================================
*      VDP UTILITIES
*
** VDP single byte write
*
VSBWEN BL   @WVDPWA          Write out address
       MOVB @2(R13),@VDPWD   Write data
       RTWP                  Return to calling program
*
```

```
** VDP multiple byte write
*
VMBWEN BL   @WVDPWA          Write out address
VWTMOR MOVB *R1+,@VDPWD      Write a byte
       DEC  R2               Decrement byte count
       JNE  VWTMOR           More to write?
       RTWP                  Return to calling Program
*
** VDP single byte read
*
VSBREN BL   @WVDPRA          Write out address
       MOVB @VDPRD,@2(R13)   Read data
       RTWP                  Return to calling program
*
** VDP multiple byte read
*
VMBREN BL   @WVDPRA          Write out address
VRDMOR MOVB @VDPRD,*R1+      Read a byte
       DEC  R2               Decrement byte count
       JNE  VRDMOR           More to read?
       RTWP                  Return to calling program
*
** VDP write to register
*
VWTREN MOV  *R13,R1          Get register number and value
       MOVB @1(R13),@VDPWA   Write out value
       ORI  R1,>8000         Set for register write
       MOVB R1,@VDPWA        Write out register number
       RTWP                  Return to calling program
*
** Set up to write to VDP
*
WVDPWA LI   R1,>4000
       JMP  WVDPAD
*
** Set up to read VDP
*
WVDPRA CLR  R1
*
** Write VDP address
*
WVDPAD MOV  *R13,R2          Get VDP address
       MOVB @R2LB,@VDPWA     Write low byte of address
       SOC  R1,R2            Properly adjust VDP write bit
       MOVB R2,@VDPWA        Write high byte of address
       MOV  @2(R13),R1       Get CPU RAM address
       MOV  @4(R13),R2       Get byte count
       RT                    Return to calling routine
*
*===========================================================
*      CIF - Convert integer to floating                   *
*
CIF    LI   R4,FAC            Will convert into the FAC
       MOV  *R4,R0            Get integer into register
       MOV  R4,R6             Copy ptr to FAC to clear it
       CLR  *R6+              Clear FAC,FAC+1
       CLR  *R6+              IN CASE HAD A STRING IN FAC
       MOV  R0,R5             IS INTEGER EQUAL TO ZERO?
       JEQ  CIFRT             YES - ZERO RESULT AND RETURN
       ABS  R0                GET ABS VALUE OF ARG
       LI   R3,>40            GET EXPONENT BIAS
       CLR  *R6+              CLEAR WORDS IN RESULT THAT
       CLR  *R6                 MIGHT NOT GET A VALUE
       CI   R0,100            IS INTEGER < 100?
       JL   CIF02             YES-JUST PUT IN 1ST FRACTION
*                                PART
```

```
       CI   R0,10000          NO-IS ARG < 100,2?
       JL   CIF01             YES-JUST 1 DIVISION NECESSARY
*                             NO - 2 DIVISIONS ARE NECESSARY
       INC  R3                ADD 1 TO EXPONENT FOR 1ST DIV
       MOV  R0,R1             PUT # IN LOW ORDER WORD FOR
*                               THE DIVIDE
       CLR  R0                CLEAR HIGH ORDER WORD FOR THE
*                                DIVIDE
       DIV  @C100,R0          DIVIDE BY THE RADIX
       MOVB @R1LB,@3(R4)      MOVE THE RADIX DIGIT IN
CIF01
       INC  R3                ADD 1 TO EXPONENT FOR DIVIDE
       MOV  R0,R1             PUT IN LOW ORDER FOR DIVIDE
       CLR  R0                CLEAR HIGH ORDER FOR DIVIDE
       DIV  @C100,R0          DIVIDE BY THE RADIX
       MOVB @R1LB,@2(R4)      PUT NEXT RADIX DIGIT IN
CIF02
       MOVB @R0LB,@1(R4)      PUT HIGHEST ORDER RADIX DIGIT
*                             IN
       MOVB @R3LB,*R4         PUT EXPONENT IN
       INV  R5                IS RESULT POSITIVE?
       JLT  CIFRT             YES - SIGN IS CORRECT
       NEG  *R4               NO - MAKE IT NEGATIVE
CIFRT  RT
*
*============================================================
*** Link to device service routine
*
DLENTR MOV  *R14+,R5          Fetch program type for link
       SZCB @H20,R15          Reset equal bit
       MOV  @SCNAME,R0        Fetch pointer into PAB
       MOV  R0,R9             Save pointer
       AI   R9,-8             Adjust pointer to flag byte
       BLWP @VSBR             Read device name length
       MOVB R1,R3             Store it elsewhere
       SRL  R3,8              Make it a word value
       SETO R4                Initialize a counter
       LI   R2,NAMBUF         Point to NAMBUF
LNK$LP INC  R0                Point to next char of name
       INC  R4                Increment character counter
       C    R4,R3             End of name?
       JEQ  LNK$LN            Yes
       BLWP @VSBR             Read current character
       MOVB R1,*R2+           Move it to NAMBUF
       CB   R1,@DECMAL        Is it a decimal point?
       JNE  LNK$LP            No
LNK$LN MOV  R4,R4             Is name length zero?
       JEQ  LNKERR            Yes, error
       CI   R4,7              Is name length > 7?
       JGT  LNKERR            Yes, error
       CLR  @CRULST
       MOV  R4,@SCLEN-1       Store name length for search
       MOV  R4,@SAVLEN        Save device name length
       INC  R4                Adjust it
       A    R4,@SCNAME        Point to position after name
       MOV  @SCNAME,@SAVPAB   Save pointer into device name
*
*** Search ROM CROM GROM for DSR
*
SROM   LWPI GPLWS             Use GPL workspace to search
       CLR  R1                Version found of DSR etc.
       LI   R12,>0F00         Start over again
NOROM  MOV  R12,R12           Anything to turn off
       JEQ  NOOFF             No
       SBZ  0                 Yes, turn it off
NOOFF  AI   R12,>0100         Next ROM'S turn on
```

```
       CLR  @CRULST           Clear in case we're finished
       CI   R12,>2000         At the end
       JEQ  NODSR             No more ROMs to turn on
       MOV  R12,@CRULST       Save address of next CRU
       SBO  0                 Turn on ROM
       LI   R2,>4000          Start at beginning
       CB   *R2,@HAA          Is it a valid ROM?
       JNE  NOROM             No
       A    @TYPE$,R2          Go to first pointer
       JMP SGO2
SGO    MOV  @SADDR,R2         Continue where we left off
       SBO  0                 Turn ROM back on
SGO2   MOV  *R2,R2            Is address a zero
       JEQ  NOROM             Yes, no program to look at
       MOV  R2,@SADDR         Remember where we go next
       INCT R2                Go to entry point
       MOV  *R2+,R9           Get entry address
*
*** See if name matches
*
       MOVB @SCLEN,R5         Get length as counter
       JEQ  NAME2             Zero length, don't do match
       CB   R5,*R2+           Does length match?
       JNE  SGO               No
       SRL  R5,8              Move to right place
       LI   R6,NAMBUF         Point to NAMBUF
NAME1  CB   *R6+,*R2+         Is character correct?
       JNE  SGO               No
       DEC  R5                More to look at?
       JNE  NAME1             Yes
NAME2  INC  R1                Next version found
       MOV  R1,@SAVVER        Save version number
       MOV  R9,@SAVENT        Save entry address
       MOV  R12,@SAVCRU       Save CRU address
       BL   *R9               Match, call subroutine
       JMP  SGO               Not right version
       SBZ  0                 Turn off ROM
       LWPI DLNKWS            Select DSRLNK workspace
       MOV  R9,R0             Point to flag byte in PAB
       BLWP @VSBR             Read flag byte
       SRL  R1,13             Just want the error flags
       JNE  IOERR             Error!
       RTWP
*
*** Error handling
*
NODSR  LWPI DLNKWS       Select DSRLNK workspace
LNKERR CLR  R1           Clear the error flags
IOERR  SWPB R1
       MOVB R1,*R13      Store error flags in calling R0
       SOCB @H20,R15     Indicate an error occured
       RTWP              Return to caller
**
**
**
SVGPRT DATA 0            Save GPL return address
SAVCRU DATA 0            CRU address of peripheral
SAVENT DATA 0            Entry address of DSR
SAVLEN DATA 0            Save device name length
SAVPAB DATA 0            Ptr into device name in PAB
SAVVER DATA 0            Version number of DSR
NAMBUF DATA 0,0,0,0
*
*** General utility workspace registers (Overlaps next WS)
UTILWS DATA 0,0
       BYTE 0
```

```
R2LB   BYTE 0
*
*** DSR link routine workspace registers (Overlaps prev. WS)
DLNKWS DATA 0,0,0,0,0
TYPE$  DATA 0,0,0,0,0,0,0,0,0,0,0
*
*===========================================================

LINBUF BSS  80             BUFFER FOR SCROLLING
KEYCNT DATA -1             USED IN CURSOR FLASH LOGIC
CURCHR BSS  2              CHAR AT CURSOR POSITION
GRMSAV BSS  2              SAVE GROM ADDRESS DURING DSRLNK
INTACT DATA 0              NON-ZERO DURING INTERRUPT SERVICE

*===========================================================
*
```

N.3 *fbForth_dictionary.a99*

The file fbForth_dictionary.a99 contains the resident portion of the **fbForth** dictionary, which includes the routine that cold-starts **fbForth** (label at FF9900).

```
*    _____   ____         __  __     ___ ___
*   / _/ /  / __/__  ____/ /_/ /    <  // _ \
*  / _/ _ \/ _// _ \/ __/ __/ _ \   / // // /
* /_//_.__/_/  \___/_/  \__/_//_/  /_(_)___/
*
*              ____ __        __                   __
*             / _(_) /__ ____/ /  ___ ____ ___ ___/ /
*      _ _ _ / _/ / / -_)___/ _ \/ _ `(_-</ -_) _  /
*     (_|_|_)_//_/_/\__/   /_.__/\_,_/___/\__/\_,_/
*
*             __________ ____         __  __
*            /_  __/  _/ / __/__  ____/ /_/ /
*             / / _/ /  / _// _ \/ __/ __/ _ \
*            /_/ /___/ /_/  \___/_/  \__/_//_/

* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*                                                                   *
* fbForth --- a file-based TI Forth                                 *
* (C) Lee Stewart 2013                                              *
* ---written in TMS9900 Assembler for the TI-99/4A and based on     *
*    the the original public domain code written by Leon Tietz,     *
*    Leslie O'Hagan and Edward E. Ferguson                          *
*                                                                   *
* Filename: fbForth_dictionary.a99                                  *
*                                                                   *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

*    ___         _    __        __
*   / _ \___ ___ (_)__/ /__ ___  / /_
*  / , _/ -_|_-</ / _  / -_) _ \/ __/
* /_/|_|\__/___/_/\_,_/\__/_//_/\__/
*
*            ___ _     __  _
*           / _ \(_)___/ /_(_)__  ___  ___ _______ __
*          / // / / __/ __/ / _ \/ _ \/ _ `/ __/ // /
*         /____/_/\__/\__/_/\___/_//_/\_,_/_/  \_, /
*                                             /___/

* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*                                                                           *
*   This file contains the code for fbForth's resident dictionary.  It      *
*   is basically TI Forth with block (TI Forth's 'screen') I/O converted    *
*   to using DF128 files instead of direct disk-sector I/O.  Some enhance-  *
*   ments have been made; but, words written for TI Forth should work with  *
*   fbForth after accommodating the block I/O changes.                      *
*                                                                           *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*
*
       AORG >A000
FF9900 LI   IP,COLD+2
       MOV  @$U0(TEMP1),U
       MOV  TEMP1,@$UCONS(U)
       MOV  @$UCONS(U),TEMP1
       MOV  @$S0(TEMP1),SP
       MOV  @$R0(TEMP1),R
       MOV  @$U0(TEMP1),U
       MOV  U,@$U0(U)
       LI   NEXT,$NEXT
       B    *NEXT
*
*
*** EXECUTE ***
       DATA >0
L1000  DATA >8745,>5845,>4355,>54C5
EXECUT DATA $+2
       MOV  *SP+,W
       B    @DOEXEC
*
*
*** LIT ***
       DATA L1000
L1001  DATA >834C,>49D4
LIT    DATA $+2
       DECT SP
       MOV  *IP+,*SP
       B    *NEXT
*
*
*** BRANCH ***
       DATA L1001
L1002  DATA >8642,>5241,>4E43,>48A0
BRANCH DATA $+2
BRAN2  A    *IP,IP
       B    *NEXT
*
*
*** 0BRANCH ***
       DATA L1002
L1003  DATA >8730,>4252,>414E,>43C8
ZBRAN  DATA $+2
       MOV  *SP+,TEMP1
       JEQ  ZBRAN1
       INCT IP
       B    *NEXT
ZBRAN1 A    *IP,IP
       B    *NEXT
*
*
*** (OF) ***
       DATA L1003
L1004  DATA >8428,>4F46,>29A0
POF    DATA $+2
       C    *SP+,*SP
       JNE  POF1
       INCT SP
       INCT IP
       B    *NEXT
POF1   A    *IP,IP
       B    *NEXT
*
*
*** (LOOP) ***
       DATA L1004
L1005  DATA >8628,>4C4F,>4F50,>29A0
PLOOP  DATA $+2
       INC  *R
       C    *R,@2(R)
       JLT  PLOOPA
       AI   R,4
       INCT IP
       B    *NEXT
PLOOPA A    *IP,IP
       B    *NEXT
*
```

