COLORFORTH™
V 2.0

FORTH™ LANGUAGE COMPILER

FOR
TRS-80® COLOR COMPUTER
OR
TDP® SYSTEM 100

16K THRU 64K
CASSETTE or RS/DISK

ARMADILLO INT'L SOFTWARE

TABLE of CONTENTS

COLORFORTH Version 2.0

FORTH for the TRS-80 Color Computer

INTRODUCTION

Congratulations on purchasing version 2.0 of COLORFORTH from Armadillo Int'l Software. COLORFORTH is a natural for the Color Computer. It is more than just a language; it is a complete environment, allowing you to write, test, and run applications (programs). It spans the entire range from high level language down to low level machine access. It gives you full control over the hardware on one hand and structured programming constructs on the other. At the same time, it excells in speed, compactness, and modularity.

The purpose of this manual is to explain how to use COLORFORTH and how to take advantage of its unique features. Its purpose is not to teach you FORTH. It includes two glossaries. One is a glossary of the words we have added or modified. The other is the figFORTH glossary. This manual by itself will probably not be enough for someone brand new to FORTH, but following through it step by step, typing the examples as you go, will at least serve as an introduction to the language. If you are new to FORTH, you should get a copy of one or both of the following books. The first is FORTH PROGRAMMING by Leo Scanlon. It contains many examples and sample programs, and carefully delineates between figFORTH (the model COLORFORTH follows) and FORTH-79. The second book is STARTING FORTH, by Leo Brodie. This is an excellent book as well, and contains many illustrations and discusses many of the finer points of this language. Both these books are available locally or direct from Armadillo Int'l Software. We also recommend joining the FORTH INTEREST GROUP. Write them at Box 1105, San Carlos, CA 94070, for their list of publications.

COLORFORTH is an implementation of the F.I.G. model. We have entirely rewritten all of the 6809 machine language primitives and these are proprietary to us. Most of the high level words are taken directly from the public domain figFORTH model. We have customized the system with additional high level words to make it even more convenient to use with the Color Computer.

Here are some of the special features:

1. All 64K of RAM (in 64K machines) can be accessed.
2. I/O words are vectored for more flexibility.
3. DEFER and IS create and point vectored words.
4. The memory map has been set up so COLORFORTH version 2.0 will run in any size machine (16K, 32K, or 64K) without changes.
5. FREEZE updates the cold start values so the current dictionary can be saved to tape or disk.
6. LIST has two versions: the regular version and a much faster version.
7. TITLE lets you print a title on your listings.
8. Ending machine language definitions has been made more convenient with the words NEXT, PUTD, and PUSHD,.
9. Cassette version simulated disk screens do not go through a buffer, but are accessed directly.
10. CSAVEM command is available without requiring Extended Basic.

These changes are discussed at greater length in the following pages.

SYSTEM REQUIREMENTS

COLORFORTH will work with either a cassette or with a disk. It requires a minimum of 16K of memory. It will work in any configuration of Color Computer from 16K through 64K. It does not require Extended Basic, and will work with Basic ROMs 1.0, 1.1, or 1.2. The cassette version WILL NOT work with the disk controller plugged in. The disk version WILL work in a cassette only system, except for the disk I/O. For the disk I/O to work requires the Radio Shack disk controller and Extended Basic 1.0 or 1.1 and Disk Basic 1.0 or 1.1.

DISTRIBUTION

COLORFORTH is distributed on a cassette tape. We call it the "distribution tape." Both the disk version and the cassette version are recorded on the distribution tape. And, each version is recorded twice, in case one copy fails to load. The only thing you should use the distribution tape for is making backup and working copies. Transfer the cassette version (from the distribution tape) to another cassette. Transfer the disk version (from the distribution tape) to a disk. Then, put away the distribution tape where it will be safe and use just the working copy.

STARTING THE PROGRAM and MAKING A BACKUP COPY

To get started with the system, the very first thing you should do is make a back-up copy of the cassette. To do this, make sure the Disk Controller is NOT plugged in (the cassette version "CLRFTH-C" cannot be used while the Disk Controller is plugged in), turn your computer on, put the distribution tape in the recorder, set it to Rewind, and type:

```
CLOADM"CLRFTH-C" <ENTER> ( for cassette version)
   or
CLOADM"CLRFTH-D" <ENTER> ( for disk version)
```

This will start the cassette's motor and allow the tape to rewind. Switch the cassette unit to Play. The cassette (or disk) version should then load without any trouble. It will take about a minute to load. After it loads, type:

```
EXEC <ENTER>
```

You will now be in COLORFORTH, and this message should appear on the screen:

```
COLORFORTH VERSION 2.0
(C)1983 ARMADILLO INT'L SOFTWARE
```

Get another cassette tape (blank) and rewind it. Set up the unit to record. Your tapes will load more reliably if you don't start recording at the very beginning of the tape, so skip over the leader and a few seconds into the tape by typing:

```
MON  <ENTER>  ( to get into BASIC)
MOTOR ON  <ENTER>
( wait a few seconds)
MOTOR OFF <ENTER>
EXEC  <ENTER>  ( to back into COLORFORTH)
```

Now, type in the following:

```
HEX 700 700 HERE CSAVEM <ENTER>  (for cassette version)
```

or

```
HEX 1200 1200 HERE CSAVEM <ENTER> (for disk version)
```

The computer will return with:

```
NAME?
```

Type in the word CLRFTH-C or CLRFTH-D. (NOTE: Be certain your recorder is set to RECORD before you type in the name as the recorder will start as soon as the 8th letter of the name is typed in. Do not use quotation marks.)

When COLORFORTH returns with an OK, your back-up copy will be on that new tape. Now, just to be on the safe side, without touching the recorder, type that same line (HEX 700 700 HERE CSAVEM or HEX 1200 1200 HERE CSAVEM) again to make a second back-up copy immediately following the first one. Now to see if it worked, rewind the tape, turn the computer off and then on again (we call this a "master reset"), set the tape unit to play, and type:

CLOADM"CLRFTH-C" <ENTER>

or

CLOADM"CLRFTH-D" <ENTER>

to load COLORFORTH from the copy you just made. If it doesn't load, try again and/or use a different tape for the back-up, etc. until it works. Don't do anything further until you can make a good back-up tape.

Make a back-up on tape of both versions (CLRFTH-C and CLRFTH-D). Then put away the distribution tape where it will be safe. Don't use the distribution tape for any purpose except making a backup or working copy. If you damage your working copy you can make up another working copy from the distribution tape. To put a working copy on a disk (disk version only, of course) see the instructions in the next section.

HOW TO PUT COLORFORTH ON THE DISK

Be certain you have made your back-up copy. (See previous section.) First, get a blank diskette and initialize it with BASIC's "DSKINI0". Then load COLORFORTH (Disk version) into the computer from the cassette as described in page 4.

When you have completed this, you must format the disk for use by FORTH. Assuming you are in FORTH, type:

FORMAT <ENTER>

This sets up the diskette directory to reserve 60 granules for 135 FORTH disk screens, and leaves 8 granules on the disk to store anything else. To store COLORFORTH there, you need to know the starting and ending addresses of COLORFORTH. The starting address is 4608 (1200 hex). That is also the entry address. To find the ending address, type:

DECIMAL HERE . <ENTER>

Make a note of this value, then get back into BASIC by typing:

MON <ENTER>

and save COLORFORTH to disk by typing:

SAVEM"COLORFTH",4608, <ending address>,4608 <ENTER>

To get back into FORTH, just type:

EXEC <ENTER>

EXAMPLES

This section is designed to introduce beginners to COLORFORTH. Don't just read it; type in the examples. You type in everything after "YOU:" and the computer will answer you with the lines that begin with "CC:". Note that <ENTER> means that you press the ENTER key. Be sure to use upper case. (normal rather than reversed letters). Spaces are very important in FORTH. When several numbers or words are typed on the same line they MUST be separated by one or more spaces. Here we go with some examples:

```
YOU:  7 <ENTER>
CB:   OK
```

(you just put the number 7 on the parameter stack)

```
YOU:  . <ENTER>
CC:   7 OK
```

(you just printed and removed the top number from the stack)

```
YOU:  7 5 12 15 173 <ENTER>
CC:   OK
```

(you just entered those numbers onto the stack - the number 173 is on top)

```
YOU:  . <ENTER>
CC:   173 OK
```

(you printed the top number on the stack - now the number 15 is on top of the stack)

```
YOU:  . . . <ENTER>
CC:   15 12 5 OK
```

(you printed the next three numbers that were on the stack)

```
YOU:  3 5 + . <ENTER>
CC:   8 OK
```

(you put the numbers 3 and 5 on the stack and added them together - which removed the 2 and 5 and left their sum - then you printed the top number on the stack with the dot)

```
YOU:  15 5 / . <ENTER>
CC:   3 OK
```

(15 divided by 5 = 3)

```
YOU: 15 5 * . <ENTER>
CC:  75 OK
```

(15 times 5 is 75)

```
YOU:  12 7 - . <ENTER>
CC:   5 OK
```

(7 subtracted from 12 is 5)

```
YOU:  7 12 - . <ENTER>
CC:   -5 OK
```

(12 from 7 is -5)

```
YOU:    2 3 SWAP . . <ENTER>
CC:   2 3 OK
```

```
YOU:  2 3 DUP . . . <ENTER>
CC:   3 3 2 OK
```

```
YOU:  ." HELLO THERE" <ENTER>
CC:   HELLO THERE OK
```

(the FORTH word ." must be followed by a space)

```
YOU:  : SILLY ." THIS IS ONLY A SILLY TEST" ; <ENTER>
CC:   OK
```

(you just defined a brand new FORTH word whose name is SILLY. The ":" followed by a space marks the beginning of a new word and this is followed by its name. Then, you enter all the FORTH commands (words) that you want your new word to do when it is executed - then you end the definition with a ";")

```
YOU:  SILLY <ENTER>
CC:   THIS IS ONLY A SILLY TEST OK
```

```
YOU:  : TST  0 DO I . LOOP ; <ENTER>
CC:   OK
```

(you defined a new word TST that will count from zero to limit. You supply the limit when you execute TST)

```
YOU:  3 TST <ENTER>
CC:   0 1 2 OK
```

```
YOU:  5 TST <ENTER>
CC:   0 1 2 3 4 OK
```

```
YOU:  : ADD3 + + ; <ENTER>
CC:   OK
```

(you just defined a new word that will add 3 numbers on the stack and leave only their sum)

```
YOU:  7 8 9 ADD3 . <ENTER>
CC:   24 OK
```

(you first gave it the three numbers it needed by putting them on the stack - then you printed out the final result with the dot.)

See the glossary for the complete list of FORTH words available.

Disk Screens and EDITOR

FORTH organizes data on a disk (or simulated disk) in groups of 1024 bytes called "SCREENS". The disk screens are numbered from 0 through 134. The simulated disk screens reside in memory and start with number 1, not 0.

You can store 135 FORTH screens on a disk, leaving 8 granules on the disk to store anything else you want. You will probably want to store a copy of COLORFORTH there.

While the FORTH screens can be used for many purposes (input data to FORTH programs, output data from FORTH programs, lists of error messages, etc.), their usual use is for storing SOURCE CODE that you write. If, instead of typing in new definitions directly at the keyboard, you will first put them onto a FORTH screen, it will let you (1) refer to your source code later so you can see how you defined a particular word and (2) make changes easily (with the editor) rather than typing everything in again from the keyboard.

It is easy to use the screens. The EDITOR helps you work with them. The EDITOR is resident - which means it is in memory at all times to make it quick and easy to use. To get into the EDITOR, all you have to do is type:

```
EMPTY-BUFFERS <ENTER>

EDITOR <ENTER>
```

EMPTY-BUFFERS only has to be typed once per session. (If you want to get out of the EDITOR all you have to do is type:

```
FORTH <ENTER>)
```

So, get into the editor and then inspect the first screen by typing:

```
1 LIST <ENTER>
```

The contents of screen #1 will pour "forth" onto your screen. If you are listing a screen that has not had useful data put on it, you will only see garbage. Don't worry about this, just clear the screen by typing:

```
1 CLEAR <ENTER>
```

Since you are still in EDITOR, you can relist the screen by typing:

```
L <ENTER>
```

This is probably a good place to tell you about a

special feature of COLORFORTH. As you can see, the screen rolls by too fast to be useful. To stop the listing, merely press any key (except BREAK). To restart, press any key again. If you press BREAK, you will exit the listing command.