```
*
*** (+LOOP) ***
       DATA L1005
L1006  DATA >8728,>2B4C,>4F4F,>50A9
PPLOOP DATA $+2
       MOV  *SP+,TEMP1
       A    TEMP1,*R
       MOV  TEMP1,TEMP1
       JLT  PLOOP2
PLOOP1 C    *R,@2(R)
       JLT  PLOOP3
       AI   R,4
       INCT IP
       B    *NEXT
PLOOP2 C    *R,@2(R)
       JGT  PLOOP3
       AI   R,4
       INCT IP
       B    *NEXT
PLOOP3 A    *IP,IP
       B    *NEXT
*
*
*** (DO) ***
       DATA L1006
L1007  DATA >8428,>444F,>29A0
PDO    DATA $+2
       AI   R,-4
       MOV  *SP+,*R
       MOV  *SP+,@2(R)
       B    *NEXT
*
*
*** I ***
       DATA L1007
L1008  DATA >81C9
I      DATA $+2
       DECT SP
       MOV  *R,*SP
       B    *NEXT
*
*
*** J ***
       DATA L1008
J1008  DATA >81CA
J      DATA $+2
       DECT SP
       MOV  @4(R),*SP
       B    *NEXT
*
*
*** DIGIT ***
       DATA J1008
L1009  DATA >8544,>4947,>49D4
DIGIT  DATA $+2
       MOV  *SP+,TEMP1
       MOV  *SP,TEMP2
       AI   TEMP2,->0030
       CI   TEMP2,10
       JL   DIGIT1
       AI   TEMP2,-7
       CI   TEMP2,10
       JHE  DIGIT1
DIGIT2 CLR  *SP
       B    *NEXT
DIGIT1 C    TEMP2,TEMP1
```

```
       JHE  DIGIT2
       MOV  TEMP2,*SP
       DECT  SP
       SETO *SP
       NEG  *SP
       B    *NEXT
*
*
*** (FIND) ***
       DATA L1009
L100A  DATA >8628,>4649,>4E44,>29A0
PFIND  DATA $+2
       MOV  *SP,TEMP1
       JEQ  PFIND4
PFIND1 MOV  TEMP1,TEMP2
       MOV  @2(SP),TEMP3
       MOVB *TEMP2+,W
       ANDI W,>3F00
       CB   W,*TEMP3+
       JNE  PFIND3
PFIND2 MOVB *TEMP2+,W
       JLT  PFIND5
       CB   W,*TEMP3+
       JEQ  PFIND2
PFIND3 MOV  @-2(TEMP1),TEMP1
       JNE  PFIND1
PFIND4 INCT SP
       CLR  *SP
       B    *NEXT
PFIND5 ANDI W,>7F00
       CB   W,*TEMP3
       JNE  PFIND3
       INCT TEMP2
       MOV  TEMP2,@2(SP)
       CLR  *SP
       MOVB *TEMP1,@1(SP)
       DECT SP
       SETO *SP
       NEG  *SP
       B    *NEXT
*
*
*** ENCLOSE ***
       DATA L100A
L100B  DATA >8745,>4E43,>4C4F,>53C5
ENCLOS DATA $+2
       MOV  *SP+,TEMP1
       MOV  *SP,TEMP2
       SWPB TEMP1
       SETO TEMP3
ENCL1  INC  TEMP3
       CB   TEMP1,*TEMP2+
       JEQ  ENCL1
       DEC  TEMP2
       AI   SP,-6
       MOV  TEMP3,@4(SP)
       MOV  TEMP3,*SP
       INC  TEMP3
       MOV  TEMP3,@2(SP)
       MOVB *TEMP2,W
       JNE  ENCL4
       B    *NEXT
ENCL4  INC  TEMP2
ENCL2  MOV  TEMP3,@2(SP)
       MOVB *TEMP2,W
       JEQ  ENCL3
       INC  TEMP3
       CB   TEMP1,*TEMP2+
       JNE  ENCL2
ENCL3  MOV  TEMP3,*SP
       B    *NEXT
*
*
*** kEY ***
       DATA L100B
L100C  DATA >836B,>45D9
KE     DATA $+2
       LI   TEMP1,-2
       MOV  @$SYS(U),LINK
       BL   *LINK
       DECT SP
       MOV  TEMP0,*SP
       B    *NEXT
*
*
*** KEY ***
       DATA L100C
L100CX DATA >834B,>45D9
KEY    DATA DOCOL,KE,LIT,>7F,_AND,SEMIS
*
*
*** KEY8 ***
       DATA L100CX
L100CY DATA >844B,>4559,>38A0
KEY8   DATA DOCOL,KE,SEMIS
*
*
*** EMIT ***
       DATA L100CY
L100D  DATA >8445,>4D49,>54A0
EMIT   DATA $+2
       MOV  *SP+,TEMP2
       ANDI TEMP2,>007F
       LI   TEMP1,-4
       MOV  @$SYS(U),LINK
       BL   *LINK
       INC  @$OUT(U)
       B    *NEXT
*
*
*** EMIT8 ***
       DATA L100D
L100DX DATA >8545,>4D49,>54B8
EMIT8  DATA $+2
       MOV  *SP+,TEMP2
       ANDI TEMP2,>00FF
       LI   TEMP1,-4
       MOV  @$SYS(U),LINK
       BL   *LINK
       INC  @$OUT(U)
       B    *NEXT
*
*
*** CR ***
       DATA L100DX
L100E  DATA >8243,>52A0
CR     DATA $+2
       LI   TEMP1,-6
       MOV  @$SYS(U),LINK
       BL   *LINK
       B    *NEXT
*
```

```
*
*** ?TERMINAL ***
       DATA L100E
L100F  DATA >893F,>5445,>524D,>494E,>41CC
QTERM  DATA $+2
       LI   TEMP1,-8
       MOV  @$SYS(U),LINK
       BL   *LINK
       DECT SP
       MOV  TEMP0,*SP
       B    *NEXT
*
*
*** ?KEY ***
       DATA L100F
L1010  DATA >843F,>4B45,>59A0
QKEY   DATA $+2
       LI   TEMP1,-10
       MOV  @$SYS(U),LINK
       BL   *LINK
       ANDI TEMP0,>007F
       DECT SP
       MOV  TEMP0,*SP
       B    *NEXT
*
*
*** ?KEY8 ***
       DATA L1010
L1010X DATA >853F,>4B45,>59B8
QKEY8  DATA $+2
       LI   TEMP1,-10
       MOV  @$SYS(U),LINK
       BL   *LINK
       ANDI TEMP0,>00FF
       DECT SP
       MOV  TEMP0,*SP
       B    *NEXT
*
*
*** GOTOXY ***
       DATA L1010X
L1011  DATA >8647,>4F54,>4F58,>59A0
GOTOXY DATA $+2
       MOV  *SP+,TEMP3
       MOV  *SP+,TEMP2
       LI   TEMP1,-12
       MOV  @$SYS(U),LINK
       BL   *LINK
       B    *NEXT
*
*
*** BLKRW ***   blocks I/O utility routine called by DO_BRW below
*       ( {[bfnaddr]|[#blocks bfnaddr]|[bufaddr block#]} opcode --- flag )
*       ...# items required on stack depends on opcode as follows:
*               ( bfnaddr -14 --- )
*       ( #blocks bfnaddr -16 --- )
*       (  bufaddr block# -18 --- )
*       (  bufaddr block# -20 --- )
*           ...note that 2 or 3 items may be required on the stack,
*           depending on the opcode
*
       DATA L1011
BF000  DATA >8542,>4C4B,>52D7
BLKRW  DATA $+2
       MOV  *SP+,TEMP1      pop opcode to R1 for system call
       MOV  @$SYS(U),LINK   get system support address to R11
       BL   *LINK           call system support
*   all stack values into this routine have now been popped
       DECT SP              make room on stack for error return
       MOV  TEMP0,*SP       put error return on stack
       B    *NEXT
*
*
*** DO_BRW ***  helper routine that executes BLKRW and processes returned flag
*       ( {[bfnaddr]|[#blocks bfnaddr]|[bufaddr block#]} opcode --- )
*       ...# items required on stack depends on opcode as follows:
*               ( bfnaddr -14 --- )
*       ( #blocks bfnaddr -16 --- )
*       (  bufaddr block# -18 --- )
*       (  bufaddr block# -20 --- )
*
       DATA BF000
BF001  DATA >8644,>4F5F,>4252,>57A0
DOBRW  DATA DOCOL,BLKRW     call blocks I/O utility routine
       DATA DUP,QERROR      deal with any error
       DATA SEMIS
*
*
*** WBLK ***    write a block to blocks file
*           ( bufaddr block# --- )
       DATA BF001
BF002  DATA >8457,>424C,>4BA0
WBLK   DATA DOCOL,LIT,-20   "write block" opcode to stack
       DATA DOBRW,SEMIS     write block and handle any error
*
*
*** RBLK ***    read a block from blocks file
*           ( bufaddr block# --- )
       DATA BF002
BF003  DATA >8452,>424C,>4BA0
RBLK   DATA DOCOL,LIT,-18   "read block" opcode to stack
       DATA DOBRW,SEMIS     read block and handle any error
*
*
*** BFLNAM ***  helper routine that gets blocks filename into PAD|HERE and
*               passes name pointer if flag is true [command line], but passes
*               nothing if flag is false [compiled by SLIT] )
*           ( flag --- [bfnaddr] | [] )
       DATA BF003
BF004  DATA >8642,>464C,>4E41,>4DA0
BFNAM  DATA DOCOL,ZBRAN,BF0041-$
       DATA PAD,HERE,SUB,DUP,ALLOT  temporarily put HERE at PAD, saving distance
       DATA MINUS       negate distance for restoring HERE
       DATA BL,WORD     get the string to HERE [old PAD]
       DATA ALLOT       restore HERE
       DATA PAD,BRANCH,BF0042-$
BF0041 DATA BL,WORD             get the string to HERE
       DATA HERE,CAT            retrieve block-file-pathname length
       DATA ONEP,ECELLS,ALLOT   move HERE past block file pathname
*                               on an even-word boundary
BF0042 DATA SEMIS
*
*** DEFBF ***  get default blocks filename to PAD and leave PAD address
*       ( --- bfnaddr )
       DATA BF004
BF005  DATA >8544,>4546,>42C6
DEFBF  DATA DOCOL,DBF,DUP,_VSBR,ONEP,PAD,SWAP,_VMBR,PAD,SEMIS
*
*
*** MKBFL ***   Create a blocks file from string and number of blocks in
*               input stream
*       usage:  MKBFL DSK1.BLOCKS 80
*           this routine uses PAD for temporary storage of pathname of file until
*           successful return; hopefully, nothing tramples it while we're gone!?!
*           ( --- )
       DATA BF005
BF006  DATA >854D,>4B42,>46CC
MKBF   DATA DOCOL,ONE,BFNAM             process filename from input stream
*                                       and get bfnaddr
       DATA BL,WORD,HERE,NUMBER,DROP    get # of blocks from input stream
       DATA SWAP                        change stack order to [#blocks bfnaddr]
       DATA DKBUF,AT,LIT,128,BL,VFILL   fill Forth disk buffer with blanks
       DATA LIT,-16                     "create file" opcode to stack
       DATA DOBRW,SEMIS                 create file and handle any error
*
*
*** TLC ***  Load true lowercase and zero patch to vaddr from storage in VRAM
*            ( vaddr --- )
       DATA BF006
BF006G DATA >8354,>4CC3
TLC    DATA DOCOL
       DATA BPB$,LIT,4,SUB,OVER,LIT,>17D,SUB,THREE,VMOVE  Patch zero pattern
       DATA TLC$,SWAP,LIT,248,VMOVE,SEMIS       load true lowercase
*
*
*** (UB) ***    Runtime routine for USEBFL that changes current blocks file
*               to file pointed to by bfnaddr
*           ( bfnaddr --- )
*
       DATA BF006G
BF006J DATA >8428,>5542,>29A0
PUB    DATA DOCOL,LIT,-14           "use file" opcode to stack
       DATA DOBRW,SEMIS             set up new blocks file and handle any error
*
*
*** USEBFL ***  [ IMMEDIATE word ]
*           Select a different blocks file from input stream
*           This routine uses PAD|HERE for temporary pathname storage
*           until successful return;
*           Hopefully, nothing tramples it while we're gone!?!
*       usage:  USEBFL DSK1.BLOCKS
*
       DATA BF006J
BF007  DATA >C655,>5345,>4246,>4CA0
USEBF  DATA DOCOL,STATE,AT,ZBRAN,BF0071-$
       DATA COMPIL,EMPTYB       at execution, unconditionally abandon any dirty blocks
       DATA COMPIL,SLIT         at execution, will put address of blocks filename on
*                               ...stack and step over it
       DATA ZERO,BFNAM          get block filename to HERE but do not return address;
*                               ...SLIT will provide it
       DATA COMPIL,PUB,BRANCH,BF0072-$  make current the blocks file parsed by BFLNAM
BF0071 DATA EMPTYB              unconditionally abandon any dirty blocks
       DATA ONE,BFNAM           process filename from input stream and get pointer to it
       DATA PUB                 make current the blocks file parsed by BFLNAM
BF0072 DATA SEMIS
*
*** CMOVE ***   move cnt bytes from src RAM to dst RAM
*       ( src dst cnt --- )
*
       DATA BF007
L1015  DATA >8543,>4D4F,>56C5
CMOVE  DATA $+2
       MOV  *SP+,TEMP1
       MOV  *SP+,TEMP2
       MOV  *SP+,TEMP3
       MOV  TEMP1,TEMP1
       JEQ  CMOVE2
CMOVE1 MOVB *TEMP3+,*TEMP2+
       DEC  TEMP1
       JNE  CMOVE1
CMOVE2 B    *NEXT
*
*** MOVE ***   move cnt cells from src RAM to dst RAM
*       ( src dst cnt --- )
*
       DATA L1015
A1000  DATA >844D,>4F56,>45A0
MOVE   DATA $+2
       MOV  *SP+,TEMP1
       MOV  *SP+,TEMP2
       MOV  *SP+,TEMP3
       MOV  TEMP1,TEMP1
       JEQ  MOVE2
MOVE1  MOV  *TEMP3+,*TEMP2+
       DEC  TEMP1
       JNE  MOVE1
MOVE2  B    *NEXT
*
*** SWPB ***
       DATA A1000
A1001  DATA >8453,>5750,>42A0
SWPB   DATA $+2
       SWPB *SP
       B    *NEXT
*
*
*** SRL ***
       DATA A1001
A1002  DATA >8353,>52CC
SRL    DATA $+2
       MOV  *SP+,TEMP0
       MOV  *SP,TEMP1
       SRL  TEMP1,0
       MOV  TEMP1,*SP
       B    *NEXT
*
*
*** SLA ***
       DATA A1002
A1003  DATA >8353,>4CC1
SLA    DATA $+2
       MOV  *SP+,TEMP0
       MOV  *SP,TEMP1
       SLA  TEMP1,0
       MOV  TEMP1,*SP
       B    *NEXT
*
*
*** SRA ***
       DATA A1003
A1004  DATA >8353,>52C1
SRA    DATA $+2
       MOV  *SP+,TEMP0
       MOV  *SP,TEMP1
       SRA  TEMP1,0
       MOV  TEMP1,*SP
       B    *NEXT
*
*
*** SRC ***
       DATA A1004
A1005  DATA >8353,>52C3
SRC    DATA $+2
       MOV  *SP+,TEMP0
       MOV  *SP,TEMP1
       SRC  TEMP1,0
       MOV  TEMP1,*SP
       B    *NEXT
*
*
*** U* ***
       DATA A1005
L1016  DATA >8255,>2AA0
MULT   DATA $+2
       MOV  *SP+,TEMP2
       MPY  *SP,TEMP2
       MOV  TEMP3,*SP
       DECT SP
       MOV  TEMP2,*SP
       B    *NEXT
*
*
*** U/ ***
       DATA L1016
L1017  DATA >8255,>2FA0
DIV    DATA $+2
       MOV  @2(SP),TEMP2
       MOV  @4(SP),TEMP3
       DIV  *SP+,TEMP2
       MOV  TEMP2,*SP
       MOV  TEMP3,@2(SP)
       B    *NEXT
*
*
*** AND ***
       DATA L1017
L1018  DATA >8341,>4EC4
_AND   DATA $+2
       INV  *SP
       SZC  *SP+,*SP
       B    *NEXT
*
*
*** OR ***
       DATA L1018
L1019  DATA >824F,>52A0
_OR    DATA $+2
       SOC  *SP+,*SP
       B    *NEXT
*
*
*** XOR ***
       DATA L1019
L101A  DATA >8358,>4FD2
_XOR   DATA $+2
       MOV  *SP+,TEMP1
       XOR  *SP,TEMP1
       MOV  TEMP1,*SP
       B    *NEXT
*
*
*** SP@ ***
       DATA L101A
L101B  DATA >8353,>50C0
SPAT   DATA $+2
       DECT SP
       MOV  SP,*SP
       INCT *SP
       B    *NEXT
*
```

```
*
*** SP! ***
       DATA L101B
L101C  DATA >8353,>50A1
SPSTOR DATA $+2
       MOV  @$S0(U),SP
       B    *NEXT
*
*
*** RP! ***
       DATA L101C
L101D  DATA >8352,>50A1
RSTOR  DATA $+2
       MOV  @$R0(U),R
       B    *NEXT
*
*
*** ;S ***
       DATA L101D
L101E  DATA >823B,>53A0
SEMIS  DATA $SEMIS
*
*
*** LEAVE ***
       DATA L101E
L101F  DATA >854C,>4541,>56C5
LEAVE  DATA $+2
       MOV  *R,@2(R)
       B    *NEXT
*
*
*** >R ***
       DATA L101F
L1020  DATA >823E,>52A0
TOR    DATA $+2
       DECT R
       MOV  *SP+,*R
       B    *NEXT
*
*
*** R> ***
       DATA L1020
L1021  DATA >8252,>3EA0
FROMR  DATA $+2
       DECT SP
       MOV  *R+,*SP
       B    *NEXT
*
*
*** R ***
       DATA L1021
L1022  DATA >81D2
RR     DATA $+2
       DECT SP
       MOV  *R,*SP
       B    *NEXT
*
*
*** U ***
       DATA L1022
L1023  DATA >81D5
UU     DATA $+2
       DECT SP
       MOV  U,*SP
       B    *NEXT
*
```

```
*
*** 0= ***
       DATA L1023
L1024  DATA >8230,>3DA0
ZEQU   DATA $+2
       MOV  *SP,TEMP1
       JEQ  ZEQUTR
       CLR  *SP
       B    *NEXT
ZEQUTR SETO *SP
       NEG  *SP
       B    *NEXT
*
*
*** 0< ***
       DATA L1024
L1025  DATA >8230,>3CA0
ZLESS  DATA $+2
       MOV  *SP,TEMP1
       JLT  PUSHTR
PUSHFL CLR  *SP
       B    *NEXT
PUSHTR SETO *SP
       NEG  *SP
       B    *NEXT
*
*
*** + ***
       DATA L1025
L1026  DATA >81AB
PLUS   DATA $+2
       A    *SP+,*SP
       B    *NEXT
*
*
*** D+ ***
       DATA L1026
L1027  DATA >8244,>2BA0
DPLUS  DATA $+2
       A    *SP+,@2(SP)
       A    *SP+,@2(SP)
       JNC  DPLUS1
       INC  *SP
DPLUS1 B    *NEXT
*
*
*** MINUS ***
       DATA L1027
L1028  DATA >854D,>494E,>55D3
MINUS  DATA $+2
       NEG  *SP
       B    *NEXT
*
*
*** DMINUS ***
       DATA L1028
L1029  DATA >8644,>4D49,>4E55,>53A0
DMINUS DATA $+2
       INV  @2(SP)
       INV  *SP
       INC  @2(SP)
       JNC  DM1
       INC  *SP
DM1    B    *NEXT
*
*
```

```
*** OVER ***
       DATA L1029
L102A  DATA >844F,>5645,>52A0
OVER   DATA $+2
       DECT SP
       MOV  @4(SP),*SP
       B    *NEXT
*
*
*** DROP ***
       DATA L102A
L102B  DATA >8444,>524F,>50A0
DROP   DATA $+2
       INCT SP
       B    *NEXT
*
*
*** SWAP ***
       DATA L102B
L102C  DATA >8453,>5741,>50A0
SWAP   DATA $+2
       MOV  *SP,TEMP1
       MOV  @2(SP),*SP
       MOV  TEMP1,@2(SP)
       B    *NEXT
*
*
*** DUP ***
       DATA L102C
L102D  DATA >8344,>55D0
DUP    DATA $+2
       DECT SP
       MOV  @2(SP),*SP
       B    *NEXT
*
*
*** +! ***
       DATA L102D
L102E  DATA >822B,>21A0
PSTORE DATA $+2
       MOV  *SP+,TEMP1
       A    *SP+,*TEMP1
       B    *NEXT
*
*
*** TOGGLE ***
       DATA L102E
L102F  DATA >8654,>4F47,>474C,>45A0
TOGGLE DATA $+2
       MOV  *SP+,TEMP1
       MOV  *SP+,TEMP2
       MOVB *TEMP2,TEMP3
       SWPB TEMP1
       XOR  TEMP1,TEMP3
       MOVB TEMP3,*TEMP2
       B    *NEXT
*
*
*** @ ***
       DATA L102F
L1030  DATA >81C0
AT     DATA $+2
       MOV  *SP,TEMP1
       MOV  *TEMP1,*SP
       B    *NEXT
*
```

```
*
*** C@ ***
       DATA L1030
L1031  DATA >8243,>40A0
CAT    DATA $+2
       MOV  *SP,TEMP1
       MOVB *TEMP1,TEMP1
       SRL  TEMP1,8
       MOV  TEMP1,*SP
       B    *NEXT
*
*
*** ! ***
       DATA L1031
L1032  DATA >81A1
STORE  DATA $+2
       MOV  *SP+,TEMP1
       MOV  *SP+,*TEMP1
       B    *NEXT
*
*
*** C! ***
       DATA L1032
L1033  DATA >8243,>21A0
CSTORE DATA $+2
       MOV  *SP+,TEMP1
       MOVB @1(SP),*TEMP1
       INCT SP
       B    *NEXT
*
*
*** 1+ ***
       DATA L1033
L1034  DATA >8231,>2BA0
ONEP   DATA $+2
       INC  *SP
       B    *NEXT
*
*
*** 2+ ***
       DATA L1034
L1035  DATA >8232,>2BA0
TWOP   DATA $+2
       INCT *SP
       B    *NEXT
*
*
*** 1- ***
       DATA L1035
L1035A DATA >8231,>2DA0
ONEM   DATA $+2
       DEC  *SP
       B    *NEXT
*
*
*** 2- ***
       DATA L1035A
L1035B DATA >8232,>2DA0
TWOM   DATA $+2
       DECT *SP
       B    *NEXT
*
*
*** - ***
       DATA L1035B
L1036  DATA >81AD
```

```
SUB    DATA $+2
       S    *SP+,*SP
       B    *NEXT
*
*
*** =CELLS ***
       DATA L1036
L1037  DATA >863D,>4345,>4C4C,>53A0
ECELLS DATA $+2
       MOV  *SP,TEMP1
       INC  TEMP1
       ANDI TEMP1,>FFFE
       MOV  TEMP1,*SP
       B    *NEXT
*
*
*** S->D ***
       DATA L1037
L1038  DATA >8453,>2D3E,>44A0
STOD   DATA $+2
       SETO TEMP1
       MOV  *SP,TEMP2
       JLT STOD1
       CLR  TEMP1
STOD1  DECT SP
       MOV  TEMP1,*SP
       B    *NEXT
*
*
*** ABS ***
       DATA L1038
L1039  DATA >8341,>42D3
ABS    DATA $+2
       ABS  *SP
       B    *NEXT
*
*
*** MIN ***
       DATA L1039
L103A  DATA >834D,>49CE
MIN    DATA $+2
       C    @2(SP),*SP
       JLT  MIN1
       MOV  *SP,@2(SP)
MIN1   INCT SP
       B    *NEXT
*
*
*** MAX ***
       DATA L103A
L103B  DATA >834D,>41D8
MAX    DATA $+2
       C    *SP,@2(SP)
       JLT  MAX1
       MOV  *SP,@2(SP)
MAX1   INCT SP
       B    *NEXT
*
*
*** U< ***
       DATA L103B
L103C  DATA >8255,>3CA0
ULESS  DATA $+2
       MOV  *SP+,TEMP2
       MOV  *SP,TEMP1
       CLR  *SP
       C    TEMP1,TEMP2
       JHE  ULESS1
       INC  *SP
ULESS1 B    *NEXT
*
*
*** 0 ***
       DATA L103C
L103F  DATA >81B0
ZERO   DATA DOCON,>0
*
*** 1 ***
       DATA L103F
L1040  DATA >81B1
ONE    DATA DOCON,>1
*
*** 2 ***
       DATA L1040
L1041  DATA >81B2
TWO    DATA DOCON,>2
*
*** 3 ***
       DATA L1041
L1042  DATA >81B3
THREE  DATA DOCON,>3
*
*** BL ***
       DATA L1042
L1043  DATA >8242,>4CA0
BL     DATA DOCON,>20
*
*** DKB+ ***    Defining word to create words that calculate addresses
*               from user variables containing offsets from Forth's disk
*               buffer. Execution of the defined word pushes to the stack
*               an address calculated by adding the disk buffer address
*               to the offset passed in the user variable, whose user-
*               variable-table offset is the parameter field value.
*
*   USAGE:  userVarOffset DKB+ <new word>
*
       DATA L1043
L1043A DATA >8444,>4B42,>2BA0
DKBP   DATA DOCOL,BUILDS,COMMA
DODKBP EQU  $+2
       DATA PDOES
       DATA >6A0,DODOES     same as ' BL @DODOES '
       DATA AT,UU,PLUS,AT   get offset in user variable
       DATA DKBUF,AT        get Forth's disk buffer address
       DATA PLUS,SEMIS      add to get new address to leave on stack
*
*** UCONS$ ***
       DATA L1043A
L1044  DATA >8655,>434F,>4E53,>24A0
UCONS$ DATA DOUSER,>6
*
*** S0 ***
       DATA L1044
L1045  DATA >8253,>30A0
S0     DATA DOUSER,>8
*
*** R0 ***
       DATA L1045
L1046  DATA >8252,>30A0
RR0    DATA DOUSER,>A
*
*** U0 ***
```