Okay, the screen should look a lot neater now. Look in the glossary in the section on the EDITOR for more information on using it, but try out the following short examples for a quick introduction. Type:

```
1 P ( THIS IS HOW TO PUT TEXT ON LINE 1) <ENTER>

2 P : TEST CR ." HELLO, THERE !" CR ; <ENTER>

3 P ;S <ENTER>
```

That is how you put new data to the screen... using the EDITOR's command "P" (it stands for "put"). Now list out your screen by typing:

```
L <ENTER>
```

WARNING - this screen is now stored only in RAM and has not been copied back to disk. To get the changed version of the screen back to the disk, type:

```
FLUSH <ENTER>
```

You only have to type FLUSH at the end of the editing session. During the editing session, you can list and change screen after screen and FORTH will automatically save the changes back to disk, except for the last screen you listed. This last screen gets copied back to the disk when you type FLUSH.

After storing your source code on screen #1, you can LOAD it. This is similar to "compiling" in other languages. When you load screen #1, COLORFORTH treats everything on the screen as if you were just now typing it in from the keyboard. Try it. Type:

```
1 LOAD <ENTER>
```

When COLORFORTH replies with "OK", you now should have a new word in the dictionary named TEST. Try executing that word. Type:

```
TEST <ENTER>
```

and COLORFORTH will reply with:

```
HELLO, THERE !
```

If you get error messages anywhere along the line, it is most likely that you typed something in wrong. Spaces

are very important in FORTH. Did you put the space after the "(" in line #1? How about after the '."' in line #2? Keep trying until it works.

Now try executing it several times in a row. Type:

TEST TEST TEST <ENTER>

and COLORFORTH will reply:

HELLO, THERE !

HELLO, THERE !

HELLO THERE !

To see how you made TEST work, relist screen #1 by typing:

1 LIST <ENTER>

and then get back into the editor by typing:

EDITOR <ENTER>

Let's explain what you see on the screen. Line 0 should be blank because we did not put anything on it. Line 1 is a comment because of the parentheses. Line 2 holds the actual definition of the word TEST. It begins with a ":" and ends with a ";". Line 3's ";S" marks the end of the screen and tells COLORFORTH that you wish it to stop loading from the screen and return to listening to the keyboard.

Now let's modify the screen. Suppose you want to remove the comma after HELLO. Type:

2 T <ENTER>

(T stands for "TYPE.") This will display line 2 with the editing "cursor" at the beginning of the line. Then type:

X , <ENTER>

This will delete the comma, and display the editing cursor at the position where the comma used to be. Note that the X must be followed by exactly one space. It then looks for the "string" you type in and deletes it if it can find it. If it can not find it, you will get an error message, "? MSG #0". We are finished with our changes so let's list the screen again by typing:

L <ENTER>

After listing the screen with L, inspect it to see if it is the way you like it. Let's say you changed your mind and want the word TEST to say "HELLO TO YOU" instead of

"HELLO THERE !". Type:

2 T <ENTER> (to point to the line we want to change.)

X THERE ! <ENTER> (to delete "THERE !")

C TO YOU <ENTER> (to "COPY" in the new text.)

List the screen again to be certain that it reads the way you want it to read. Play around with changing it this way and that until you feel comfortable using the editor. It is really easy as soon as you get the hang of it. Remember that your latest copy is only in RAM and not back onto the disk yet, so when you are finished be certain to use FLUSH.

Now that you have changed your definition of TEST, recompile it by typing:

1 LOAD <ENTER>

Execute TEST to see if it does what you want it to do. When you typed "1 LOAD" you probably got error MSG #4. This is only a warning to let you know that the word TEST has previously been defined. No problem. Although you have both definitions in the dictionary, only the most recent one will be used when you execute "TEST". However, if you want to clean it up, type:

FORTH <ENTER> (to make certain you are not still in the EDITOR)

FORGET TEST <ENTER> (this forgets the most recent version)

FORGET TEST <ENTER> (this forgets the first version)

1 LOAD <ENTER> (this restores the new version)

That's a quick introduction to the EDITOR and almost all there is to it. There are some additional words in the EDITOR. See the EDITOR glossary for more details. Briefly, here are some more examples:

TOP <ENTER> (this positions the editing cursor at the top of the current screen)

F CATFISH <ENTER> (this searches the screen from the current position of the editing cursor looking for the string "CATFISH". If it finds "CATFISH", it will display that line and position the editing cursor just after it. If "F" cannot find the string "CATFISH" on the screen it will give a MSG #0 message.)

Suppose it finds "CATFISH". Then type:

B <ENTER> (this backs up the editing cursor to the beginning of the string it just found.)

TILL <string> <ENTER> (this deletes everything from the current editing cursor position to and including the string)

7 D <ENTER> (this deletes line 7)

10 S <ENTER> (this "spreads" the screen at line 10 to open a line up. Line 15 falls off the screen, never to be heard from again)

Try it out. Note that this editor is different from the editor described in the book STARTING FORTH.

SCREEN NUMBERS

You cannot LOAD from disk or simulated disk from block number zero. (Block number zero indicates to FORTH that input is coming from the keyboard.) So, in the disk version, you can LIST and EDIT screen #0, but you cannot LOAD screen #0. In a small cassette system, where all of your screens need to be in memory, there is no room to spare for a screen number that cannot be LOADed, so simulated disk screens (in either the cassette or disk versions) have no screen #0 at all.

SIMULATED DISK SCREENS

In the cassette version there are no "disk" buffers. Simulated disk screens are accessed directly. There is no particular point in moving the simulated screen (in memory) to another place in memory called a "disk" buffer. (With a disk, there IS a point in moving it from the actual disk into memory). This means that the words EMPTY-BUFFERS FLUSH and UPDATE really have no use in the cassette version. When you use the EDITOR on the simulated disk screen, the actual screen image is altered immediately (rather than altering a copy in the disk buffer). Thus there is no need for FLUSH to copy the buffer back to the screen and no ability to undo any changes before it gets written back to the screen with EMPTY-BUFFERS and no need to mark anything with UPDATE. This will give you faster LOADING and editing (but it was already fast enough that you probably won't notice a difference.) Those three newly useless words are still present in the cassette version for people who are used to typing them, and for compatibility with the disk version. However, in the cassette version they do nothing.

Note, the above only applies to the cassette version. In the disk version you can still use simulated disk screens but they DO go through the regular disk buffers. So, if you are using a cassette based system and do not like our having eliminated the "disk" buffers, just load in the disk version and use it as a cassette FORTH. You probably won't want to do that in a 16K system because you won't have as much room in memory for your simulated screens.

The constants LO and HI define the area used for simulated disk screens. Feel free to move these around to take advantage of the memory you have. For example, to move them up and expand them for a 64K system, you could type

```
HEX  5000 ' LO !  ( give more room for the)
                  ( dictionary to grow into)
     FBFF ' HI !  ( to give you lots of screens)
```

The area set aside for simulated screens should be a multiple of 1024 ($0400) bytes, which is the standard length of one screen. In the cassette version, HI should be 1 greater than the last byte in the simulated disk area. In the disk version HI should be the last byte in the simulated disk area.

Cassette Versus Disk

COLORFORTH (disk version) will work with either actual or simulated disk screens. Type: CAS <ENTER> to use simulated disk screens, or type: DISK <ENTER> to use actual disk screens.

More Than One Drive?

If you have a second disk drive, you can switch between the two by typing:

DR0 <ENTER>

or

DR1 <ENTER>

HOW TO SAVE SIMULATED DISK SCREENS
(Both Cassette & Disk Versions)

The problem with saving your source code on a simulated disk screen instead of a real disk screen is that when you turn off the power (or a program runs amok) the source code disappears. A way to save it is to store it on cassette. The following will save the entire simulated disk area:

Cassette version

HEX LO 704 HI CSAVEM <ENTER>

NAME? <invented name> <ENTER>

Disk version

HEX LO 1204 HI CSAVEM <ENTER>

NAME? <invented name>

The screens can then be reloaded from either FORTH or BASIC. If you are in FORTH, type:

EMPTY-BUFFERS <ENTER>

CLOADM <ENTER> (and the tape will load)

If loading from BASIC, type:

CLOADM <ENTER> (and the screens will load)

EXEC <ENTER> (to get back into FORTH)

EMPTY-BUFFERS <ENTER>

You can then use EDITOR, FLUSH, and LOAD as before.

BLOCK SIZE

In COLORFORTH the size of a disk block is 256 bytes. Thus, a screen consists of 4 blocks. For example, scr #1 is made up of blocks 4, 5, 6, and 7. Some books say that a FORTH block is 1024 bytes, which is not always correct! See B/BUF and B/SCR in the fig glossary.

FAST and SLOW LISTs

You can choose either the regular LIST or a faster LIST. The regular LIST displays each line with its full 64 characters, including spaces, then ends the line with the symbol "<". The faster LIST leaves off the ending spaces and the symbol "<."

Printing the "<" helps you see where the end of the line is and, if the "<"s are not in line, it shows you that a non-printable character is in your text. If such a character is present, your screens will usually mysteriously fail to load. And, you can't "see" what's wrong because of the non-printing character. Perhaps you hit the break key by mistake. The null ($00) is the most common offender. It can come about if you type C or P followed by a carriage return or by a space and then carriage return while in the EDITOR. The solution to this is to position the cursor to the top of the screen with TOP and then type X followed immediately with a carriage return (while in the EDITOR). This hunts for and deletes the null. So, showing the full line has its uses. The problem is it takes a lot longer to print a FORTH screen this way, and the screen takes up a lot more room on your display. As shipped, COLORFORTH version 2.0 will use the faster version. to switch over type

```
' <SLST> IS LIST  ( slow list)
```

and to switch back, type

```
' <FLST> IS LIST  ( fast list)
```

MACHINE LANGUAGE

If you want to build a new COLORFORTH word using machine language, here is how to do it. First create the header by typing:

CREATE <name> <ENTER>

Then type in your object code two bytes at a time, followed by a comma, until you have typed in the entire routine. End your routine with a jump to "Next" with either NEXT, or PUTD, or PUSHD, and then type in the word SMUDGE. Remember, you should not alter S, U, or Y without saving and restoring it. Here is an example of a machine language routine named ETEST, that loads the B register with the ascii code for the letter "E", and then pushes the value (as the D register) to the stack. The U register is used as the stack pointer, but because of PUSHD, we do not have to directly reference it. Type:

HEX CREATE ETEST C645 , PUSHD, SMUDGE <ENTER>

Type in the above definition of ETEST, then execute it by typing:

ETEST <ENTER>

Did it work? To check it, type:

EMIT <ENTER>

and see if an "E" mysteriously appears on the screen. That is how easy it is to put machine code FORTH words into your system. Note, once the machine language word has been defined it is used thereafter the same as any other FORTH word.

JSR

JSR is included so that you can execute machine language routines that end with a return to subroutine (RTS) instruction (instead of ending with a jump to "Next." Since COLORFORTH uses U for the parameter stack pointer, S for the return stack pointer, and Y for the interpretive pointer, be certain your routine leaves them like it found them. For machine language, " 0,U " points to the "TOP" word on the stack, and " 0,S " points to the "TOP" word on the return stack.