```
       DATA L1046
L1047  DATA >8255,>30A0
U0     DATA DOUSER,>C
*
*** TIB ***
       DATA L1047
L1048  DATA >8354,>49C2
TIB    DATA DOUSER,>E
*
*** WIDTH ***
       DATA L1048
L1049  DATA >8557,>4944,>54C8
WIDTH  DATA DOUSER,>10
*
*** DP ***
       DATA L1049
L104A  DATA >8244,>50A0
DP     DATA DOUSER,>12
*
*** SYS$ ***
       DATA L104A
L104B  DATA >8453,>5953,>24A0
SYS$   DATA DOUSER,>14
*
*** CURPOS ***
       DATA L104B
L104C  DATA >8643,>5552,>504F,>53A0
TERM$  DATA DOUSER,>16
*
*** INTLNK ***
       DATA L104C
L104D  DATA >8649,>4E54,>4C4E,>4BA0
DISK$  DATA DOUSER,>18
*
*** WARNING ***
       DATA L104D
L104E  DATA >8757,>4152,>4E49,>4EC7
WARNIN DATA DOUSER,>1A
*
*** C/L$ ***
       DATA L104E
L104F  DATA >8443,>2F4C,>24A0
CL$    DATA DOUSER,>1C
*
*** FIRST$ ***
       DATA L104F
L1050  DATA >8646,>4952,>5354,>24A0
FIRST$ DATA DOUSER,>1E
*
*** LIMIT$ ***
       DATA L1050
L1051  DATA >864C,>494D,>4954,>24A0
LIMIT$ DATA DOUSER,>20
*
*** MGT ***
       DATA L1051
L1052  DATA >834D,>47D4
SYSM$  DATA DODKBP,>22
*
*** LCT ***
       DATA L1052
L1053  DATA >834C,>43D4
TLC$   DATA DODKBP,>24
*
*
*** DBF ***
```

```
       DATA L1053
BF00A  DATA >8344,>42C6
DBF    DATA DODKBP,>2A
**
*** DISK_BUF ***
       DATA BF00A
BF00B  DATA >8844,>4953,>4B5F,>4255,>46A0
DKBUF  DATA DOUSER,>2C
*
*** PABS ***
       DATA BF00B
X0005  DATA >8450,>4142,>53A0
PABS   DATA DOUSER,>2E
*
*** SCRN_WIDTH ***
       DATA X0005
X0006  DATA >8A53,>4352,>4E5F,>5749,>4454,>48A0
       DATA DOUSER,>30
*
*** SCRN_START ***
       DATA X0006
X0007  DATA >8A53,>4352,>4E5F,>5354,>4152,>54A0
       DATA DOUSER,>32
*
*** SCRN_END ***
       DATA X0007
X0008  DATA >8853,>4352,>4E5F,>454E,>44A0
       DATA DOUSER,>34
*
*** ISR ***
       DATA X0008
X0009  DATA >8349,>53D2
       DATA DOUSER,>36
*
*** ALTIN ***
       DATA X0009
X000A  DATA >8541,>4C54,>49CE
       DATA DOUSER,>38
*
*** ALTOUT ***
       DATA X000A
X000B  DATA >8641,>4C54,>4F55,>54A0
       DATA DOUSER,>3A
*
*** VDPMDE ***
       DATA X000B
X000C  DATA >8656,>4450,>4D44,>45A0
VDPM$  DATA DOUSER,>3C
*
*** BPB ***
       DATA X000C
X000D  DATA >8342,>50C2
BPB$   DATA DODKBP,>3E
*
*** BPOFF ***
       DATA X000D
X000E  DATA >8542,>504F,>46C6
       DATA DOUSER,>40
*
*** FENCE ***
       DATA X000E
L1054  DATA >8546,>454E,>43C5
FENCE  DATA DOUSER,>42
*
*** BLK ***
       DATA L1054
```

```
L1055  DATA >8342,>4CCB
BLK    DATA DOUSER,>44
*
*** IN ***
       DATA L1055
L1056  DATA >8249,>4EA0
IN     DATA DOUSER,>46
*
*** OUT ***
       DATA L1056
L1057  DATA >834F,>55D4
OUT    DATA DOUSER,>48
*
*** SCR ***
       DATA L1057
L1058  DATA >8353,>43D2
SCR    DATA DOUSER,>4A
*
*** CONTEXT ***
       DATA L1058
L105A  DATA >8743,>4F4E,>5445,>58D4
CONTEX DATA DOUSER,>4C
*
*** CURRENT ***
       DATA L105A
L105B  DATA >8743,>5552,>5245,>4ED4
CURREN DATA DOUSER,>4E
*
*** STATE ***
       DATA L105B
L105C  DATA >8553,>5441,>54C5
STATE  DATA DOUSER,>50
*
*** BASE ***
       DATA L105C
L105D  DATA >8442,>4153,>45A0
BASE   DATA DOUSER,>52
*
*** DPL ***
       DATA L105D
L105E  DATA >8344,>50CC
DPL    DATA DOUSER,>54
*
*** FLD ***
       DATA L105E
L105F  DATA >8346,>4CC4
FLD    DATA DOUSER,>56
*
*** CSP ***
       DATA L105F
L1060  DATA >8343,>53D0
CSP    DATA DOUSER,>58
*
*** R# ***
       DATA L1060
L1061  DATA >8252,>23A0
RNUM   DATA DOUSER,>5A
*
*** HLD ***
       DATA L1061
L1062  DATA >8348,>4CC4
HLD    DATA DOUSER,>5C
*
*** USE ***
       DATA L1062
L1063  DATA >8355,>53C5
```

```
USE    DATA DOUSER,>5E
*
*** PREV ***
       DATA L1063
L1064  DATA >8450,>5245,>56A0
PREV   DATA DOUSER,>60
*
*** ECOUNT ***
       DATA L1064
L1066  DATA >8645,>434F,>554E,>54A0
ECOUNT DATA DOUSER,>62          <---changed from >64
*
*** VOC-LINK ***
       DATA L1066
L1066X DATA >8856,>4F43,>2D4C,>494E,>4BA0
VLINK  DATA DOUSER,>64          <---changed from >66
*
*
_RLAST EQU  $
*
       DORG 0
UBASE  BSS  6             BASE OF USER VARIABLES
$UCONS BSS  2             06 USER UCONS
$S0    BSS  2             08 USER S0
$R0    BSS  2             0A USER R0 { R0$
$U0    BSS  2             0C USER U0
       BSS  2             0E USER TIB
       BSS  2             10 USER WIDTH
$DP    BSS  2             12 USER DP
$SYS   BSS  2             14 USER SYS$
CURPO$ BSS  2             16 USER CURPOS
$INTLK BSS  2             18 USER INTLNK
       BSS  2             1A USER WARNING
       BSS  2             1C USER C/L$ { CL$
       BSS  2             1E USER FIRST$
       BSS  2             20 USER LIMIT$
       BSS  2             22 USER MGT gets disk buffer offset for system messages
*                                     from here, + disk offset and pushes to stack
       BSS  2             24 USER LCT gets disk buffer offset for true lowercase
*                                     from here, + disk offset and pushes to stack
       BSS  2             26 USER JMODE    loaded by graphics primitives; used by JOYST
       BSS  2             28 USER      <---available
       BSS  2             2A USER DBF gets disk buffer offset for default blocks filename
*                                     from here, + disk offset and pushes to stack
       BSS  2             2C USER DISK_BUF
       BSS  2             2E USER PABS
       BSS  2             30 USER SCRN_WIDTH
       BSS  2             32 USER SCRN_START
       BSS  2             34 USER SCRN_END
       BSS  2             36 USER ISR
       BSS  2             38 USER ALTIN
       BSS  2             3A USER ALTOUT
       BSS  2             3C USER VDPMDE
       BSS  2             3E USER BPB gets disk buffer offset for blocks file PABs
*                                     from here, + disk offset and pushes to stack
       BSS  2             40 USER BPOFF  Offset into BPABS for current blocks file
ULNGTH EQU  $
       BSS  2             42 USER FENCE
       BSS  2             44 USER BLK
       BSS  2             46 USER IN
$OUT   BSS  2             48 USER OUT
       BSS  2             4A USER SCR
       BSS  2             4C USER CONTEXT
       BSS  2             4E USER CURRENT
       BSS  2             50 USER STATE
       BSS  2             52 USER BASE
       BSS  2             54 USER DPL
       BSS  2             56 USER FLD
       BSS  2             58 USER CSP
       BSS  2             5A USER R# { RNUM
       BSS  2             5C USER HLD
       BSS  2             5E USER USE
       BSS  2             60 USER PREV
       BSS  2             62 USER ECOUNT
       BSS  2             64 VOC-LINK
UMAX   BSS  0
       AORG _RLAST
*
*** C/L ***
       DATA L1066X
L1067  DATA >8343,>2FCC
CSL    DATA DOCOL,CL$,AT,SEMIS
*
*** B/BUF ***   <now explicitly 1024>
       DATA L1067
L1068  DATA >8542,>2F42,>55C6
BSLBUF DATA DOCON,1024
*
*** B/SCR ***    <now explicitly 1 for backward compatibility>
       DATA L1068
L1069  DATA >8542,>2F53,>43D2
BSLSCR DATA DOCON,1

*** FIRST ***
       DATA L1069
L106A  DATA >8546,>4952,>53D4
FIRST  DATA DOCOL,FIRST$,AT,SEMIS
*
*** LIMIT ***
       DATA L106A
L106B  DATA >854C,>494D,>49D4
LIMIT  DATA DOCOL,LIMIT$,AT,SEMIS
*
*** HERE ***
       DATA L106B
L106D  DATA >8448,>4552,>45A0
HERE   DATA DOCOL,DP,AT,SEMIS
*
*** ALLOT ***
       DATA L106D
L106E  DATA >8541,>4C4C,>4FD4
ALLOT  DATA DOCOL,SPAT,OVER,HERE,PLUS,LIT,>80
       DATA PLUS,ULESS,TWO,QERROR,DP,PSTORE
       DATA SEMIS
*
*** , ***
       DATA L106E
L106F  DATA >81AC
COMMA  DATA DOCOL,HERE,STORE,TWO,ALLOT,SEMIS
*
*** C, ***
       DATA L106F
L1070  DATA >8243,>2CA0
CCOMMA DATA DOCOL,HERE,CSTORE,ONE,ALLOT,SEMIS
*
*** = ***
       DATA L1070
L1071  DATA >81BD
EQUAL  DATA DOCOL,SUB,ZEQU,SEMIS
*
*** < ***
       DATA L1071
```

```
L1072  DATA >81BC
LESS   DATA $+2
       CLR  TEMP1
       C    *SP+,*SP
       JLT  LESS1
       JEQ  LESS1
       INC  TEMP1
LESS1  MOV  TEMP1,*SP
       B    *NEXT
*
*** > ***
       DATA L1072
L1073  DATA >81BE
GREAT  DATA DOCOL,SWAP,LESS,SEMIS
*
*** SGN *** return sign of n or 0
*           ( n --- -1|0|+1 )
       DATA L1073
L1073A DATA >8353,>47CE
SGN    DATA DOCOL,DUP,ABS,DDIV,SEMIS
*
*** ROT ***
       DATA L1073A
L1074  DATA >8352,>4FD4
ROT    DATA DOCOL,TOR,SWAP,FROMR,SWAP,SEMIS
*
*** SPACE ***
       DATA L1074
L1075  DATA >8553,>5041,>43C5
SPACE  DATA DOCOL,BL,EMIT,SEMIS
*
*** -DUP ***
       DATA L1075
L1076  DATA >842D,>4455,>50A0
DDUP   DATA DOCOL,DUP,ZBRAN,L1077-$,DUP
L1077  DATA SEMIS
*
*** TRAVERSE ***
       DATA L1076
L1078  DATA >8854,>5241,>5645,>5253,>45A0
TRAVER DATA DOCOL,SWAP
L1079  DATA OVER,PLUS,LIT,>7F,OVER,CAT,LESS,ZBRAN
       DATA L1079-$,SWAP,DROP,SEMIS
*
*** CFA ***
       DATA L1078
L107A  DATA >8343,>46C1
CFA    DATA DOCOL,TWOM,SEMIS
*
*** NFA ***
       DATA L107A
L107B  DATA >834E,>46C1
NFA    DATA DOCOL,THREE,SUB,LIT,>FFFF,TRAVER,SEMIS
*
*** PFA ***
       DATA L107B
L107C  DATA >8350,>46C1
PFA    DATA DOCOL,ONE,TRAVER,THREE,PLUS,SEMIS
*
*** LFA ***
       DATA L107C
L107D  DATA >834C,>46C1
LFA    DATA DOCOL,NFA,TWOM,SEMIS
*
*** LATEST ***
       DATA L107D
```

```
L107E  DATA >864C,>4154,>4553,>54A0
LATEST DATA DOCOL,CURREN,AT,AT,SEMIS
*
*** !CSP ***
       DATA L107E
L107F  DATA >8421,>4353,>50A0
STRCSP DATA DOCOL,SPAT,CSP,STORE,SEMIS
*
*** ?ERROR ***
*        ( flag msg# --- )
       DATA L107F
L1080  DATA >863F,>4552,>524F,>52A0
QERROR DATA DOCOL,SWAP,ZBRAN,L1081-$,ERROR,BRANCH
       DATA L1082-$
L1081  DATA DROP
L1082  DATA SEMIS
*
*** ?COMP ***
       DATA L1080
L1083  DATA >853F,>434F,>4DD0
QCOMP  DATA DOCOL,STATE,AT,ZEQU,LIT,>11,QERROR
       DATA SEMIS
*
*** ?EXEC ***
       DATA L1083
L1084  DATA >853F,>4558,>45C3
QEXEC  DATA DOCOL,STATE,AT,LIT,>12,QERROR,SEMIS
*
*** ?PAIRS ***
       DATA L1084
L1085  DATA >863F,>5041,>4952,>53A0
QPAIRS DATA DOCOL,SUB,LIT,>13,QERROR,SEMIS
*
*** ?CSP ***
       DATA L1085
L1086  DATA >843F,>4353,>50A0
QCSP   DATA DOCOL,SPAT,CSP,AT,SUB,LIT,>14,QERROR
       DATA SEMIS
*
*** ?LOADING ***
       DATA L1086
L1087  DATA >883F,>4C4F,>4144,>494E,>47A0
QLOADI DATA DOCOL,BLK,AT,ZEQU,LIT,>16,QERROR,SEMIS
*
*** COMPILE ***
       DATA L1087
L1088  DATA >8743,>4F4D,>5049,>4CC5
COMPIL DATA DOCOL,QCOMP,FROMR,DUP,TWOP,TOR,AT,COMMA
       DATA SEMIS
*
*** [ ***       [ IMMEDIATE word ]
       DATA L1088
L1089  DATA >C1DB
LBRCKT DATA DOCOL,ZERO,STATE,STORE,SEMIS
*
*** ] ***
       DATA L1089
L108A  DATA >81DD
RBRCKT DATA DOCOL,LIT,>C0,STATE,STORE,SEMIS
*
*** SMUDGE ***
       DATA L108A
L108B  DATA >8653,>4D55,>4447,>45A0
SMUDGE DATA DOCOL,LATEST,LIT,>20,TOGGLE,SEMIS
*
*** HEX ***
```

```
       DATA L108B
L108C  DATA >8348,>45D8
HEX    DATA DOCOL,LIT,>10,BASE,STORE,SEMIS
*
*** DECIMAL ***
       DATA L108C
L108D  DATA >8744,>4543,>494D,>41CC
DECIMA DATA DOCOL,LIT,>A,BASE,STORE,SEMIS
*
*** COUNT ***
       DATA L108D
L108E  DATA >8543,>4F55,>4ED4
COUNT  DATA DOCOL,DUP,ONEP,SWAP,CAT,SEMIS
*
*** TYPE ***
       DATA L108E
L108F  DATA >8454,>5950,>45A0
TYPE   DATA DOCOL,DDUP,ZBRAN,L1090-$,ZERO,PDO
L1091  DATA DUP,CAT,EMIT,ONEP,PLOOP,L1091-$
L1090  DATA DROP,SEMIS
*
*** -TRAILING ***
       DATA L108F
L1092  DATA >892D,>5452,>4149,>4C49,>4EC7
DTRAIL DATA DOCOL,DUP,ZERO,PDO
L1093  DATA OVER,OVER,PLUS,ONEM,CAT,BL,SUB,ZBRAN
       DATA L1094-$,LEAVE,BRANCH,L1095-$
L1094  DATA ONEM
L1095  DATA PLOOP,L1093-$,SEMIS
*
*** ?STACK ***
       DATA L1092
L1096  DATA >863F,>5354,>4143,>4BA0
QSTACK DATA DOCOL,SPAT,S0,AT,SWAP,ULESS,ONE,QERROR
       DATA SPAT,HERE,LIT,>80,PLUS,ULESS
       DATA LIT,>7
       DATA QERROR,SEMIS
*
*** EXPECT ***
       DATA L1096
L1097  DATA >8645,>5850,>4543,>54A0
EXPECT DATA DOCOL,ZERO,PDO
L1098  DATA KEY,DUP,LIT,>D,EQUAL,ZBRAN,L1099-$
       DATA DROP,SPACE,LEAVE,ZERO,BRANCH,L109A-$
L1099  DATA DUP,LIT,>8,EQUAL,ZBRAN,L109B-$,DROP
       DATA I,ZEQU,ZBRAN,L109C-$,LIT,>7,EMIT,ZERO
       DATA BRANCH,L109D-$
L109C  DATA LIT,>8,EMIT,FROMR,ONEM,TOR,ONEM
       DATA ZERO
L109D  DATA BRANCH,L109E-$
L109B  DATA DUP,EMIT,OVER,CSTORE,ONEP,ONE
L109E
L109A  DATA PPLOOP,L1098-$,ZERO,SWAP,OVER,OVER
       DATA CSTORE,ONEP,CSTORE,SEMIS
*
*** QUERY ***
       DATA L1097
L109F  DATA >8551,>5545,>52D9
QUERY  DATA DOCOL,TIB,AT,LIT,>50,EXPECT,ZERO,IN
       DATA STORE,SEMIS
*
*** FILL ***
       DATA L109F
L10A0  DATA >8446,>494C,>4CA0
FILL   DATA DOCOL,SWAP,TOR,OVER,CSTORE,DUP,ONEP
       DATA FROMR,ONEM,CMOVE,SEMIS
```

```
*
*** ERASE ***
       DATA L10A0
L10A1  DATA >8545,>5241,>53C5
ERASE  DATA DOCOL,ZERO,FILL,SEMIS
*
*** BLANKS ***
       DATA L10A1
L10A2  DATA >8642,>4C41,>4E4B,>53A0
BLANKS DATA DOCOL,BL,FILL,SEMIS
*
*** HOLD ***
       DATA L10A2
L10A3  DATA >8448,>4F4C,>44A0
HOLD   DATA DOCOL,LIT,>FFFF,HLD,PSTORE,HLD,AT,CSTORE
       DATA SEMIS
*
*** PAD ***
       DATA L10A3
L10A4  DATA >8350,>41C4
PAD    DATA DOCOL,HERE,LIT,>44,PLUS,SEMIS
*
*** WORD ***
       DATA L10A4
L10A5  DATA >8457,>4F52,>44A0
WORD   DATA DOCOL,BLK,AT,ZBRAN,L10A6-$,BLK,AT,BLOCK
       DATA BRANCH,L10A7-$
L10A6  DATA TIB,AT
L10A7  DATA IN,AT,PLUS,SWAP,ENCLOS,HERE,LIT,>22
       DATA BLANKS,IN,PSTORE,OVER,SUB,DUP,TOR,HERE
       DATA CSTORE,PLUS,HERE,ONEP,FROMR,CMOVE,SEMIS
*
*** (.") ***
       DATA L10A5
L10A8  DATA >8428,>2E22,>29A0
PTYPE  DATA DOCOL,RR,COUNT,DUP,ONEP,ECELLS,FROMR
       DATA PLUS,TOR,TYPE,SEMIS
*
*** ." ***              [ IMMEDIATE word ]
       DATA L10A8
L10A9  DATA >C22E,>22A0
STRNG  DATA DOCOL,LIT,>22,STATE,AT,ZBRAN,L10AA-$
       DATA COMPIL,PTYPE,WORD,HERE,CAT,ONEP,ECELLS
       DATA ALLOT,BRANCH,L10AB-$
L10AA  DATA WORD,HERE,COUNT,TYPE
L10AB  DATA SEMIS
*
*** (NUMBER) ***
       DATA L10A9
L10AC  DATA >8828,>4E55,>4D42,>4552,>29A0
PNUMBR DATA DOCOL
L10AD  DATA ONEP,DUP,TOR,CAT,BASE,AT,DIGIT,ZBRAN
       DATA L10AE-$,SWAP,BASE,AT,MULT,DROP,ROT
       DATA BASE,AT,MULT,DPLUS,DPL,AT,ONEP,ZBRAN
       DATA L10AF-$,ONE,DPL,PSTORE
L10AF  DATA FROMR,BRANCH,L10AD-$
L10AE  DATA FROMR,SEMIS
*
*** NUMBER ***
       DATA L10AC
L10B0  DATA >864E,>554D,>4245,>52A0
NUMBER DATA DOCOL,ZERO,ZERO,ROT,DUP,ONEP,CAT,LIT
       DATA >2D,EQUAL,DUP,TOR,PLUS,LIT,>FFFF
L10B1  DATA DPL,STORE,PNUMBR,DUP,CAT,BL,SUB,ZBRAN
       DATA L10B2-$,DUP,CAT,LIT,>2E,SUB,ZERO,QERROR
       DATA ZERO,BRANCH,L10B1-$
```

```
L10B2  DATA DROP,FROMR,ZBRAN,L10B3-$,DMINUS
L10B3  DATA SEMIS
*
*** -FIND ***
       DATA L10B0
L10B4  DATA >852D,>4649,>4EC4
DFIND  DATA DOCOL,BL,WORD,HERE,CONTEX,AT,AT,PFIND
       DATA DUP,ZEQU,ZBRAN,L10B5-$,DROP,HERE,LATEST
       DATA PFIND
L10B5  DATA SEMIS
*
*** (ABORT) ***
       DATA L10B4
L10B6  DATA >8728,>4142,>4F52,>54A9
PABORT DATA DOCOL,ABORT,SEMIS
*
*** ERROR ***
*       ( msg# --- IN BLK )
       DATA L10B6
L10B7  DATA >8545,>5252,>4FD2
ERROR  DATA DOCOL,WARNIN,AT,ZLESS,ZBRAN,L10B8-$
       DATA PABORT,BRANCH,L10B9-$
L10B8  DATA ECOUNT,AT,ZEQU,ZBRAN,L10BA-$,ONE,ECOUNT
       DATA STORE,HERE,COUNT,TYPE,PTYPE,>420,>203F
       DATA >2020,MESSAG
L10BA
L10B9  DATA ZERO,ECOUNT,STORE,SPSTOR,IN,AT,BLK
       DATA AT,QUIT,SEMIS
*
*** ID. ***
       DATA L10B7
L10BB  DATA >8349,>44AE
IDDOT  DATA DOCOL,PAD,LIT,>20,LIT,>5F,FILL,DUP
       DATA ONE,TRAVER,OVER,SUB,DUP,TOR,ONEP,PAD
       DATA SWAP,CMOVE,PAD,FROMR,PLUS,LIT,>80,TOGGLE
       DATA PAD,COUNT,LIT,>1F,_AND,TYPE,SPACE,SEMIS
*
*** CREATE ***
       DATA L10BB
L10BC  DATA >8643,>5245,>4154,>45A0
CREATE DATA DOCOL,HERE,ECELLS,DP,STORE
       DATA LATEST,COMMA,DFIND,ZBRAN,L10BD-$
       DATA DROP,NFA,IDDOT,LIT,>4,MESSAG,SPACE
L10BD  DATA HERE,DUP,CAT,WIDTH,AT,MIN,ONEP,ECELLS
       DATA ALLOT,DUP,LIT,>A0,TOGGLE,HERE,ONEM
       DATA LIT,>80,TOGGLE,CURREN,AT,STORE,HERE
       DATA TWOP,COMMA,SEMIS
*
*** [COMPILE] ***       [ IMMEDIATE word ]
       DATA L10BC
L10BE  DATA >C95B,>434F,>4D50,>494C,>45DD
BCOMPI DATA DOCOL,DFIND,ZEQU,ZERO,QERROR,DROP,CFA
       DATA COMMA,SEMIS
*
*** LITERAL ***         [ IMMEDIATE word ]
       DATA L10BE
L10BF  DATA >C74C,>4954,>4552,>41CC
LITERA DATA DOCOL,STATE,AT,ZBRAN,L10C0-$,COMPIL
       DATA LIT,COMMA
L10C0  DATA SEMIS
*
*** DLITERAL ***        [ IMMEDIATE word ]
       DATA L10BF
L10C1  DATA >C844,>4C49,>5445,>5241,>4CA0
DLITER DATA DOCOL,STATE,AT,ZBRAN,L10C2-$,SWAP,LITERA
       DATA LITERA
```