ENDING MACHINE LANGUAGE DEFINITIONS

NEXT NEXT, PUTD, PUSHD,

In COLORFORTH version 1.0 the word NEXT was used to end machine language definitions by laying down the code to do FORTH's inner interpreter routine (named "Next"). In some FORTH systems, NEXT is a constant, the address of the inner interpreter routine. So in those FORTHs you would end your machine language routines with HEX 7E C, (6809 JMP instruction) NEXT , (the address to jump to). In COLORFORTH version 1.0, NEXT was more active than a constant, in that it laid down the full code to do the operation of "Next." Well, all of that has changed slightly in COLORFORTH version 2.0. We now have three possible endings for a machine language routine. These are NEXT, which lays down the code for a jump to "Next" PUTD, which lays down the code for a jump to "PutD" and PUSHD, which lays down the code for a jump to "PushD." Note, that each of the words ends with a comma. This is to indicate that these words actively "comma" information into the dictionary. They all end up eventually at the routine "Next." The differences are that NEXT, does not affect the stack, PUTD, replaces the top item on the stack with the contents of the D register, and PUSHD, pushes the contents of the D register onto the stack. This should make the use of machine language words from COLORFORTH even easier.

COLORFORTH AS A MACHINE LANGUAGE MONITOR

C@ and @ let you fetch one or two bytes at a time. C! and ! let you store numbers into one or two bytes at a time. The words C, and , let you "comma" into the dictionary one or two bytes at a time. DUMP displays any range of memory in both its hexadecimal and ASCII equivalents. FILL and ERASE and BLANKS and CMOVE extend your capability for setting up RAM the way you want it. These words give you many of the features of a machine language monitor, and they are built right in. You don't have to get out of COLORFORTH and load a separate program.

DUMP

Another special feature is our DUMP utility. When dumping to the screen, the address is printed followed by eight bytes as two HEX characters per byte. On the next line, it prints each byte as an ascii character if the byte is a printable character. If it is not a printable character, a period is printed. This format of eight bytes per line makes it readable. However, when you dump to a printer, it would be nice to have a little more on each line. COLORFORTH automatically prints 16 bytes per

line on the printer. If you want to change this, you need to change the value of the variable DLEN. It must be a multiple of 8. Right now, for dumping to the screen, the value is 8, and for dumping to the printer the value is 16. If you wanted to change the value to 24 bytes per line, you would type:

```
DECIMAL 24 DLEN ! <ENTER>
```

This will be a temporary change - good only until you type in P-ON or P-OFF.

PROGRAM CONFIGURATION

MEMORY MAP

In COLORFORTH version 2.0 you do not have to alter anything at all in order to set it up for different size machines (16K vs 32K vs 64K). This is because the stacks, user area, and TIB (terminal input buffer) and disk buffers have all been placed in low memory. Cold and warm start addresses are NOT the same as in COLORFORTH version 1.0. See the memory map.

MEMORY MAP - Cassette Version 2.0
(addresses are in hexadecimal)

```
4000     HI

         Ram simulation of disk screens 1 - 6

2800     LO


         PAD   text buffer floats above dictionary

2541     HERE  initial top of dictionary - grows up from
               here


               dictionary

0722     first word in dictionary

               boot up literals

               warm start entry = 0704
0700     ORIGIN cold start entry = 0700

              user variables

06C0     UP (64 bytes from 06C0 up)
06C0     RETURN STACK  (grows down)


0640     TIB  (grows up)
0640     STACK (grows down for 64 bytes)
```

note, unlike COLORFORTH version 1.0, there are no "disk" buffers in Cassette COLORFORTH version 2.0.

MEMORY MAP - Disk Version 2.0
(addresses are in hexadecimal)

16383 3FFF HI

Ram simulation of disk screens 1 & 2

14336 3800 LO

PAD text buffer floats above dictionary

13259 33CB HERE initial top of dictionary - grows up from here

dictionary

4642 1222 first word in dictionary

boot up literals

warm start entry = 1204
4608 1200 ORIGIN cold start entry = 1200

11E5 LIMIT (first byte past disk buffer)

disk buffers

0DD5 FIRST (first byte of disk buffer)

user variables

0C7E UP
0C7E RETURN STACK (grows down)

0B8A TIB (grows up)
0B8A STACK (grows down)

098A lowest available memory for stack

COLD START VALUES

Cold start (or boot up) literals can be altered and then the COLORFORTH image can be saved to tape or disk. Next time it is loaded back into memory (or when the word COLD is executed), the new values will be used. All of these cold start values are referenced relative to the program origin (cassette=$0700, disk=$1200) with the word +ORIGIN, so you do not need to use actual addresses. The following cold start addresses apply to both the cassette and disk versions:

```
08 +ORIGIN    cpu
0A +ORIGIN    version number
0C +ORIGIN    topmost word in FORTH vocabulary
0E +ORIGIN    backspace character
10 +ORIGIN    holds addr of user variable area
12 +ORIGIN    addr of top of parameter stack
14 +ORIGIN    addr of top of return stack
16 +ORIGIN    addr of TIB
18 +ORIGIN    name field width
1A +ORIGIN    WARNING (cold start value)
1C +ORIGIN    FENCE (cold start value)
1E +ORIGIN    DP (cold start value)
20 +ORIGIN    VOC-LINK (cold start value)
```

At least until you are experienced with FORTH, you will have no need at all to change any of those values. The only exceptions are the cold start values for FENCE and HERE and the topmost word. If you expand the dictionary (by defining your own words) and want those words to be present each time you bring up COLORFORTH, you will need to alter those three cold start values. You could do so by typing

```
FORTH DEFINITIONS HEX
LATEST  0C +ORIGIN  !
HERE    1C +ORIGIN  !
HERE    1E +ORIGIN  !
```

but you don't need to. Instead, type

```
FREEZE
```

and it will do it for you.

How To Save New Definitions
(Cassette Version)

After defining your new words, type:

FREEZE <ENTER>

then, save this version to your cassette tape as described earlier, with

HEX 700 700 HERE CSAVEM <ENTER>

NAME? COLORFTH

How To Save New Definitions
(Disk Version)

After defining your new words, type:

FREEZE <ENTER>

then, save this version to your disk as described earlier,

HEX

HERE . (make a note of this addr)

MON (to get into BASIC)

SAVEM"COLORFTH",&H1200,&HXXXX,&H1200
(where XXXX is the value of HERE)

CSAVEM

COLORFORTH has a built in CSAVEM command. If you don't have Extended Basic, then you probably know that your Color Computer was set up to load machine language programs, but not to save any that you may have written. So now, even if you don't have Extended Basic, you have CSAVEM. You must give it all three arguments in this order: (first address) (entry point) (last address), then type in the command CSAVEM <ENTER>. Be certain you know which number base you are in. Usually, for saving machine language programs, you will want to be in HEX. You might also want to make a note on the cassette label of these addresses in both HEX and DECIMAL. To convert a HEX number to DECIMAL, type:

HEX <number> DECIMAL . <ENTER>

Be certain to leave a space between the word "HEX", the number to be converted, the word "DECIMAL", and the dot. Spaces are very important in FORTH.

CLOADM

COLORFORTH also has the companion command CLOADM. This command does not take any arguments. The program is loaded to the address specified when the tape was created. If you want to load a tape into a different area of memory, you must get back to BASIC and type:

CLOADM"",(offset in decimal) <ENTER>

Note that when you load any tape, the address to which you will automatically jump when you type EXEC, will be set to the entry-point specified on the most recently loaded tape. So, when saving data onto a tape, you might want to set the (entry-point) to HEX 704 (for cassette version) or HEX 1204 (for disk version). These are the warm start entry-points.

PRINTER - P-ON and P-OFF

P-ON and P-OFF are commands included to give you easy access to your printer. For example, if you want to dump 50 (HEX) bytes beginning at address 4000 (HEX) to the printer, you could type:

HEX P-ON 4000 50 DUMP P-OFF <ENTER>

NOTE: If you want to change the value of DLEN and thus the number of bytes per line to be printed, remember, DLEN

is good only until the program encounters P-ON or P-OFF. Therefore, you would have to type something like:

HEX P-ON DECIMAL 24 DLEN ! HEX 4000 50 DUMP P-OFF <ENTER>

?TERMINAL and HALTING the DISPLAY

Another feature you should know about is ?TERMINAL. This word is used in CR, VLIST, and DUMP, and can be used by you in your own words. It checks the terminal to see if you have pressed the BREAK key. If you have, it returns <true>, otherwise, it returns <false>. Thus DUMP and VLIST, etc., can be exited if you have seen enough. This means that if you press any key other than the red BREAK key, everything halts until you press any key again. You can easily control the flow of data onto your screen. For example, instead of dumping 30 (HEX) or so bytes at a time (you would get tired of entering the commands over and over if you wanted to dump 2000 bytes), you can dump the entire ammount at one time and start and stop it when you are ready by hitting any key. The red BREAK key will halt the DUMP, and return you to COLORFORTH.

LEFT and RIGHT BRACKETS

Some FORTH words require the symbols [and] (brackets). To type these, use shifted down arrow and shifted right arrow.

ACCESS TO THE HARDWARE

COLORFORTH makes it easy to talk directly to the Color Computer's hardware. This allows you direct control over the SAM chip (6883), the two parallel port chips (6821 or 6822), and the Video Display Generator (VDG) chip (6847), the disk controller, and RAM. Thus you can control such things as music or tone generation, graphics, serial I/O, A/D and D/A converters, joysticks, the disk controller, etc.

To work with this hardware you need to understand it, at least somewhat. Various books and magazines that have discussed this hardware, Motorola data sheets on the specific chips, and especially the Radio Shack service manual are good places to look for this information. This section is not designed to cover the hardware in detail, but to show you how to use COLORFORTH once you understand the hardware.

The hardware devices are memory mapped, which means that they are addressed as if they were RAM. Here are two examples to give you the general approach to take:

First, let's use COLORFORTH to turn the cassette

motor relay on and off. Bit 3 (numbered 0 through 7) of the 6821 chip parallel port at address hex FF21 controls this relay. If a "one" is stored there, the relay goes on. If a "zero" is stored there, the relay goes off. Set up a tape recorder to play and watch it as you type in the following examples.

```
HEX FF21 C@  08  OR   FF21 C!  <ENTER>

    FF21 C@  F7  AND  FF21 C!  <ENTER>
```

The first line should have made the cassette motor turn on and the second line should have made the cassette motor turn off.

Those lines are too long to type in if we have to do it very often, so let's put them into a new definition.

```
: MOTOR-ON  FF21 C@  08 OR  FF21 C! ;

: MOTOR-OFF FF21 C@  F7 AND  FF21 C! ;
```

Now type MOTOR-ON and MOTOR-OFF over and over while you watch the cassette player until you believe it works, then read on for the explanation.

We read the port first (with FF21 C@). Then we changed only bit 3 (with 08 OR) then we wrote it back (with FF21 C!). Why didn't we just say 08 FF21 C! and then 0 FF21 C! ? Well, we could have; it will work. Try it if you like. The reason we didn't is that some of the other bit positions control things other than the cassette relay and we don't want to disturb them! So, AND and OR are the tricks we use to alter a single bit position with out affecting the other bit positions. The trick is this, ORing a "1" into a bit position forces it on, ORing a "0" into a bit position leaves it unchanged. And, ANDing a "0" into a bit position forces it off, ANDing a "1" into a bit position leaves it unchanged. Look at what we OR'd with FF21 to turn on bit 3. We OR'd a hex 08. This is equivalent to a binary 00001000. Note that counting from the right hand side starting with zero, the only "1" is in bit position number 3. When we AND'd the F7 notice that it is equivalent to binary 11110111 and that the only "0" is also in bit position number 3.

If you understand all of this, great. If you don't, go over it again and again until you do. It is very simple, but if you don't grasp it you will never be able to understand how to handle the hardware.

If all you wanted to do with that port was turn the relay on and off then the above definitions for MOTOR-ON and MOTOR-OFF are entirely satisfactory. Note that once you have defined them, you can forget the hardware details. Thus, COLORFORTH insulates the rest of your

program (in which it is assumed you are going to use MOTOR-ON and MOTOR-OFF) from these nasty details. This is good programming practice.

We could go one step further. Instead of using FF21 over and over, we could even isolate that hardware detail with

```
HEX  FF21  CONSTANT  MOTOR-PORT
```

then we have a symbolic name, MOTOR-PORT, that can be used in place of the number FF21. Using it, we could rewrite our first definition as

```
: MOTOR-ON   MOTOR-PORT C@  08 OR  MOTOR-PORT C!  ;
```

it gives a slight advantage in readability.

Second, let's control the single bit sound output to make your TV speaker sqwawk. This is controlled by bit 1 of the parallel port at address hex $FF22. By turning it on and off rapidly you will make the speaker vibrate at the same rate.

```
HEX  : DELAY    ( n --) 0  DO  LOOP ;
  ( an empty loop to kill time)

: S-ON  FF22 C@  02 OR  FF22 C! ;
: S-OFF FF22 C@  FD AND FF22 C! ;

: SQWAWK  ( pitch -- )
   BEGIN   S-ON  DUP  DELAY
           S-OFF DUP  DELAY
    ?TERMINAL
   UNTIL   DROP  ;
```

Try it with different pitches such as

```
10 SQWAWK
250 SQWAWK
etc.
```

Be sure to put a number on the stack before calling SQWAWK since it expects such a number. The sqwawking should continue until you hit the BREAK key. Hitting any other key should make it pause temporarily.

There! You should have enough information now to control your hardware from COLORFORTH.

GRAPHICS

The low-resolution graphics are the simplest. All that is required is to store a number greater than 127 (decimal) or 7F (hex) into the display RAM. The display RAM for your regular alphanumerics display is hex 400 to 5FF unless you specifically change it. You can get to this memory with C! Try typing the following

```
500 C!  501 C!  502 C!

HEX
: LOW-DEMO  100 80  DO I 500 I + C! LOOP ;
LOW-DEMO
```

or try typing

```
F8 500 C!  D6 501 C!
```

To use the other graphics modes you need to set up both the VDG chip and the SAM chip for the mode you want, set up the starting address for the display memory (you need to move it somewhere where there is room for it, since it could take up to 6K for the highest resolution graphics), and fill the graphics memory with the particular bit patterns you want to display.

OTHER FEATURES

64K

In machines that have 64K, Version 2.0 lets you access all of it. Using the words 64K-ON and 64K-OFF you can pick whether the upper 32K of RAM or the upper 32K of ROM is active. The lower 32K of RAM is always active, so you actually have access to 96K of address space.

If you don't have 64K in your machine yet, call or write us for the name and address of the people we recommend for good, fast service at reasonable rates on memory upgrades and repairs.

To turn on the upper 32K of RAM type

```
64K-ON
```

then any accesses (fetches and stores, etc.) of the upper 32K will refer to the RAM. To switch back to accessing the ROM space, just type

```
64K-OFF
```

Having the upper 32K of RAM available lets the dictionary grow up into this area or lets you use it for simulated disk screens. This area of RAM cannot be saved directly to disk or tape, because to access it, the ROMs have to be OFF, and to write to disk or tape the ROMs have to be ON. However, you can save this area to tape or disk by moving it (with CMOVE) to the lower 32K and saving it from there. Your usual disk and tape I/O can occur with 64K-ON just as long as the source or destination of that I/O is within the lower 32K.

VECTORED WORDS

A new feature in COLORFORTH version 2.0 is the vectoring of the following words:

```
EMIT  KEY  DSKCON  ?TERM  CR  CSCODE  CLOADM
LIST  TITLE  R/W
```

By vectored we mean that you can re-direct those words to any other word you choose. For example, the Color Computer normally sends out only a carriage return to a printer or terminal. For some terminals or printers you might want to send both a carriage return and a line feed. It is simple to do this in COLORFORTH version 2.0 by typing

```
' <CRLF>  IS  CR
```

This points the word CR to the word <CRLF> which sends both the carriage return and line feed codes. Note the tick ("'") preceeding <CRLF>. To switch back to sending only a carriage return, type

```
' <CR>  IS  CR
```

Note that <CR> and <CRLF> are already in COLORFORTH version 2.0's dictionary. You can, however, define your own new word to be used for CR (or any other vectored word). For example, suppose you are printing to a printer that allows special type styles that only remain effective until a carriage return, at which time the type style reverts to normal. Well, define your own version of carriage return that sends the special type style code again right after doing a regular carriage return. Suppose the codes involved are hex 1B 54 (ESC-T). Define your new word as follows

```
HEX : EMPHASIZED-CR  <CR> 1B EMIT  54 EMIT  ;
```

then, when you want to turn on the emphasized printing, type

```
1B EMIT 54 EMIT  '  EMPHASIZED-CR  IS  CR
```

and to go back to the regular carriage return, type

```
' <CR> IS  CR
```

The words 64K-ON and 64K-OFF use this approach to switch between <EMIT> <KEY> <?TERM> <DSKCON> <CSCODE> and <CLOADM> (which are used to access the lower 32K RAM and the upper 32K of ROM) and the words EMIT-64 KEY-64 ?TERM-64 DSK-64 CSM-64 CLM-64 (which are used to access the lower 32K RAM and the upper 32K RAM).

You can define your own vectored words with the word DEFER. Example:

```
DEFER  GRAPES
```

creates a new word, GRAPES. Be sure to use IS to point it somewhere, otherwise, the system will crash when you type GRAPES.

LESS THAN

There was a bug in the figFORTH model for the less than operator ("<"). It has been fixed in COLORFORTH version 2.0.

TITLE

The word TITLE appears within the words INDEX and VLIST to print a title at the end of each full page. TITLE is a vectored word and is initially set up to point to the word NOP (no operation) so it will do nothing. You can define your own title words and vector TITLE to them. You can also use TITLE in other words you define. Here's an example:

```
: TT  CR CR  ." This was printed using COLORFORTH
version 2.0" ;
     ' TT  IS  TITLE
```

NUMBER BASES

FORTH is handy for working in different number bases. The base you are in depends on the value of the variable BASE. When you first crank up COLORFORTH, you are in DECIMAL. To change over to hexadecimal, type:

```
HEX <ENTER>
```

To change to binary, type:

```
2 BASE ! <ENTER>
```

To change to octal, type:

```
DECIMAL 8 BASE ! <ENTER>
```

(You typed in DECIMAL first, because the 8 would not be recognized if you were in binary)

To get back to decimal, type:

```
DECIMAL <ENTER>
```

You can use nearly any base you can think of, such as base 3, base 4, base 23, etc. but binary, octal, decimal, and hexadecimal are the most commonly used ones. If you wish, you can define new words to make it even easier to change to binary or octal by typing:

```
: BINARY 2 BASE ! ; <ENTER>

: OCTAL 8 BASE ! ; <ENTER>
```

Converting numbers from one base to another is very easy. For example, type:

```
DECIMAL 255 HEX . <ENTER>
```

COLORFORTH will respond with:

FF

which is the hexadecimal equivalent of decimal 255.

STACK OVERFLOW

In the cassette version there is no stack overflow message (message #7). If the stack overflows it will begin to fill up the display memory (from $05FF down to $0400), thus it should be fairly obvious. If your application requires a very large stack, it is available (at the cost of an interesting looking display). The disk version has 512 bytes available for its stack and an overflow message.

SPEED

COLORFORTH is considerably faster than BASIC. It is not exactly 35 times faster or 100 times faster or 8 times faster because it depends on what you are doing. Of course, that is just execution speed. In many cases that is not nearly as critical as the speed of program development.

Because of COLORFORTH's modularity you can test each little piece of your application. Write your application in small pieces. Each more advanced word is made from a small group of your previously defined building blocks. COLORFORTH lets you use names for your words that are up to 31 characters in length. CALCULATE-OVERTIME-PAY makes a lot more sense than "GOSUB 3192"! Thus it is easier to think with and debug and understand 2 days or 2 months later.

COLORFORTH's execution speed is probably fast enough that you will have little or no need for using machine language. However, when you need all-out speed, COLORFORTH is as fast as machine language because you can recode the key time-taking parts of your application as a machine language FORTH word. The rest of your application will refer to this definition just as it did before when that word was written in high-level FORTH. So, you can write your application entirely in high-level FORTH for ease of development and testing. Then, only if you need the extra speed, re-code selected words in machine language, without altering the structure of your application. COLORFORTH gives you the best of both worlds.

If you do any speed comparisons between COLORFORTH and other languages (including BASIC) and/or other machines, we are interested in hearing your results.

TURNKEY

Here is an additional COLORFORTH word to add to your version if you wish. Its purpose is to strip out the headers of COLORFORTH, so that you can sell your applications WITHOUT also selling a useable copy of COLORFORTH.

The new word is TURNKEY, and here is how to install it. Type:

```
FORTH DEFINITIONS HEX <ENTER>
: SAVE-IT BEGIN 0 +ORIGIN DUP HERE CSAVEM AGAIN ; <ENTER>
: BYPASS-X ' QUERY NFA ' FILL LFA ! ; <ENTER>
' FILL LFA @ <ENTER>
     DUP CONSTANT X-NFA <ENTER>
     PFA LFA CONSTANT X-LFA <ENTER>
: DELINK X-NFA ' 2SWAP LFA ! 0 X-LFA ! ; <ENTER>
: TURNKEY CFA ' ABORT 6 + ! ' ;S CFA ' ABORT 8 + ! <ENTER>
DELINK BYPASS-X ' 2DUP NFA BEGIN <ENTER>
     DUP PFA LFA @ SWAP <ENTER>
     DUP C@ 1F AND 3 + OVER + SWAP <ENTER>
     DO 0 I C! LOOP <ENTER>
     DUP 0= UNTIL DROP SAVE-IT ; <ENTER>
```

(NOTE: If you do this on a screen, be certain to end with a ;S)

Instructions For Use Of TURNKEY

TURNKEY can be loaded into your system before your application or after it. Either way, save the words in your application as described on page 24 of the manual. To use TURNKEY you must first define the word that you want to be executed when your application starts up. This definition must end with "QUIT". Put this word's PFA on the stack and type TURNKEY (e.g. Suppose your start-up word is "START"... Type ' START TURNKEY <ENTER>). After it strips out the headers, it goes into an endless loop to make cassette copies. To exit that loop, use the reset button. If you want disk copies instead of cassette copies, be sure to jot down the beginning and ending addresses. (BEG = 0 +ORIGIN .) (END = HERE .). Use the reset button and from DISK BASIC use the SAVEM command.

EXAMPLE:

```
  : TEST ." THIS IS A TEST OF TURNKEY" CR ; <ENTER>
  : START ." NEXT TYPE TEST" CR QUIT ; <ENTER>
  FREEZE   <ENTER>
  ' START TURNKEY <ENTER>  (Then record your tapes)
```

TRY IT. GOOD LUCK. WE'LL BE HAPPY TO MARKET YOUR APPLICATIONS. SEND A LETTER TO REQUEST ROYALTY INFORMATION.

What If...
It Doesn't Work?

What do you do if you cannot get some part of COLORFORTH to work? Here are some tips. First of all, simply try it again. Perhaps you accidently misspelled a word. You cannot always tell this by looking at the CRT. Why not? Because some keys you can hit do not get displayed and yet COLORFORTH hunts through the dictionary trying to find, not what you thought you typed, but what you actually typed. So, try it again.

Suppose you are loading from a disk screen and everything hangs up. A likely possibility is that you left off the ;S at the end of the screen. You should end every disk screen either with "-->" or with ";S". "-->" means "next screen" - continue loading with the next sequentially numbered screen.

"Yes," you say, "but I tried those things and I still can't get it to work." Did you get an error message? If so, then look it up in the list of error messages. This should give you some clue. If the error message occurs while you are loading from disk screens, you should type:

WHERE <ENTER>

and COLORFORTH will tell you which screen is at fault and will show you the offending text.

If that doesn't resolve things you have probably misunderstood how to use a particular word. Look it up in the glossaries and in STARTING FORTH and see if this clears things up for you.

If all of the above fails, then you need to take a break and come back to it fresh and try it all over.