```
L10C2  DATA SEMIS
*
*** INTERPRET ***
       DATA L10C1
L10C3  DATA >8949,>4E54,>4552,>5052,>45D4
INTERP DATA DOCOL
L10C4  DATA DFIND,ZBRAN,L10C5-$,STATE,AT,LESS,ZBRAN
       DATA L10C6-$,CFA,COMMA,BRANCH,L10C7-$
L10C6  DATA CFA,EXECUT
L10C7  DATA QSTACK,BRANCH,L10C8-$
L10C5  DATA HERE,NUMBER,DPL,AT,ONEP,ZBRAN,L10C9-$
       DATA DLITER,BRANCH,L10CA-$
L10C9  DATA DROP,LITERA
L10CA  DATA QSTACK
L10C8  DATA BRANCH,L10C4-$,SEMIS
*
*** IMMEDIATE ***
       DATA L10C3
L10CB  DATA >8949,>4D4D,>4544,>4941,>54C5
IMMEDI DATA DOCOL,LATEST,LIT,>40,TOGGLE,SEMIS
*
*** ( ***               [ IMMEDIATE word ]
       DATA L10CB
L10CC  DATA >C1A8
PAREN  DATA DOCOL,LIT,>29,WORD,SEMIS
*
*** FORTH ***           [ IMMEDIATE word ]
       DATA L10CC
L10CD  DATA >C546,>4F52,>54C8
FORTHV EQU  $+2         vocabulary link field
FORTHP EQU  $+4         pseudo name field
FORTHL EQU  $+6         chronological link field
FORTH  DATA DOVOC,$TASK1+16,>81A0,0     (may need to modify)
*
*** DEFINITIONS ***
       DATA L10CD
L10CE  DATA >8B44,>4546,>494E,>4954,>494F,>4ED3
DEFINI DATA DOCOL,CONTEX,AT,CURREN,STORE,SEMIS
*
*** QUIT ***
       DATA L10CE
L10CF  DATA >8451,>5549,>54A0
QUIT   DATA DOCOL,ZERO,BLK,STORE,LBRCKT
L10D0  DATA RSTOR,CR,QUERY,INTERP,STATE,AT,ZEQU
       DATA ZBRAN,L10D1-$,PTYPE,>420,>6F6B,>3A20,DEPTH,DOT
L10D1  DATA BRANCH,L10D0-$,SEMIS
*
*** ABORT ***   finishes by displaying "fbForth 1.0"
       DATA L10CF
L10D2  DATA >8541,>424F,>52D4
ABORT  DATA DOCOL,SPSTOR,DECIMA,ZERO,ECOUNT,STORE,CR
       DATA PTYPE,>0B66,>6246,>6F72,>7468,>2031,>2E30
       DATA FORTH,DEFINI,QUIT
       DATA SEMIS
*
*** +- ***
       DATA L10D2
L10D3  DATA >822B,>2DA0
PM     DATA DOCOL,ZLESS,ZBRAN,L10D4-$,MINUS
L10D4  DATA SEMIS
*
*** D+- ***
       DATA L10D3
L10D5  DATA >8344,>2BAD
DPM    DATA DOCOL,ZLESS,ZBRAN,L10D6-$,DMINUS
L10D6  DATA SEMIS
```

```
*
*** DABS ***
       DATA L10D5
L10D7  DATA >8444,>4142,>53A0
DABS   DATA DOCOL,DUP,DPM,SEMIS
*
*** M* ***
       DATA L10D7
L10D8  DATA >824D,>2AA0
MSTAR  DATA DOCOL,OVER,OVER,_XOR,TOR,ABS,SWAP,ABS
       DATA MULT,FROMR,DPM,SEMIS
*
*** M/ ***
       DATA L10D8
L10D9  DATA >824D,>2FA0
MSLASH DATA DOCOL,OVER,TOR,TOR,DABS,RR,ABS,DIV
       DATA FROMR,RR,_XOR,PM,SWAP,FROMR,PM,SWAP
       DATA SEMIS
*
*** * ***
       DATA L10D9
L10DA  DATA >81AA
TIMES  DATA DOCOL,MULT,DROP,SEMIS
*
*** /MOD ***
       DATA L10DA
L10DB  DATA >842F,>4D4F,>44A0
DMOD   DATA DOCOL,TOR,STOD,FROMR,MSLASH,SEMIS
*
*** / ***
       DATA L10DB
L10DC  DATA >81AF
DDIV   DATA DOCOL,DMOD,SWAP,DROP,SEMIS
*
*** MOD ***
       DATA L10DC
L10DD  DATA >834D,>4FC4
MOD    DATA DOCOL,DMOD,DROP,SEMIS
*
*** */MOD ***
       DATA L10DD
L10DE  DATA >852A,>2F4D,>4FC4
MDMOD  DATA DOCOL,TOR,MSTAR,FROMR,MSLASH,SEMIS
*
*** */ ***
       DATA L10DE
L10DF  DATA >822A,>2FA0
MD     DATA DOCOL,MDMOD,SWAP,DROP,SEMIS
*
*** M/MOD ***
       DATA L10DF
L10E0  DATA >854D,>2F4D,>4FC4
MSLMOD DATA DOCOL,TOR,ZERO,RR,DIV,FROMR,SWAP,TOR
       DATA DIV,FROMR,SEMIS
*
*** SPACES ***
       DATA L10E0
L10E1  DATA >8653,>5041,>4345,>53A0
SPACES DATA DOCOL,ZERO,MAX,DDUP,ZBRAN,L10E2-$,ZERO
       DATA PDO
L10E3  DATA SPACE,PLOOP,L10E3-$
L10E2  DATA SEMIS
*
*** <# ***
       DATA L10E1
L10E4  DATA >823C,>23A0
```

```
STRTCN DATA DOCOL,PAD,HLD,STORE,SEMIS
*
*** #> ***
       DATA L10E4
L10E5  DATA >8223,>3EA0
STOPCN DATA DOCOL,DROP,DROP,HLD,AT,PAD,OVER,SUB
       DATA SEMIS
*
*** SIGN ***
       DATA L10E5
L10E6  DATA >8453,>4947,>4EA0
SIGN   DATA DOCOL,ROT,ZLESS,ZBRAN,L10E7-$,LIT,>2D
       DATA HOLD
L10E7  DATA SEMIS
*
*** # ***
       DATA L10E6
L10E8  DATA >81A3
NUMSGN DATA DOCOL,PAD,HLD,AT,SUB,DPL,AT,EQUAL,ZBRAN
       DATA L10E9-$,LIT,>2E,HOLD
L10E9  DATA BASE,AT,MSLMOD,ROT,LIT,>9,OVER,LESS
       DATA ZBRAN,L10EA-$,LIT,>7,PLUS
L10EA  DATA LIT,>30,PLUS,HOLD,SEMIS
*
*** #S ***
       DATA L10E8
L10EB  DATA >8223,>53A0
NUMS   DATA DOCOL
L10EC  DATA NUMSGN,OVER,OVER,_OR,ZEQU,ZBRAN,L10EC-$
       DATA SEMIS
*
*** D.R ***
       DATA L10EB
L10ED  DATA >8344,>2ED2
DDOTR  DATA DOCOL,TOR,SWAP,OVER,DABS,STRTCN,NUMS
       DATA SIGN,STOPCN,FROMR,OVER,SUB,SPACES,TYPE
       DATA SEMIS
*
*** D. ***
       DATA L10ED
L10EE  DATA >8244,>2EA0
DDOT   DATA DOCOL,ZERO,DDOTR,SPACE,SEMIS
*
*** .R ***
       DATA L10EE
L10EF  DATA >822E,>52A0
DOTR   DATA DOCOL,TOR,STOD,FROMR,DDOTR,SEMIS
*
*** . ***
       DATA L10EF
L10F0  DATA >81AE
DOT    DATA DOCOL,STOD,DDOT,SEMIS
*
*** ? ***
       DATA L10F0
L10F1  DATA >81BF
QMARK  DATA DOCOL,AT,DOT,SEMIS
*
*** UD.R ***
       DATA L10F1
L10F2  DATA >8455,>442E,>52A0
UDDOTR DATA DOCOL,TOR,STRTCN,NUMS,STOPCN,FROMR
       DATA OVER,SUB,SPACES,TYPE,SEMIS
*
*** UD. ***
       DATA L10F2
```

```
L10F3  DATA >8355,>44AE
UDDOT  DATA DOCOL,ZERO,UDDOTR,SPACE,SEMIS
*
*** U.R ***
       DATA L10F3
L10F4  DATA >8355,>2ED2
UDOTR  DATA DOCOL,TOR,ZERO,FROMR,UDDOTR,SEMIS
*
*** U. ***
       DATA L10F4
L10F5  DATA >8255,>2EA0
UDOT   DATA DOCOL,ZERO,UDDOT,SEMIS
*
*** +BUF ***
       DATA L10F5
L10F6  DATA >842B,>4255,>46A0
PLSBUF DATA DOCOL,BSLBUF,LIT,>4,PLUS,PLUS,DUP,LIMIT
       DATA EQUAL,ZBRAN,L10F7-$,DROP,FIRST
L10F7  DATA DUP,PREV,AT,SUB,SEMIS
*
*** BUFFER ***
*       ( block# --- addr )
       DATA L10F6
L10F8  DATA >8642,>5546,>4645,>52A0
BUFFER DATA DOCOL,USE,AT,DUP,TOR
L10F9  DATA PLSBUF,ZBRAN,L10F9-$,USE,STORE,RR,AT
       DATA ZLESS,ZBRAN,L10FA-$,RR,TWOP,RR,AT,LIT
       DATA >7FFF,_AND,ZERO,RSLW
L10FA  DATA RR,STORE,RR,PREV,STORE,FROMR,TWOP,SEMIS
*
*** UPDATE ***
       DATA L10F8
L10FB  DATA >8655,>5044,>4154,>45A0
UPDATE DATA DOCOL,PREV,AT,AT,LIT,>8000,_OR,PREV
       DATA AT,STORE,SEMIS
*
*** FLUSH ***
       DATA L10FB
L10FC  DATA >8546,>4C55,>53C8
FLUSH  DATA DOCOL,LIMIT,FIRST,SUB,BSLBUF,LIT,>4
       DATA PLUS,DDIV,ONEP,ZERO,PDO
L10FD  DATA LIT,>7FFF,BUFFER,DROP,PLOOP,L10FD-$
       DATA SEMIS
*
*** EMPTY-BUFFERS ***
       DATA L10FC
L10FE  DATA >8D45,>4D50,>5459,>2D42,>5546,>4645
       DATA >52D3
EMPTYB DATA DOCOL,FIRST,LIMIT,OVER,SUB,ERASE,FLUSH
       DATA FIRST,USE,STORE,FIRST,PREV,STORE,SEMIS
*
*
*** CLEAR ***
*       ( block# --- )
       DATA L10FE
L10FF  DATA >8543,>4C45,>41D2
CLEAR  DATA DOCOL,DUP,SCR,STORE,FLUSH
       DATA BUFFER,BSLBUF,BLANKS,UPDATE,SEMIS
*
*** CLR_BLKS ***    CLEAR a range of blocks to blanks in the current
*            blocks file.  The blocks are FLUSHed to disk when done.
*       ( firstblock# lastblock# --- )
       DATA L10FF
L1100  DATA >8843,>4C52,>5F42,>4C4B,>53A0
CLRBLS DATA DOCOL,ONEP,SWAP,PDO
L1100A DATA I,CLEAR,PLOOP,L1100A-$,FLUSH,SEMIS
```

```
*
*** BLOCK ***
*       ( block# --- addr )
       DATA L1100
L1101  DATA >8542,>4C4F,>43CB
BLOCK  DATA DOCOL,TOR,PREV,AT,DUP
       DATA AT,RR,SUB,DUP,PLUS,ZBRAN,L1102-$
L1103  DATA PLSBUF,ZEQU,ZBRAN,L1104-$,DROP,RR,BUFFER
       DATA DUP,RR,ONE,RSLW,TWOM
L1104  DATA DUP,AT,RR,SUB,DUP,PLUS,ZEQU,ZBRAN
       DATA L1103-$,DUP,PREV,STORE
L1102  DATA FROMR,DROP,TWOP,SEMIS
*
*
*** (LINE) ***
       DATA L1101
L1105  DATA >8628,>4C49,>4E45,>29A0
PLINE  DATA DOCOL,TOR,CSL,BSLBUF,MDMOD,FROMR
       DATA PLUS,BLOCK,PLUS,CSL,SEMIS
*
*** .LINE ***
       DATA L1105
L1106  DATA >852E,>4C49,>4EC5
DOTLN  DATA DOCOL,PLINE,DTRAIL,TYPE,SEMIS
*
*** MSGMAX ***
       DATA L1106
L1106X DATA >864D,>5347,>4D41,>58A0
MSMAX  DATA DOCOL,SYSM$,ONEP,_VSBR,SEMIS
*
*** MSG# ***
*       ( msg# --- )
       DATA L1106X
L1106Y DATA >844D,>5347,>23A0
MSGNUM DATA DOCOL,PTYPE,>056D,>7367,>2023,DOT,SEMIS
*
*
*** MESSAGE ***
       DATA L1106Y
L1107  DATA >874D,>4553,>5341,>47C5
MESSAG DATA DOCOL,WARNIN,AT,ZBRAN,L1108-$,DDUP
       DATA ZBRAN,L1109-$,DUP,ZLESS,OVER,MSMAX
       DATA GREAT,_OR,ZBRAN,L1109M-$,MSGNUM
       DATA PTYPE,>0208,>3F20
L1109N DATA BRANCH,L1109-$
L1109M DATA DUP,LIT,MTIDX,PLUS,CAT,DDUP,ZBRAN,L1109L-$
       DATA PLUS,SYSM$,PLUS,PAD,OVER,_VSBR,ONEP,_VMBR
       DATA PAD,COUNT,TYPE,BRANCH,L1109-$
L1109L DATA MSGNUM,PTYPE,>0208,>3F20
L1109  DATA BRANCH,L110A-$
L1108  DATA MSGNUM
L110A  DATA SEMIS
*
*
*** LOAD ***
       DATA L1107
L110B  DATA >844C,>4F41,>44A0
LOAD   DATA DOCOL,DDUP,ZEQU,LIT,8,QERROR,BLK,AT
       DATA TOR,IN,AT,TOR,ZERO,IN
       DATA STORE,BLK,STORE,INTERP
       DATA FROMR,IN,STORE,FROMR
       DATA BLK,STORE,SEMIS
*
*
*** --> ***              [ IMMEDIATE word ]
       DATA L110B
```