If it still doesn't work, we want to hear from you. Write us and we will try to help sort things out. It is possible that a bug is present in COLORFORTH. If so, we want to know so we can fix it. If it is your error, we are still glad to help. It is better to write. If you phone and we need to return your call, we will do so COLLECT.

COMPATIBILITY WITH STARTING FORTH

If you are typing in the examples from the book by Leo Brodie, STARTING FORTH, here are some tips on how to get them to work with COLORFORTH:

VARIABLEs in STARTING FORTH do not require an initial value on the stack and are not initialized until you do so explicitly. In COLORFORTH, you must put the initial value on the stack. If STARTING FORTH says

```
VARIABLE CATFISH
```

COLORFORTH should say

```
0 VARIABLE CATFISH
```

(otherwise COLORFORTH will take SOMETHING off the stack to initialize the variable, and your stack will be messed up).

STARTING FORTH's CREATE DOES> pair should be replaced with COLORFORTH's <BUILDS DOES> pair.

In COLORFORTH, CREATE is generally used to begin a machine language definition rather than to begin an array. When you use it to create a machine language definition, the smudge bit in the header is set. This way the word cannot be found in the dictionary. After you finish the definition, you must toggle this smudge bit by typing SMUDGE so the word can be found in the dictionary. See page 18 for an example of creating a machine language word.

Instead of using STARTING FORTH's CREATE to set up arrays, you could use COLORFORTH's 0 VARIABLE -2 ALLOT.

The EDITORs used are also different. To see how to use COLORFORTH's (the standard fig line editor) see the EDITOR USER MANUAL in the COLORFORTH manual.

EXECUTE in COLORFORTH takes a code field address (CFA) instead of a parameter field address (PFA). So, do not say

```
' <word> EXECUTE
```

Instead, say

```
' <word> CFA EXECUTE
```

In COLORFORTH the word WORD does not return an address, but just leaves the count and word at HERE. In

STARTING FORTH, WORD returns the address. So, STARTING FORTH's WORD should become COLORFORTH's WORD HERE.

In COLORFORTH the block size is 256 bytes, not 1024. But, screens are 1024 bytes long.

These hints should help you use the book.

These words are listed in alphabetical order, ignoring special characters. For regular FORTH words, please see the fig Glossary later in this manual.

64K-OFF (--) This turns the upper RAM off and vectors I/O words to <EMIT> <KEY> <?TERM> <DSKCON> <CSCODE> and <CLOADM>.

64OFF (--) This word turns the ROM on and the RAM off and re-enables interrupts. It is used within 64K-OFF and at the beginning of the 64K I/O words (such as EMIT-64 and KEY-64) so they can access the ROM. See 64ON.

64K-ON (--) This turns the upper RAM on and vectors I/O words to EMIT-64 KEY-64 ?TERM-64 DSK-64 CSM-64 and CLM-64.

64ON (--) This word disables interrupts, turns the ROM off and the RAM on. It does not re-vector any I/O words. It is used within 64K-ON and at the end of the 64K I/O words (such as EMIT-64 and KEY-64). See 64OFF.

BPT (-- addr) See DSKCON. Use ! to store buffer address at BPT ("buffer pointer"). Disk version only.

CAS (--) This word enables the simulated disk (an area in RAM) and disables the real disk for saving, editing, and loading "disk" screens. Disk version only. See DISK.

CLM-64 (--) Used by CLOADM when 64K is on.

<CLOADM> (--) Used by CLOADM when 64K is off.

CLOADM (--) This COLORFORTH word reads in the next file from cassette (into the area of memory from which it was originally saved).

CR (--) See fig glossary. In COLORFORTH CR does not reset OUT. (Nor does <CR> or <CRLF>.) You must reset it explicitly (e.g. 0 OUT !) or re-vector CR to something like : <CR-2> <CR> 0 OUT ! ; ' <CR-2> IS CR. We think this approach makes OUT more flexible, for counting words per page, etc.

<CR> (--) This word emits a carriage return ($0D). CR is usually vectored to <CR>.

<CRLF> (--) This word emits a carriage return ($0D) and then emits a line feed ($0A). CR can be vectored to <CRLF> by typing ' <CRLF> IS CR.

CR/W (addr blk f --) R/W vectors to CR/W for

simulated disk screens. See R/W in glossary. The "C" stands for cassette, in which systems simulated disk screens are most often used. Disk version only.

CSAVEM (start exec end --) This is used to save a memory image to tape. It saves the area of memory from start through end. The tape file may be re-loaded from either COLORFORTH or BASIC with CLOADM. At that time the execution address will be set to exec. CSAVEM and CLOADM may be used to save and restore simulated disk screens.

<CSCODE> (--) Used by CSCODE within CSAVEM when 64K is off.

CSCODE (--) This vectors to either <CSCODE> or CSM-64. It is used within CSAVEM.

CSM-64 (--) Used by CSCODE within CSAVEM when 64K is on.

DEFER (--) This creates a new vectored word. The newly created word can be vectored or pointed to other words. The words that actually execute may be defined either before or after the word created by DEFER is defined. In COLORFORTH, the words CR LIST EMIT KEY ?TERM R/W TITLE DSKON CSCODE CLOADM were defined with DEFER and may be pointed to alternate words.

DISK (--) This word enables the real disk and disables the simulated disk for having, editing, and loading "disk" screens. Disk version only. See CAS.

DLEN (n --) This is the variable that holds the number of bytes to be displayed per line by DUMP. It should be a multiple of 8. It can be changed temporarily by typing in <number> DLEN ! <ENTER>.

DRV (-- addr) See DSKCON. Use C! to store drive number at DRV.
Disk version only.

DR/W (addr blk f --) R/W vectores to DR/W for actual disk screens. See R/W in glossary. "D" stands for disk. Disk version only.

DSK-64 (--) Used by DSKCON when 64K is on. Disk version only.

<DSKCON> (--) Used by DSKCON when 64K is off. Disk version only.

DSKCON (--) This FORTH word performs a call to DISK BASIC for disk operation. It vectors either to <DSKCON> or DSK-64. To use DSKCON first set up OPC DRV TRK SEC BPT with the correct values for the operation, drive, track, sector, and address to write from or read to. Then read

the result from STA (status). Example: suppose you want to read track 5, sector 3 into an area of memory from hex 5000 through 50FF, type

```
HEX  5 TRK C!   3 SEC C!   0 DRV C!  2 OPC C!
5000 BPT !  DSKCON   STA C@  .
```

If the status was zero, then no error occured. For more information see Radio Shack's "COLOR COMPUTER DISK SYSTEM OWNERS MANUAL & PROGRAMMING GUIDE." Disk version only.

DSK-IMG (--) This sets up the disk directory and file allocation table images at FIRST which are copied to the disk by FORMAT.

DUMP (addr count --) This dumps count bytes of memory beginning at addr in both hex and ascii format. See DLEN.

<EMIT> (c --) Used by EMIT when 64K is off.

EMIT-64 (c --) Used by EMIT when 64K is on.

<FLST> (scr# --) Used within LIST to list a screen without showing the ending spaces and "<" on each line. <FLST> stands for "fast list." See <SLST>. To use it, type

```
' <FLST> IS LIST
```

FORMAT (--) This word is used with the disk version to initialize a diskette for use with COLORFORTH. The diskette must previously have been formatted with DISK BASIC's DSKINI0. It sets up the disk directory to show the 60 granules's set aside for disk screens as a DISK BASIC file named FTHSCR.BIN, thus preventing you from overwriting this area if you copy or save other files to the same disk from BASIC. Disk version only.

FREEZE (--) this fixes cold start values so they hold the current top of dictionary, etc. Thus, you can save the current dictionary to tape or disk and later reload it with all of your currently defined words still intact. To save COLORFORTH to tape from COLORFORTH type

```
FREEZE 0 +ORIGIN  DUP  HERE CSAVEM
```

to save COLORFORTH to disk, make a note of the same values (type 0 +ORIGIN . HERE .), type FREEZE, exit to BASIC with MON and type

```
SAVEM"COLORFTH",<0 +ORIGIN>,<HERE>,<0 +ORIGIN>
```

HDR (start exec end --) This sets up the header information for CSAVEM.

HI (-- addr) This constant marks the end of the RAM area used for simulated disk screens. You may change it to whatever suits your system. See LO.

INDEX (lst-scr# last-scr# --) This works about the way INDEX worked in version 1.0, but it now does page breaks and prints a title (as vectored by TITLE) at the bottom of each page. Generally, do not use zero as the lst-scr# as a page break will immediately follow.

IS (PFA --) This is the word that points a vectored word to the word that will be executed when the vectored word is executed. IS must only be used with words created by DEFER. For example ' <CRLF> IS CR alters CR to point to the word <CRLF>.

JSR (addr --) This jumps to a machine language subroutine at addr. The subroutine should end with RTS (return to subroutine) so that control will return to FORTH. It's purpose is to execute machine language subroutines other than FORTH words.

<KEY> (-- c) Used by KEY when 64K is off.

KEY-64 (-- c) Used by KEY when 64K is on.

LO (-- addr) This constant marks the beginning of the RAM area used for simulated disk screens. You may change it to whatever suits your system. See HI.

<MON> (--) jumps to BASIC. The ROM better be on!

MON (--) See MON in the fig glossary. It saves the warm start addr so a later EXEC from BASIC will do a warm start to COLORFORTH, turns the ROM back on in case it was off, and jumps to BASIC. Be careful what you do in BASIC. Don't type a program, for example. MON is used to get to BASIC so it can do special I/O for you - such as SKIPF or DIR or SAVE or KILL.

NEXT, (--) Ends a machine code definition by laying down the code for a jump to "Next."

NOP (--) Does nothing except take up time and space. It stands for "no operation."

OPC (-- addr) See DSKCON. With C!, store the disk operation code at OPC. 0 = restore head to track 0; 1 = no operation; 2 = read sector; 3 = write sector. Disk version only.

P-OFF (--) This stops the sending of output to the printer and starts it back to the screen. See P-ON.

P-ON (--) This routes all output to the printer. See P-OFF.

PUSHD, (--) Ends a machine code definition by laying down the code for a jump to "PushD." When the code at "PushD" is executed, it pushes the contents of the D register to the stack and then executes "Next."

PUTD, (--) Ends a machine code definition by laying down the code for a jump to "PutD." When the code at "PutD" is executed, it replaces the top 16 bit number on the stack with the contents of the D register, then executes "Next."

SEC (-- addr) See DSKCON. Use C! to store sector number at SEC. Disk version only.

SHOW (lst-scr# last-scr# --) This lists a range of screens. It repeatedly calls TRIAD, so it will start with a screen evenly divisable by 3, whether lst-scr# is evenly divisable by 3 or not! TITLE is called within TRIAD. See TITLE.

<SLST> (scr# --) Used with LIST to list a screen showing all of each line, and a "<" at the end of each line. <SLST> stands for "slow list." To use it, type

```
' <SLST> IS LIST.
```

STA (-- addr) See DSKCON. Use C@ to check the status after a disk operation. Disk version only.

<?TERM> (-- c) Used by ?TERM when 64K is off.

?TERM-64 (--c) Used by ?TERM when 64K is on.

?TERM (-- c) This checks the keyboard and returns the ASCII code of the key, if one is currently pressed, otherwise it returns a zero. It does not wait for a key to be pressed.

?TERMINAL (-- f) See the fig glossary. In COLORFORTH, ?TERMINAL returns a zero if no key is pressed, or a one if the BREAK key is pressed. If any key other than BREAK is pressed it halts and waits for a key to be pressed again, at which time it returns a zero unless the BREAK key was pressed. It is used within CR and you may used it or CR within your own words to allow the halting of a scrolling display.

TITLE (--) This is used within TRIAD (and, therefore, SHOW) and INDEX to print an optional title at the bottom of the page. TITLE is initially vectored to NOP. To use it, define a new word that prints your message (e.g. : TT CR CR ." CHAPTER TWO " ;) then vector TITLE to it (e.g. ' TT IS TITLE).

TRK (-- addr) See DSKCON. Use C! to store track number at TRK. Disk version only.

fig-FORTH GLOSSARY

This glossary contains all of the word definitions in Release 1 of fig-FORTH. The definitions are presented in the order of their ascii sort.

The first line of each entry shows a symbolic description of the action of the procedure on the parameter stack. The symbols indicate the order in which input parameters have been placed on the stack. Three dashes "---" indicate the execution point; any parameters left on the stack are listed. In this notation, the top of the stack is to the right.

The symbols include:

addr	memory address
b	8 bit byte (i.e. hi 8 bits zero)
c	7 bit ascii character (hi 9 bits zero)
d	32 bit signed double interger, most significant portion with sign on top of stack.
f	boolean flag. 0=false, non-zero= true
ff	boolean false flag = 0
n	16 bit signed interger number
u	16 bit unsigned interger
tf	boolean true flag = non-zero

The capital letters on the right show definition characteristics.

C	May only be used within a colon definition. A digit indicates number of memory addresses used, if other than one.
E	Intended for execution only.
L0	Level Zero definition of FORTH-78
L1	Level One definition of FORTH-78
P	Has precedence bit set. Will execute even when compiling.
U	A user variable

This publication has been made available
by the Forth Interest Group,
P.O. Box 1105, San Carlos, Ca 94070
Some typographical and omission corrections
by Armadillo Int'l Software.

Unless otherwise noted, all references to numbers are for 16 bit signed intergers. On 8 bit data bus computers, the high byte of a number is on top of the stack, with the sign in the leftmost bit. For 32 bit signed double numbers, the most significant part (with the sign) is on top.

All arithmetic is implicitly 16 bit signed interger math, with error and under-flow indication unspecified.

! n addr --- L0
Store 16 bits of n at address.
Pronounced "store"

!CSP
Save the stack position in CSP. Used as part of the compiler security.

d1 --- d2 L0
Generate a double number d1, the next ascii character which is placed in an output string. Result d2 is the quotient after division by BASE, and is maintained for further processing. Used between <# and #> . See #S.

#> d --- addr count L0
Terminates numeric output conversion by dropping d, leaving the text address and character count suitable for TYPE.

#S d1 --- d2 L0
Generates ascii text in the text output buffer, by the use of #, until a zero double number n2 results. Used between <# and #> .

' --- addr P,L0
Used in the form:
' nnnn
Leaves the parameter field address of dictionary word nnnn. As a compiler directive, executes in a colon-definition to compile the address as a literal. If the word is not found after a search of CONTEXT and CURRENT, an appropriate error message is given. Pronounced "tick".

(P,L0
Used in the form:
(cccc)
Ignore a comment that will be delimited by

a right parenthesis on the same line. May occur during execution or in a colon-definition. A blank after the leading parenthesis is required.

(.")

C+

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

(;CODE)

C

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

(+LOOP) n ---

C2

The run-time procedure compiled by +LOOP, which increments the loop index by n and tests for loop completion. See +LOOP.

(ABORT)

Executes after an error when WARNING is -1. This word normally executes ABORT, but may be altered (with care) to a user's alternative procedure.

(DO)

C

The run-time procedure compiled by DO which moves the loop control parameters to the return stack. See DO.

(FIND) addr1 addr2 --- pfa b tf (ok)
addr1 addr2 --- ff (bad)
Searches the dictionary starting at the name field address addr2, matching to the text at addr1. Returns parameter field address, length byte of name field and boolean true for a good match. If no match is found, only a boolean false is left.

(LINE) n1 n2 --- addr count
Convert the line number n1 and the screen n2 to the disc buffer address containing the data. A count of 64 indicates the full line text length.

(LOOP)

C2

The run-time procedure compiled by LOOP which increments the loop index and tests for loop completion. See LOOP.

(NUMBER) d1 addr1 --- d2 addr2
Convert the ascii text beginning at addr1+1 with regard to BASE. The new value is

accumulated into double number d1, being left as d2. Addr2 is the address of the first unconvertable digit. Used by NUMBER.

`*` n1 n2 --- prod L0
Leave the signed product of two signed numbers.

`*/` n1 n2 n3 --- n4 L0
Leave the ratio n4 = n1*n2/n3 where all are signed numbers. Retention of an intermediate 31 bit product permits greater accuracy than would be available with the sequence: n1 n2 * n3 /

`*/MOD` n1 n2 n3 --- n4 n5 L0
Leave the quotient n5 and remainder n4 of the operation n1*n2/n3 . A 31 bit intermediate product is used as for */.

`+` n1 n2 --- sum L0
Leave the sum of n1+n2.

`+!` n addr --- L0
Add n to the value at the address. Pronounced "plus-store".

`+-` n1 n2 --- n3
Apply the sign of n2 to n1, which is left as n3.

`+BUF` addr1 --- addr2 f
Advance the disc buffer address addr1 to the address of the next buffer addr2. Boolean f is false when addr2 is the buffer presently pointed to by variable PREV.

`+LOOP` n1 --- (run)
addr n2 --- (compile) P,C2,L0
Used in a colon-definition in the form:
DO ... n1 +LOOP
At run-time, +LOOP selectively controls branching back to the corresponding DO based on n1, the loop index and the loop limit. The signed increment n1 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 (n1 > 0), or until the new index is equal to or less than the limit (n1 < 0). Upon exiting the loop, the parameters are discarded and execution continues ahead.

At compile time, +LOOP compiles the run-time word (+LOOP) and the branch offset computed from HERE to the address left on the stack by DO. n2 is used for compile time error checking.

+ORIGIN n --- addr
Leave the memory address relative by n to the origin parameter area. n is the minimum address unit, either byte or word. This definition is used to access or modify the boot-up parameters at the origin area.

, n --- L0
Store n into the next available dictionary memory cell, advancing the dictionary pointer. (comma)

- n1 n2 --- diff L0
Leave the difference of n1-n2.

--> P,L0
Continue interpretation with the next disc screen. (pronounced next-screen).

-DUP n1 -- n1 (if zero)
n1 -- n1 n1 (non-zero) L0
Reproduce n1 only if it is non-zero. This is usually used to copy a value just before IF, to eliminate the need for an ELSE part to drop it.

-FIND --- pfa b tf (found)
--- ff (not found)
Accepts the next word (delimited by blanks) in the input stream to HERE, and searches the CONTEXT and then CURRENT vocabularies for a matching entry. If found, the dictionary entry's parameter field address, its length byte, and a boolean true is left. Otherwise, only a boolean false is left.

-TRAILING addr n1 --- addr n2
Adjusts the character count n1 of a text string beginning address to suppress the output of trailing blanks. i.e. the characters at addr+n1 to addr+n2 are blanks.

. n --- L0
Print a number from a signed 16 bit two's complement value, converted

according to the numeric BASE. A trailing blank follows. Pronounced "dot".

." P,L0
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 ". The maximum number of characters may be an installation dependent value. See (.").

.LINE line scr ---
Print on the terminal device, a line of text from the disc by its line and screen number. Trailing blanks are suppressed.

.R n1 n2 ---
Print the number n1 right aligned in a field whose width is n2. No following blank is printed.

/ n1 n2 --- quot L0
Leave the signed quotient of n1/n2.

/MOD n1 n2 --- rem quot L0
Leave the remainder and signed quotient of n1/n2. The remainder has the sign of the dividend.

0 1 2 3 --- n
These small numbers are used so often that it is attractive to define them by name in the dictionary as constants.

0< n --- f L0
Leave a true flag if the number is less than zero (negative), otherwise leave a false flag.

0= n --- f L0
Leave a true flag if the number is equal to zero, otherwise leave a false flag.

0BRANCH f --- C2
The run-time procedure to conditionally branch. If f is false (zero), the following in-line parameter is added to the interpretive pointer to branch ahead or back. Compiled by IF, UNTIL, and WHILE.

1+ n1 --- n2 L1
Increment n1 by 1

2+ n1 --- n2
Leave n1 incremented by 2.

: P,E,L0
Used in the form called a colon-definition:
: cccc ... ;
Creates a dictionary entry defining cccc as equivalent to the following sequence of Forth word definitions '...' until the next ';' 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 set (P) are executed rather than being compiled.

; P,C,L0
Terminate a colon-definition and stop further compilation. Compiles the run-time ;S.

;CODE P,C,L0
Used in the form:
: cccc .... ;CODE
assembly mnemonics
Stop compilation and terminate a new defining word cccc by compiling (;CODE). Set the CONTEXT vocabulary to ASSEMBER, assembling to machine code the following mnemonics.

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 cccc. That is, when nnnn is executed, it does so by jumping to the code after nnnn. An existing defining word must exist in cccc prior to ;CODE.

;S P,L0
Stop interpretation of a screen. ;S is also the run-time word compiled at the end of a colon-definition which returns execution to the calling procedure.

< n1 n2 --- f L0
Leave a true flag if n1 is less than n2; otherwise leave a false flag.

<#

L0

Setup for pictured numeric output formatting using the words:

```
<#  #  #S  SIGN  #>
```

The conversion is done on a double number producing text at PAD.

<BUILDS

C,L0

Used within a colon-definition:

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

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

```
cccc nnnn
```

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

=

n1 n2 --- f

L0

Leave a true flag in n1=n2; otherwise leave a false flag.

>

n1 n2 --- f

L0

Leave a true flag if n1 is greater than n2; otherwise a false flag.

>R

n ---

C,L0

Remove a number from the computation stack and place as the most accessable on the return stack. Use should be balanced with R> in the same definition.

?

addr --

L0

Print the value contained at the address in free format according to the current base.

?COMP

Issue error message if not compiling.

?CSP

Issue error message if stack position differs from value saved in CSP.

?ERROR

f n ---

Issue an error message number n, if the boolean flag is true.

?EXEC

Issue an error message if not executing.

?LOADING

Issue an error message if not loading.

?PAIRS n1 n2 ---
Issue an error message if n1 does not equal n2. The message indicates that compiled conditionals do not match.

?STACK

Issue an error message if the stack is out of bounds. This definition may be installation dependent.

?TERMINAL --- f
Perform a test of the terminal keyboard for actuation of the break key. A true flag indicates actuation. This definition is installation dependent.

@ addr --- n L0
Leave the 16 bit contents of address.

ABORT L0
Clear the stacks and enter the execution state. Return control to the operators terminal, printing a message appropriate to the installation.

ABS n --- u L0
Leave the absolute value of n as u.

AGAIN addr n --- (compiling) P,C2,L0
Used in a colon-definition in the form:

```
BEGIN  ...  AGAIN
```

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

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

ALLOT n --- L0
Add the signed number to the dictionary pointer DP. May be used to reserve dictionary space or re-origin memory. n is with regard to computer address type (byte or word).

AND n1 n2 --- n3 L0
Leave the bitwise logical and of n1 and n2 as n3.

B/BUF --- n
This constant leaves the number of bytes per disc buffer, the byte count read from disc by BLOCK.

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

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

BASE --- addr U,L0
A user variable containing the current number base used for input and output conversion.

BEGIN --- addr n (compiling) P,L0
Occurs in a colon-definition in the form:

```
BEGIN ... UNTIL
BEGIN ... AGAIN
BEGIN ... WHILE ... REPEAT
```

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

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

BL --- c
A constant that leaves the ascii value for "blank".

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

BLK --- addr U,L0
A user variable containing the block number being interpreted. If zero, input

is being taken from the terminal input buffer.

BLOCK n --- addr L0
Leave the memory address of the block buffer containing block n. If the block is not already in memory, it is transferred from disc to which ever buffer was least recently written. If the block occupying that buffer has been marked as updated, it is rewritten to disc before block n is read into the buffer. See also BUFFER, R/W UPDATE FLUSH

BLOCK-READ
BLOCK-WRITE These are the preferred names for the installation dependent code to read and write one block to the disc.

BRANCH C2,L0
The run-time 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.

BUFFER n --- addr
Obtain the next memory buffer, assigning it to block n. If the contents of the buffer is marked as updated, it is written to the disc. The block is not read from the disc. The address left is the first cell within the buffer for data storage.