```
L110C  DATA >C32D,>2DBE
ARROW  DATA DOCOL,QLOADI,ZERO,IN,STORE
       DATA ONE,BLK,PSTORE,SEMIS
*
*** R/W ***
*       ( bufaddr block# flag --- )
       DATA L110C
BF00F  DATA >8352,>2FD7
RSLW   DATA DOCOL,ZBRAN,L110E-$,RBLK
       DATA BRANCH,L110F-$
L110E  DATA WBLK
L110F  DATA SEMIS
*
*** ' ***               [ IMMEDIATE word ]
       DATA BF00F
L1110  DATA >C1A7
TICK   DATA DOCOL,DFIND,ZEQU,ZERO,QERROR,DROP,LITERA
       DATA SEMIS
*
*** UNFORGETABLE ***
       DATA L1110
L1110X DATA >8C55,>4E46,>4F52,>4745,>5441,>424C,>45A0
UNFORG DATA DOCOL,DUP,FENCE,AT,ULESS,OVER,LIT,$TASK1
       DATA ULESS,_OR,HERE,ROT,ULESS,_OR,SEMIS
*
*** FORGET ***
       DATA L1110X
L1111  DATA >8646,>4F52,>4745,>54A0
FORGET DATA DOCOL,TICK,LFA,DUP,UNFORG,LIT,>15,QERROR
       DATA TOR,VLINK,AT
FORGE1 DATA RR,OVER,ULESS,OVER,UNFORG,ZEQU,_AND
       DATA ZBRAN,FORGE2-$,FORTH,DEFINI,AT
       DATA BRANCH,FORGE1-$
FORGE2 DATA DUP,VLINK,STORE
FORGE3 DATA DUP,TWOM
FORGE4 DATA PFA,LFA,AT,DUP,PFA,LFA,RR,ULESS,OVER
       DATA UNFORG,_OR,ZBRAN,FORGE4-$
       DATA OVER,LIT,>4,SUB,STORE,AT,DDUP,ZEQU
       DATA ZBRAN,FORGE3-$,FROMR,DP,STORE,SEMIS
*
*** : ***               [ IMMEDIATE word ]
       DATA L1111
L1112  DATA >C1BA
COLON  DATA DOCOL,QEXEC,STRCSP,CURREN,AT,CONTEX
       DATA STORE,CREATE,RBRCKT,LIT,DOCOL
       DATA HERE,TWOM,STORE,SEMIS
*
*** ; ***               [ IMMEDIATE word ]
       DATA L1112
L1113  DATA >C1BB
SEMIC  DATA DOCOL,QCSP,COMPIL,SEMIS,SMUDGE,LBRCKT
       DATA SEMIS
*
*** BACK ***
       DATA L1113
L1114  DATA >8442,>4143,>4BA0
BACK   DATA DOCOL,HERE,SUB,COMMA,SEMIS
*
*** BEGIN ***           [ IMMEDIATE word ]
       DATA L1114
L1115  DATA >C542,>4547,>49CE
_BEGIN DATA DOCOL,QCOMP,HERE,ONE,SEMIS
*
*** ENDIF ***           [ IMMEDIATE word ]
       DATA L1115
L1116  DATA >C545,>4E44,>49C6
```

```
_ENDIF DATA DOCOL,QCOMP,TWO,QPAIRS,HERE,OVER,SUB
       DATA SWAP,STORE,SEMIS
*
*** THEN ***            [ IMMEDIATE word ]
       DATA L1116
L1117  DATA >C454,>4845,>4EA0
_THEN  DATA DOCOL,_ENDIF,SEMIS
*
*** DO ***              [ IMMEDIATE word ]
       DATA L1117
L1118  DATA >C244,>4FA0
_DO    DATA DOCOL,QCOMP,COMPIL,PDO,HERE,THREE,SEMIS
*
*** LOOP ***            [ IMMEDIATE word ]
       DATA L1118
L1119  DATA >C44C,>4F4F,>50A0
_LOOP  DATA DOCOL,QCOMP,THREE,QPAIRS,COMPIL,PLOOP
       DATA BACK,SEMIS
*
*** +LOOP ***           [ IMMEDIATE word ]
       DATA L1119
L111A  DATA >C52B,>4C4F,>4FD0
PLLOOP DATA DOCOL,QCOMP,THREE,QPAIRS,COMPIL,PPLOOP
       DATA BACK,SEMIS
*
*** UNTIL ***           [ IMMEDIATE word ]
       DATA L111A
L111B  DATA >C555,>4E54,>49CC
_UNTIL DATA DOCOL,QCOMP,ONE,QPAIRS,COMPIL,ZBRAN
       DATA BACK,SEMIS
*
*** END ***             [ IMMEDIATE word ]
       DATA L111B
L111C  DATA >C345,>4EC4
_END   DATA DOCOL,_UNTIL,SEMIS
*
*** AGAIN ***           [ IMMEDIATE word ]
       DATA L111C
L111D  DATA >C541,>4741,>49CE
_AGAIN DATA DOCOL,QCOMP,ONE,QPAIRS,COMPIL,BRANCH
       DATA BACK,SEMIS
*
*** REPEAT ***          [ IMMEDIATE word ]
       DATA L111D
L111E  DATA >C652,>4550,>4541,>54A0
_RPT   DATA DOCOL,QCOMP,TOR,TOR,_AGAIN,FROMR,FROMR
       DATA TWOM,_ENDIF,SEMIS
*
*** IF ***              [ IMMEDIATE word ]
       DATA L111E
L111F  DATA >C249,>46A0
_IF    DATA DOCOL,QCOMP,COMPIL,ZBRAN,HERE,ZERO
       DATA COMMA,TWO,SEMIS
*
*** ELSE ***            [ IMMEDIATE word ]
       DATA L111F
L1120  DATA >C445,>4C53,>45A0
_ELSE  DATA DOCOL,QCOMP,TWO,QPAIRS,COMPIL,BRANCH
       DATA HERE,ZERO,COMMA,SWAP,TWO,_ENDIF,TWO
       DATA SEMIS
*
*** WHILE ***           [ IMMEDIATE word ]
       DATA L1120
L1121  DATA >C557,>4849,>4CC5
_WHILE DATA DOCOL,_IF,TWOP,SEMIS
*
```

```
*** CASE ***            [ IMMEDIATE word ]
       DATA L1121
L1122  DATA >C443,>4153,>45A0
_CASE  DATA DOCOL,QCOMP,CSP,AT,STRCSP,LIT,>4,SEMIS
*
*** OF ***              [ IMMEDIATE word ]
       DATA L1122
L1123  DATA >C24F,>46A0
_OF    DATA DOCOL,LIT,>4,QPAIRS,COMPIL,POF,HERE
       DATA ZERO,COMMA,LIT,>5,SEMIS
*
*** ENDOF ***           [ IMMEDIATE word ]
       DATA L1123
L1124  DATA >C545,>4E44,>4FC6
_ENDOF DATA DOCOL,LIT,>5,QPAIRS,COMPIL,BRANCH,HERE
       DATA ZERO,COMMA,SWAP,TWO,_ENDIF,LIT,>4,SEMIS
*
*** ENDCASE ***         [ IMMEDIATE word ]
       DATA L1124
L1125  DATA >C745,>4E44,>4341,>53C5
ENDCAS DATA DOCOL,LIT,>4,QPAIRS,COMPIL,DROP
L1126  DATA SPAT,CSP,AT,EQUAL,ZEQU,ZBRAN,L1127-$
       DATA TWO,_ENDIF,BRANCH,L1126-$
L1127  DATA CSP,STORE,SEMIS
*
*** BASE->R ***
       DATA L1125
L1128  DATA >8742,>4153,>452D,>3ED2
BASTOR DATA DOCOL,FROMR,BASE,AT,TOR,TOR,SEMIS
*
*** R->BASE ***
       DATA L1128
L1129  DATA >8752,>2D3E,>4241,>53C5
RTOBAS DATA DOCOL,FROMR,FROMR,BASE,STORE,TOR,SEMIS
*
*
*** L/SCR ***
       DATA L1129
L112A  DATA >854C,>2F53,>43D2
LPSCR  DATA DOCOL,BSLBUF,CSL,DDIV
       DATA SEMIS
*
*** PAUSE ***
       DATA L112A
L112AX DATA >8550,>4155,>53C5
PAUSE  DATA DOCOL,QKEY,DUP,TWO,EQUAL
       DATA ZBRAN,PAUSE1-$,DROP,ONE,BRANCH,PAUSE2-$
PAUSE1 DATA ZBRAN,PAUSE3-$
PAUSE4 DATA QKEY,ZEQU,ZBRAN,PAUSE4-$
PAUSE5 DATA QKEY,DDUP,ZBRAN,PAUSE5-$
       DATA TWO,EQUAL,ZBRAN,PAUSE6-$
       DATA ONE,BRANCH,PAUSE7-$
PAUSE6 DATA QKEY,ZEQU,ZBRAN,PAUSE6-$,ZERO
PAUSE7 DATA BRANCH,PAUSE2-$
PAUSE3 DATA ZERO
PAUSE2 DATA SEMIS
*
*** LIST ***
       DATA L112AX
L112B  DATA >844C,>4953,>54A0
LIST   DATA DOCOL,BASTOR,DECIMA,CR,DUP,SCR,STORE
       DATA PTYPE,>0742,>4C4F,>434B,>2023,DOT,LPSCR,ZERO
       DATA PDO
L112C  DATA CR,I,THREE,DOTR,SPACE,I,SCR,AT,DOTLN
       DATA PAUSE,ZBRAN,L112CX-$,LEAVE
L112CX DATA PLOOP,L112C-$,CR,RTOBAS,SEMIS
```

```
*
*** <BUILDS ***
       DATA L112B
L1139  DATA >873C,>4255,>494C,>44D3
BUILDS DATA DOCOL,CREATE,SMUDGE,SEMIS
*
*** (DOES>) ***
       DATA L1139
L113A  DATA >8728,>444F,>4553,>3EA9
PDOES  DATA DOCOL,FROMR,LATEST,PFA,CFA,STORE,SEMIS
*
*** DOES> ***           [ IMMEDIATE word ]
       DATA L113A
L113B  DATA >C544,>4F45,>53BE
DOES   DATA DOCOL,LIT,PDOES,COMMA,LIT,>6A0,COMMA
       DATA LIT,DODOES,COMMA,SEMIS
*
*** CONSTANT ***
       DATA L113B
L113C  DATA >8843,>4F4E,>5354,>414E,>54A0
CONSTA DATA DOCOL,BUILDS,COMMA
DOCON  EQU  $+2
       DATA PDOES
       DATA >6A0,DODOES     same as ' BL @DODOES '
       DATA AT,SEMIS
*
*** USER ***
       DATA L113C
L113D  DATA >8455,>5345,>52A0
USER   DATA DOCOL,BUILDS,COMMA
DOUSER EQU  $+2
       DATA PDOES,>6A0,DODOES,AT,UU,PLUS,SEMIS
*
*** VARIABLE ***
       DATA L113D
L113E  DATA >8856,>4152,>4941,>424C,>45A0
VARIAB DATA DOCOL,BUILDS,COMMA
DOVAR  EQU  $+2
       DATA PDOES,>6A0,DODOES,SEMIS
*
*** VOCABULARY ***
       DATA L113E
L113F  DATA >8A56,>4F43,>4142,>554C,>4152,>59A0
VOCABU DATA DOCOL,BUILDS,CURREN,AT,TWOP,COMMA,LIT
       DATA >81A0,COMMA,HERE,VLINK,AT,COMMA
       DATA VLINK,STORE
DOVOC  EQU  $+2
       DATA PDOES,>6A0,DODOES,CONTEX,STORE,SEMIS
*
*** (;CODE) ***
       DATA L113F
L1140  DATA >8728,>3B43,>4F44,>45A9
PSCODE DATA DOCOL,FROMR,LATEST,PFA,CFA,STORE,SEMIS
*
*** MYSELF ***          [ IMMEDIATE word ]
       DATA L1140
L1144  DATA >C64D,>5953,>454C,>46A0
MYSELF DATA DOCOL,LATEST,PFA,CFA,COMMA,SEMIS
*
*
*** ~ ***               [ IMMEDIATE word ]
       DATA L1144
L1145  DATA >C180
_NULL  DATA DOCOL,BLK,AT,ZBRAN,L1146-$,QEXEC
L1146  DATA FROMR,DROP,SEMIS
*
```