C! b addr ---
Store 8 bits at address. On word addressing computers, further specification is necessary regarding byte addressing.

C, b ---
Store 8 bits of b into the next available dictionary byte, advancing the dictionary pointer. This is only available on byte addressing computers, and should be used with caution on byte addressing mini-computers.

C@ addr --- b
Leave the 8 bit contents of memory address. On word addressing computers, further specification is needed regarding byte addressing.

CFA pfa --- cfa
Convert the parameter field address of a definition to its code field address.

CMOVE from to count ---
Move the specified quantity of bytes beginning at address from to address to. The contents of address from is moved first proceding toward high memory. Further specification is necessary on word addressing computers.

COLD
The cold start procedure to adjust the dictionary pointer to the minimum standard and restart via ABORT. May be called from the terminal to remove application programs and restart.

COMPILE C2
When the word containing COMPILE executes, the execution address of the word following COMPILE is copied (compiled) into the dictionary. This allows specific compilation situations to be handled in addition to simply compiling an execution address (which the interpreter already does).

CONSTANT n --- L0
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 push the value of n to the stack.

CONTEXT --- addr U,L0
A user variable containing a pointer to the vocabulary within which dictionary searches will first begin.

COUNT addr1 --- addr2 n L0
Leave the byte address addr2 and byte count n of a message text beginning at address addr1. It is presumed that the first byte at addr1 contains the text byte count and the actual text starts with the second byte. Typically, COUNT is followed by TYPE.

CR L0
Transmit a carriage return and line feed to the selected output device.

CREATE
A defining word used in the form:
CREATE cccc
by such words as CODE and CONSTANT to

create a dictionary header for a Forth definition. The code field contains the address of the words parameter field. The new word is created in the CURRENT vocabulary.

CSP --- addr U
A user variable temporarily storing the stack pointer position, for compilation error checking.

D+ d1 d2 --- dsum
Leave the double number sum of two double numbers.

D+- d1 n --- d2
Apply the sign of n to the double number d1, leaving it as d2.

D. d --- L1
Print a signed double number from a 32 bit two's compliment value. The high-order 16 bits are most accessable on the stack. Conversion is performed according to the current BASE. A blank follows. Pronounced "D-dot".

D.R d n ---
Print a signed double number d right aligned in a field n characters wide.

DABS d --- ud
Leave the absolute value ud of a double number.

DECIMAL L0
Set the numeric conversion BASE for decimal input-output.

DEFINITIONS L1
Used in the form:
    cccc DEFINITIONS
Set the CURRENT vocabulary to the CONTEXT vocabulary. In the example, executing vocabulary name cccc made it the CONTEXT vocabulary and executing DEFINITIONS made both specify vocabulary cccc.

DIGIT c n1 --- n2 tf (ok)
c n1 --- ff (bad)
Converts the ascii character c (using base n1) to its binary equivalent n2, accompanied by a true flag. If the conversion is invalid, leaves only a false flag.

DLIST

List he names of the dictionary entries in the CONTEXT vocabulary.

DLITERAL d --- d (executing)
d --- (compiling) P

If compiling, compile a stack double number into a literal. Later execution of the definition containing the literal will push it to the stack. If executing, the number will remain on the stack.

DMINUS d1 --- d2

Convert d1 to its double number two's complement.

DO n1 n2 --- (execute)
addr n --- (compile) P,C2,L0

Occurs in a colon-definition in form:

```
DO ... LOOP
DO ... +LOOP
```

At run-time, DO begins a sequence with repetitive execution controlled by a loop limit n1 and an index with initial value n2. DO removes these from the stack. 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 n1 and n2 are determined at run-time 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, LEAVE.

When compiling within the colon-definition, DO compiles (DO), leaves the following address addr and n for later error checking.

DOES> L0

A word which defines the run-time 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>. 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, multi-dimensional arrays, and compiler generation.

DP ---- addr U,L

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 ALLOT.

DPL ---- addr U,L0

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

DR0
DR1

Installation dependent commands to select disc drives, by presetting OFFSET. The contents of OFFSET is added to the block number in BLOCK to allow for this selection. Offset is suppressed for error text so that it may always originate from drive 0.

DROP n --- L0

Drop the number from the stack.

DUMP addr n --- L0

Print the contents of n memory locations beginning at addr. Both addresses and contents are shown in the current numeric base.

DUP n --- n n L0

Duplicate the value on the stack.

ELSE addr1 n1 --- addr2 n2 (compiling) P,C2,L0

Occurs within a colon-definition in the form:

IF ... ELSE ... ENDIF

At run-time, ELSE executes after the true part following IF. ELSE forces execution to skip over the following false part and resumes execution after the ENDIF. It has no stack effect.

At compile-time, ELSE emplaces BRANCH reserving a branch offset, leaves the address addr2 and n2 for error testing. ELSE also resolves the pending forward branch from IF by calculating the offset from addr1 to HERE and storing at addr1.

EMIT c --- L0
Transmit ascii character c to the selected output device. OUT is incremented for each character output.

EMPTY-BUFFERS L0
Mark all block-buffers as empty, not necessarily affecting the contents. Updated blocks are not written to the disc. This is also an initialization procedure before first use of the disc.

ENCLOSE addr1 c ---
addr1 n1 n2 n3
The text scanning primitive used by WORD. From the text address addr1 and an ascii delimiting character c, is determined the byte offset to the first non-delimiter character n1, the offset to the first delimiter after the text n2, and the offset to the first character not included. This procedure will not process past an ascii 'null', treating it as an unconditional delimiter.

END P,C2,L0
This is an 'alias' or duplicate definition for UNTIL.

ENDIF addr n --- (compile) P,C0,L0
Occurs in a colon-definition in form:
IF ... ENDIF
IF ... ELSE ... ENDIF
At run-time, ENDIF serves only as the destination of a forward branch from IF or ELSE. It marks the conclusion of the conditional structure. THEN is another name for ENDIF. Both names are supported in fig-FORTH. See also IF and ELSE.

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

ERASE addr n ---
Clear a region of memory to zero from addr over n addresses.

ERROR line --- in blk
Execute error notification and restart of system. WARNING is first examined. If 1, the text of line n, relative to screen 4 of drive 0 is printed. This line number may be positive or negative, and beyond

just screen 4. If WARNING=0, n is just printed as a message number (non-disc installation). If WARNING is -1, the definition (ABORT) is executed, which executes the system ABORT. The user may cautiously modify this execution by altering (ABORT). fig-FORTH saves the contents of IN and BLK to assist in determining the location of the error. Final action is execution of QUIT.

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

EXPECT addr count --- L0
Transfer characters from the terminal to address, until a "return" or the count of characters have been received. One or more nulls are added at the end of the text.

FENCE --- addr U
A user variable containing an address below which FORGETting is trapped. To forget below this point, the user must alter the contents of FENCE.

FILL addr quan b ---
Fill memory at the address with the specified quantity of bytes b.

FIRST --- n
A constant that leaves the address of the first (lowest) block buffer.

FLD --- addr U
A user variable for control of number output field width. Presently unused in fig-FORTH.

FORGET E,L0
Executed in the form:
    FORGET cccc
Deletes definition named cccc from the dictionary with all entries physically following it. In fig-FORTH, an error message will occur if the CURRENT and CONTEXT vocabularies are not currently the same.

FORTH P,L1
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. FORTH is immediate, so it will execute during the creation of a colon-definition, to select this vocabulary at compile-time.

HERE --- addr L0
Leave the address of the next available dictionary location.

HEX L0
Set the numeric conversion base to sixteen (hexadecimal).

HLD --- addr L0
A user variable that holds the address of the latest character of text during numeric output conversion.

HOLD c --- L0
Used between <# and #> to insert an ascii character into a pictured numeric output string. e.g. 2E HOLD will place a decimal point.

I --- n C,L0
Used within a DO-LOOP to copy the loop index to the stack. Other use is implementation dependent. See R.

ID. addr ---
Print a definition's name from its name field address.

IF f --- (run-time)
--- addr n (compile) P,C2,L0
Occurs in a colon-definition in the form:
IF (tp) ... ENDIF
IF (tp) ... ELSE (fp) ... ENDIF
At run-time, IF selects execution based on a boolean flag. If f is true (non-zero), execution continues ahead thru the true part. If f is false (zero), execution skips until just after ELSE to execute the false part. After either part, execution resumes after ENDIF. ELSE and its false part are optional. If missing, false execution skips to just after ENDIF.

At compile-time, 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.

IMMEDIATE

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 preceeding it with [COMPILE].

IN --- addr L0

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

INDEX from to ---

Print the first line of each screen over the range from, to. This is used to view the comment lines of an area of text on disc screens.

INTERPRET

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

KEY --- c L0

Leave the ascii value of the next terminal key struck.

LATEST --- addr

Leave the name field address of the topmost word in the CURRENT vocabulary.

LEAVE C,L0

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 execution proceeds normally until LOOP or +LOOP is encountered.

LFA

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

LIMIT

--- n
A constant leaving the address just above the highest memory available for a disc buffer. Usually this is the highest system memory.

LIST

n ---
L0
Display the ascii text of screen n on the selected output device. SCR contains the screen number during and after this process.

LIT

--- n
C2,L0
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

n --- (compiling)
P,C2,L0
If compiling, then compile the stack value n as a 16 bit literal. This definition is immediate so that it will execute during a colon-definition. The intended use is:

```
: xxx ( calculate ) LITERAL ;
```

Compilation is suspended for the compile-time calculation of a value. Compilation is resumed and LITERAL compiles this value.

LOAD

n ---
L0
Begin interpretation of screen n. Loading will terminate at the end of the screen or at ;S. See ;S and -->.

LOOP

addr n --- (compiling)
P,C2,L0
Occurs in a colon-definition in the form:

```
DO ... LOOP
```

At run-time, 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.

At compile-time, LOOP compiles (LOOP) and uses addr to calculate an offset to DO. n is used for error testing.

M* n1 n2 --- d
A mixed magnitude math operation which leaves the double number signed product of two signed numbers.

M/ d n1 --- n2 n3
A mixed magnitude math operator which leaves the signed remainder n2 and signed quotient n3, from a double number dividend and divisor n1. The remainder takes its sign from the dividend.

M/MOD ud1 u2 --- u3 ud4
An unsigned mixed magnitude math operation which leaves a double quotient ud4 and remainder u3, from a double dividend ud1 and single divisor u2.

MAX n1 n2 --- max L0
Leave the greater of two numbers.

MESSAGE n ---
Print on the selected output device the text of line n relative to screen 4 of drive 0. n may be positive or negative. MESSAGE may be used to print incidental text such as report headers. If WARNING is zero, the message will simply be printed as a number (disc un-available).

MIN n1 n2 --- min L0
Leave the smaller of two numbers.

MINUS n1 --- n2 L0
Leave the two's complement of a number.

MOD n1 n2 --- mod L0
Leave the remainder of n1/n2, with the same sign as n1.

MON
Exit to the system monitor, leaving a re-entry to Forth, if possible.

MOVE addr1 addr2 n ---
Move the contents of n memory cells (16 bit contents) beginning at addr1 into n cells beginning at addr2. The contents of addr1 is moved first. This definition is appropriate on word addressing computers.

NEXT
This is the inner interpreter that uses the interpretative pointer IP to execute

compiled Forth definitions. It is not directly executed but is the return point for all code procedures. It acts by fetching the address pointed by IP, storing this value in register W. It then jumps to the address pointed to by the address pointed to by W. W points to the code field of a definition which contains the address of the code which executes for that definition. This usage of indirect threaded code is a major contributor to the power, portability, and extensibility of Forth. Locations of IP and W are computer specific.

NFA — pfa --- nfa
Convert the parameter field address of a definition to its name field.

NUMBER — addr --- d
Convert a character string left at addr with a preceeding count, to a signed double number, 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.

OFFSET — --- addr — U
A user variable which may contain a block offset to disc drives. The contents of OFFSET is added to the stack number by BLOCK. Messages by MESSAGE are independent of OFFSET. See BLOCK, DR0, DR1, MESSAGE.