```
*** NOP ***
       DATA L1145
L1166  DATA >834E,>4FD0
_NOP   DATA DOCOL,SEMIS
*
*** BLOAD ***
       DATA L1166
L1166X DATA >8542,>4C4F,>41C4
BLOAD  DATA DOCOL
BLOAD1 DATA DUP,ONEP,SWAP,BLOCK
       DATA DUP,LIT,14,PLUS,AT,LIT,29801,EQUAL
       DATA ZBRAN,BLOAD2-$,DUP,AT,TOR
       DATA TWOP,DUP,AT,DUP,TOR,DP,STORE
       DATA TWOP,DUP,AT,CURREN,STORE
       DATA TWOP,DUP,AT,CURREN,AT,STORE
       DATA TWOP,DUP,AT,CONTEX,STORE
       DATA TWOP,DUP,AT,CONTEX,AT,STORE
       DATA TWOP,DUP,AT,VLINK,STORE
       DATA LIT,12,PLUS,FROMR,FROMR,SWAP
       DATA OVER,SUB,DUP,TOR,LIT,1000,MIN
       DATA CMOVE,FROMR,LIT,1001,LESS,BRANCH,BLOAD3-$
BLOAD2 DATA DROP,DROP,ZERO,ONE
BLOAD3 DATA ZBRAN,BLOAD1-$,ZEQU,SEMIS
*
*** COLD ***    >>>> Perhaps should change to reload true lowercase <<<<
       DATA L1166X
L1167  DATA >8443,>4F4C,>44A0
COLD   DATA DOCOL,UCONS$,AT,U0,AT,LIT,ULNGTH,CMOVE
       DATA LIT,$TASK0,LIT,$TASK1,OVER,SUB,TOR
       DATA HERE,RR,CMOVE,HERE,TWOP,DUP,LIT,FORTHV,STORE
       DATA FENCE,STORE
       DATA LIT,ASM002,LIT,ASMV,STORE,LIT,ASML,VLINK,STORE
       DATA FIRST,USE,STORE,FIRST,PREV,STORE,FROMR
       DATA ALLOT,EMPTYB,LIT,>FFFF,DPL,STORE
       DATA BOOT$,ABORT,SEMIS
*
*** BOOT ***
       DATA L1167
BOOTN  DATA >8442,>4F4F,>54A0
BOOT$  DATA DOCOL,SPSTOR,DECIMA,ZERO,ECOUNT
       DATA STORE,FORTH,DEFINI,ZERO,BLK,STORE
       DATA DEFBF,PUB                  set current blocks file to default
       DATA LBRCKT,ONE,LOAD,SEMIS
*
*** SYSTEM ***
       DATA BOOTN
L1168  DATA >8653,>5953,>5445,>4DA0
SYST$  DATA $+2
       MOV  *SP+,TEMP1
       MOV  @$SYS(U),LINK
       BL   *LINK
       B    *NEXT
*
*** vvv below are words added from boot, synonym and code blocks
*
*** SLIT ***
       DATA L1168
S1001  DATA >8453,>4C49,>54A0
SLIT   DATA DOCOL,FROMR,DUP,CAT,ONEP
       DATA ECELLS,OVER,PLUS,TOR,SEMIS
*
*** WLITERAL ***        [ IMMEDIATE word ]
       DATA S1001
S1002  DATA >C857,>4C49,>5445,>5241,>4CA0
WLITER DATA DOCOL,BL,STATE,AT,ZBRAN,S1002A-$
       DATA COMPIL,SLIT,WORD,HERE,CAT,ONEP,ECELLS
       DATA ALLOT,BRANCH,S1002B-$
S1002A DATA WORD,HERE
S1002B DATA SEMIS
*
*** <CLOAD> ***
       DATA S1002
S1003  DATA >873C,>434C,>4F41,>44BE
LCLOAD DATA DOCOL,CONTEX,AT,AT,PFIND,ZBRAN,S1003B-$
       DATA DROP,DROP,ZEQU,ZBRAN,S1003A-$
       DATA BLK,AT,ZBRAN,S1003A-$
       DATA FROMR,DROP,FROMR,DROP
S1003A DATA BRANCH,S1003C-$
S1003B DATA DDUP,ZBRAN,S1003C-$,LOAD
S1003C DATA SEMIS
*
*** CLOAD ***           [ IMMEDIATE word ]
       DATA S1003
S1004  DATA >C543,>4C4F,>41C4
CLOAD  DATA DOCOL,WLITER,STATE,AT,ZBRAN,S1004A-$
       DATA COMPIL,LCLOAD,BRANCH,S1004B-$
S1004A DATA LCLOAD
S1004B DATA SEMIS
*
*** VMOVE ***   move multiple bytes from one VDP location to another
*       ( vsrc vdst cnt --- )
*
       DATA S1004
S1004X DATA >8556,>4D4F,>56C5
VMOVE  DATA $+2
       LIMI 0
       MOV  *SP+,TEMP1          pop cnt to R1
       MOV  *SP+,TEMP3          pop vdst to R3
       ORI  TEMP3,>4000         prepare for VDP write
       MOV  *SP+,TEMP2          pop vsrc to R2
** copy cnt bytes from vsrc to vdst
VMVMOR MOVB @MAINWS+5,@VDPWA    write LSB of VDP read address
       MOVB TEMP2,@VDPWA        write MSB of VDP read address
       INC  TEMP2               next VDP read address
       MOVB @VDPRD,TEMP0        read VDP byte
       MOVB @MAINWS+7,@VDPWA    write LSB of VDP write address
       MOVB TEMP3,@VDPWA        write MSB of VDP write address
       INC  TEMP3               next VDP write address
       MOVB TEMP0,@VDPWD        write VDP byte
       DEC  TEMP1               decrement count
       JNE  VMVMOR              repeat if not done
       LIMI 2
       B    *NEXT
*
*** VSBW ***
       DATA S1004X
S1005  DATA >8456,>5342,>57A0
_VSBW  DATA DOCOL,ZERO,SYST$,SEMIS
*
*** VMBW ***
       DATA S1005
S1006  DATA >8456,>4D42,>57A0
_VMBW  DATA DOCOL,TWO,SYST$,SEMIS
*
*** VSBR ***
       DATA S1006
S1007  DATA >8456,>5342,>52A0
_VSBR  DATA DOCOL,LIT,4,SYST$,SEMIS
*
*** VMBR ***
       DATA S1007
S1008  DATA >8456,>4D42,>52A0
```

```
_VMBR  DATA DOCOL,LIT,6,SYST$,SEMIS
*
*** VWTR ***
       DATA S1008
S1009  DATA >8456,>5754,>52A0
_VWTR  DATA DOCOL,LIT,8,SYST$,SEMIS
*
*** GPLLNK ***
       DATA S1009
S100A  DATA >8647,>504C,>4C4E,>4BA0
GLNK   DATA DOCOL,LIT,10,SYST$,SEMIS
*
*** XMLLNK ***
       DATA S100A
S100B  DATA >8658,>4D4C,>4C4E,>4BA0
XLNK   DATA DOCOL,LIT,12,SYST$,SEMIS
*
*** DSRLNK ***
       DATA S100B
S100C  DATA >8644,>5352,>4C4E,>4BA0
DLNK   DATA DOCOL,LIT,8,LIT,14,SYST$,SEMIS
*
*** CLS ***
       DATA S100C
S100D  DATA >8343,>4CD3
CLS    DATA DOCOL,LIT,16,SYST$,SEMIS
*
*** VFILL ***
       DATA S100D
S100E  DATA >8556,>4649,>4CCC
VFILL  DATA DOCOL,LIT,20,SYST$,SEMIS
*
*** VAND ***
       DATA S100E
S100F  DATA >8456,>414E,>44A0
VAND   DATA DOCOL,LIT,22,SYST$,SEMIS
*
*** VOR ***
       DATA S100F
S1010  DATA >8356,>4FD2
VOR    DATA DOCOL,LIT,24,SYST$,SEMIS
*
*** VXOR ***
       DATA S1010
S1011  DATA >8456,>584F,>52A0
VXOR   DATA DOCOL,LIT,26,SYST$,SEMIS
*
*** MON ***
       DATA S1011
S1012  DATA >834D,>4FCE
MON    DATA $+2
       CLR  @>83C4
       BLWP @0000
*
*** random number generator routine ***
_RNDW  LI   TEMP0,>6FE5
       MPY  @>83C0,TEMP0
       AI   TEMP1,>7AB9
       SRC  TEMP1,5
       MOV  TEMP1,@>83C0
       B    *LINK
*
*** RNDW ***
       DATA S1012
S1013  DATA >8452,>4E44,>57A0
RNDW   DATA $+2
```

```
       BL   @_RNDW      get random number into TEMP1
       DECT SP
       MOV  TEMP1,*SP   get random number to stack
       B    *NEXT
*
*** RND ***
       DATA S1013
S1014  DATA >8352,>4EC4
RND    DATA $+2
       BL   @_RNDW      get random number into TEMP1
       ABS  TEMP1
       CLR  TEMP0       set up for division
       DIV  *SP,TEMP0   divide number in TEMP0--TEMP1 by num on stack
       MOV  TEMP1,*SP   return remainder on stack
       B    *NEXT
*
*** SEED ***
       DATA S1014
S1015  DATA >8453,>4545,>44A0
SEED   DATA $+2
       MOV  *SP+,@>83C0     pop and store new seed
       B    *NEXT
*
*** RANDOMIZE ***   increments a counter until VDP interrupt detected
       DATA S1015
S1016  DATA >8952,>414E,>444F,>4D49,>5AC5
RNDMZ  DATA $+2
       MOVB @>8802,TEMP0    get VDP status byte
       CLR  TEMP0           discard it
       CLR  TEMP1           clear counter
S1016A INC  TEMP1           increment counter
       MOVB @>8802,TEMP0    get VDP status byte
       ANDI TEMP0,>8000     VDP interrupt?
       JEQ  S1016A          no, increment counter
       MOV  TEMP1,@>83C0    yes, store new seed
       B    *NEXT
*
*** ASSEMBLER ***       [ IMMEDIATE word ]
       DATA S1016
S1017  DATA >C941,>5353,>454D,>424C,>45D2
ASMV   EQU  $+2         vocabulary link field
ASML   EQU  $+6         chronological link field
ASSM   DATA DOVOC,ASM002,>81A0,FORTHL   <--ASMV initially points to last word in
*                                       ...ASSEMBLER vocabulary in the kernel
*
*** CODE ***
       DATA S1017
S1018  DATA >8443,>4F44,>45A0
CODE   DATA DOCOL,QEXEC,CREATE,SMUDGE,LATEST,PFA
       DATA DUP,CFA,STORE,LBRCKT,ASSM,SEMIS
*
*** ASM: ***    synonym for CODE
       DATA S1018
S1018A DATA >8441,>534D,>3AA0
ASMCOL DATA DOCOL,CODE,SEMIS
*
*** ;CODE ***           [ IMMEDIATE word ]
       DATA S1018A
S1019  DATA >C53B,>434F,>44C5
SCODE  DATA DOCOL,QCSP,COMPIL,PSCODE
       DATA SMUDGE,LBRCKT,ASSM,SEMIS
*
*** DOES>ASM: ***   a synonym for ;CODE     [ IMMEDIATE word ]
       DATA S1019
S1019A DATA >C944,>4F45,>533E,>4153,>4DBA
DOESAS DATA DOCOL,SCODE,SEMIS
```

```
*
*** ^^^ above are words added from boot, synonym and code blocks
*** vvv below are the only 2 words in the kernel that are in the ASSEMBLER vocabulary
*
*** NEXT, ***   1st word in ASSEMBLER vocabulary
       DATA FORTHP          <--points to PNF of FORTH
ASM001 DATA >854E,>4558,>54AC
NEXTC  DATA $+2
NEXTP  LI   TEMP0,>045F
       MOV  @>12(U),TEMP1
       MOV  TEMP0,*TEMP1+
       MOV  TEMP1,@>12(U)
       MOV  @>4A(U),@>48(U)
       B    *NEXT
*
*** ;ASM ***    2nd and last word in ASSEMBLER vocabulary; points to NEXT,
*               ...pointed to by ASSEMBLER as the last word defined in the
*               ...ASSEMBLER vocabulary in the kernel
       DATA ASM001
ASM002 DATA >843B,>4153,>4DA0
SASM   DATA $+2
       JMP  NEXTP
*
*** ^^^ above are the only 2 words in the kernel that are in the ASSEMBLER vocabulary
*
*** DEPTH ***
       DATA S1019A
S1020  DATA >8544,>4550,>54C8
DEPTH  DATA $+2
       MOV  @$S0(U),TEMP0   get stack base
       S    SP,TEMP0        subtract current stack location from base
       SRA  TEMP0,1         divide by 2 to get #cells
       DECT SP              reserve stack space for return
       MOV  TEMP0,*SP       push depth to stack
       B    *NEXT           return to address interpreter
*
*** FILES ***  expects on the stack the maximum number
***            of simultaneously open files
*** maybe should make this ALC!!!
*
       DATA S1020
S1021  DATA >8546,>494C,>45D3
FILES  DATA DOCOL,ONE,PABS,AT,_VSBW,LIT,>016,PABS,AT,ONEP
       DATA _VSBW,LIT,>834C,CSTORE,PABS,AT,LIT,>8356,STORE
       DATA LIT,>0A,LIT,>0E,SYST$,SEMIS
*
*** SCREEN ***  change foreground & background colors in text mode
*               ...background color only in other modes
       DATA S1021
S1022  DATA >8653,>4352,>4545,>4EA0
       DATA DOCOL,LIT,7,_VWTR,SEMIS
*
*** .S ***  non-destructively print parameter stack
       DATA S1022
S1023  DATA >822E,>53A0
       DATA DOCOL,CR,SPAT,TWOM,S0,AT,TWOM,PTYPE,>027C,>2020
       DATA OVER,OVER,EQUAL,ZBRAN,S1023A-$
       DATA DROP,DROP,BRANCH,S1023C-$
S1023A DATA PDO
S1023B DATA I,AT,UDOT,LIT,-2,PPLOOP,S1023B-$
S1023C DATA SEMIS
*
$TASK0 EQU  $
*
*** TASK ***
       DATA S1023
```

```
L1169  DATA >8454,>4153,>4BA0
TASK   DATA DOCOL,SEMIS

$TASK1 EQU  $
DPBASE EQU  $+14
*
       END BOOT
```