OR — n1 n2 --- or — L0
Leave the bit-wise logical or of two 16 bit values.

OUT — --- addr — U
A user variable that contains a value incremented by EMIT. The user may alter and examine OUT to control display formatting.

OVER — n1 n2 --- n1 n2 n1 — L0
Copy the second stack value, placing it as the new top.

PAD — --- addr — L0
Leave the address of the text output buffer, which is a fixed offset above HERE.

PFA

nfa --- pfa
Convert the name field address of a compiled definition to its parameter field address.

POP

The code sequence to remove a stack value and return to NEXT. POP is not directly executable, but is a Forth re-entry point after machine code.

PREV

--- addr
A variable containing the address of the disc buffer most recently referenced. The UPDATE command marks this buffer to be later written to disc.

PUSH

This code sequence pushes machine registers to the computation stack and returns to NEXT. It is not directly executable, but is a Forth re-entry point after machine code.

PUT

This code sequence stores machine register contents over the topmost computation stack value and returns to NEXT. It is not directly executable, but is a Forth re-entry point after machine code.

QUERY

Input 80 characters of text (or until a "return") from the operators terminal. Text is positioned at the address contained in TIB with IN set to zero.

QUIT L1

Clear the return stack, stop compilation, and return control to the operators terminal. No message is given.

R

--- n
Copy the top of the return stack to the computation stack.

R#

--- addr U
A user variable which may contain the location of an editing cursor, or other file related function.

R/W

addr blk f ---
The fig-FORTH standard disc read-write linkage. addr specifies the source or

destination block buffer, blk is the sequential number of the referenced block; and f is a flag for f=0 write and f=1 read. R/W determines the location on mass storage, performs read-write and performs any error checking.

R> --- n L0
Remove the top value from the return stack and leave it on the computation stack. See >R and R.

R0 --- addr U
A user variable containing the initial location of the return stack. Pronounced R-zero. See RP!

REPEAT addr n --- (compiling) P,C2
Used within a colon-definition in the form:
BEGIN ... WHILE ... REPEAT
At run-time, REPEAT forces an unconditional branch back to just after the corresponding BEGIN.

At compile-time, REPEAT compiles BRANCH and the offset from HERE to addr. n is used for error checking.

ROT n1 n2 n3 --- n2 n3 n1 L0
Rotate the top three values on the stack, bringing the third to the top.

RP!
A computer dependent procedure to initalize the return stack pointer from user variable R0.

S->D n --- d
Sign extend a single number to form a double number.

S0 --- addr U
A user variable that contains the initial value for the stack pointer. Pronounced S-zero. See SP!

SCR --- addr U
A user variable containing the screen number most recently referenced by LIST.

SIGN n d --- d L0
Stores an ascii "-" sign just before a converted numeric output string in the text output buffer when n is negative. n is discarded, but double number d is

maintained. Must be used between <# and #>.

SMUDGE

Used during word definition to toggle the "smudge bit" in a definitions' name field. This prevents an uncompleted definition from being found during dictionary searches, until compiling is completed without error.

SP!

A computer dependent procedure to initialize the stack pointer from S0.

SP@ --- addr
A computer dependent procedure to return the address of the stack position to the top of the stack, as it was before SP@ was executed. (e.g. 1 2 SP@ @ . . . would type 2 2 1)

SPACE L0
Transmit an ascii blank to the output device.

SPACES n --- L0
Transmit n ascii blanks to the output device.

STATE --- addr L0,U
A user variable containing the compilation state. A non-zero value indicates compilation. The value itself may be implementation dependent.

SWAP n1 n2 --- n2 n1 L0
Exchange the top two values on the stack.

TASK

A no-operation word which can mark the boundary between applications. By forgetting TASK and re-compiling, an application can be discarded in its entirety.

THEN P,C0,L0
An alias for ENDIF.

TIB --- addr U
A user variable containing the address of the terminal input buffer.

TOGGLE addr b ---
Complement the contents of addr by the bit pattern b.

TRAVERSE addr1 n --- addr2
Move across the name field of a fig-FORTH variable length name field. addr1 is the address of either the length byte or the last letter. If n=1, the motion is toward hi memory; if n=-1, the motion is toward low memory. The addr2 resulting is address of the other end of the name.

TRIAD scr ---
Display on the selected output device the three screens which include that numbered scr, beginning with a screen evenly divisable by three. Output is suitable for source text records, and includes a reference line at the bottom taken from line 15 of screen 4.

TYPE addr count --- L0
Transmit count characters from addr to the selected output device.

U* u1 u2 --- ud
Leave the unsigned double number product of two unsigned numbers.

U/ ud u1 --- u2 u3
Leave the unsigned remainder u2 and unsigned quotient u3 from the unsigned double dividend ud and unsigned divisor u1.

UNTIL f --- (run-time)
addr n --- (compile) P,C2,L0
Occurs within a colon-definition in the form:
BEGIN ... UNTIL
At run-time, UNTIL controls the conditional branch back to the corresponding BEGIN. If f is false, execution returns to just after BEGIN; if true, execution continues ahead.

At compile-time, UNTIL compiles (0BRANCH) and an offset from HERE to addr. n is used for error tests.

UPDATE L0
Marks the most recently referenced block (pointed to by PREV) as altered. The block will subsequently be transferred automatically to disc should its buffer be required for storage of a different block.

USE

--- addr
A variable containing the address of the block buffer to use next, as the least recently written.

USER

n --- L0
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 pointer register UP 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.

VARIABLE

E,L0
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.

VOC-LINK

--- addr U
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 thru multiple vocabularies.

VOCABULARY

E,L
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.

In fig-FORTH, 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.

VLIST

List the names of the definitions in the context vocabulary. "Break" will terminate the listing.

WARNING --- addr U

A user variable containing a value controlling messages. If it = 1, disc is present, and screen 4 of drive 0 is the base location for messages. If it = 0, no disc is present and messages will be presented by number. If it = -1, execute (ABORT) for a user specified procedure. See MESSAGE, ERROR.

WHILE f --- (run-time)
adl nl --- adl nl ad2 n2 P,C2

Occurs in a colon-definition in the form:
BEGIN ... WHILE (tp) ... REPEAT
At run-time, WHILE selects conditional execution based on boolean flag f. If f is true (non-zero), WHILE continues execution of the true part thru REPEAT, which then branches back to BEGIN. If f is false (zero), execution skips to just after REPEAT, exiting the structure.

At compile-time, WHILE emplaces (0BRANCH) and leaves ad2 of the reserved offset. The stack values will be resolved by REPEAT.

WIDTH --- addr U

In fig-FORTH, a user variable containing the maximum number of letters saved in the compilation of a definitions' name. It must be 1 thru 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.

WORD c --- L0

Read the next text characters from the input stream being interpreted, until a delimiter c is found, storing the packed character string beginning at the dictionary buffer HERE. WORD leaves the character count in the first byte, the characters, and ends with two or more blanks. Leading occurances of c are ignored. If BLK is zero, text is taken from the terminal input buffer, otherwise from the disc block stored in BLK. See BLK, IN.

X

This is a pseudonym for the "null" or dictionary entry for a name of one character of ascii null. It is the execution procedure to terminate interpretation of a line of text from the terminal or within a disc buffer, as both buffers always have a null at the end.

XOR n1 n2 --- xor L1

Leave the bitwise logical exclusive-or of two values.

[P,L1

Used in colon-definitions in the form:

```
: xxx [ words ] more ;
```

Suspend compilation. The words after [are executed, not compiled. This allows calculation or compilation exceptions before resuming compilation with]. See LITERAL,].

[COMPILE] P,C

Used in a colon-definition in the form:

```
: xxx [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 xxx executes, rather than at compile-time.

] L1

Resumes compilation, to the completion of a colon-definition. See [.

EDITOR USER MANUAL

by BILL STODDART
of FIG, UNITED KINGDOM

FORTH organizes its mass storage into "screens" of 1024 characters. If, for example, a diskette of 250k byte capacity is used entirely for storing text, it will appear to the user as 250 screens numbered 0 to 249.

Each screen is organized as 16 lines with 64 characters per line. The FORTH screens are merely an arrangement of virtual memory and need not correspond exactly with the screen format of a particular terminal.

Selecting a Screen and Input of Text

To start an editing session, the user types EDITOR to invoke the appropriate vocabulary.
Next, type in EMPTY-BUFFERS.

The screen to be edited is then selected, using either:

```
n LIST (list screen n and select it for editing) OR
n CLEAR  (clear screen n and select for editing)
```

To input new text to screen n after LIST or CLEAR, the P (out) command is used.

EXAMPLE:

```
0 P THIS IS HOW
1 P TO INPUT TEXT
2 P TO LINES 0, 1, AND 2 OF THE SELECTED SCREEN.
```

Line Editing

During this description of the editor, reference is made to PAD. This is a text buffer which may hold a line of text used by or saved with a line editing command, or a text string to be found or deleted by a string editing command.

PAD can be used to transfer a line from one screen to another, as well as to perform edit operations within a single screen.

Line Editor Commands

n H — Hold line n at PAD (used by system more often than by user.)

n D — Delete line n but hold it in PAD. Line 15 becomes blank as lines n+1 to 15 move up 1 line.

n T — Type line n and save it in PAD.

n R — Replace line n with the text in PAD.

n I — Insert the text from PAD at line n, moving the old line n and following lines down. Line 15 is lost.

n E — Erase line n with blanks.

n S — Spread at line n. n and subsequent lines move down 1 line. Line n becomes blank. Line 15 is lost.

Cursor Control and String Editing

The screen of text being edited resides in a buffer area of storage. The editing cursor is a variable holding an offset into this buffer area. Commands are provided for the user to position the cursor, either directly or by searching for a string of buffer text, and to insert or delete text at the cursor position.

Commands to Position the Cursor

TOP — Position the cursor at the start of the screen.

n M — Move the cursor by a signed amount n and print the cursor line. The position of the cursor on its line is shown by an ← (arrow)

String Editing Commands

F text — Search forward from the current cursor position until string "text" is found. The cursor is left at the end of the text string and the cursor line is printed. If the string is not found, an error message is given and the cursor is repositioned at the top of screen.

B — Used after F to back up the cursor by the length of the most recent text.

N — Find the next occurrence of the string found by an F command.

X text — Find and delete the string "text".

C text — Copy in text to the cursor line at the cursor position.

TILL text — Delete on the cursor line from the cursor till the end of the text string "text".

NOTE: — Typing C with no text will copy a null into the text at the cursor position. This will abruptly stop later compiling! To delete this error, type TOP X [enter].

Screen Editing Commands

n LIST — List screen n and select it for editing.

n CLEAR — Clear screen n with blanks and select it for editing.

n1 n2 COPY — Copy screen n1 to screen n2.

L — List the current screen. The cursor line is relisted after the screen listing, to show the cursor position.

FLUSH — Used at the end of an editing session to ensure that all entries and updates of text have been transferred to disc.

Editor Glossary

TEXT — c ---
Accept following text to pad. c is text delimiter.

LINE — n --- addr
Leave address of line n of current screen. This address will be in the disc buffer area.

WHERE — n1 n2 ---
n2 is the block no., n1 is offset into block. If an error is found in the source when loading from disc, the recovery routine ERROR leaves these values on the stack to help the user locate the error. WHERE uses these to print the screen and line nos. and a picture of where the error occurred.

R# — --- addr
A user variable which contains the offset of the editing cursor from the start of the screen.

#LOCATE — --- n1 n2
From the cursor position determine the line-no n2 and the offset into the line n1.

#LEAD — --- line-address offset-to-cursor

#LAG — --- cursor-addr count-after-cursor-till-EOL

-MOVE — addr line-no ---
Move a line of text from addr to line of current screen.

H — n ---
Hold numbered line at PAD.

E — n ---
Erase line n with blanks.

S — n ---
Spread. Lines n and following move down. n becomes blank.

D — n ---
Delete line n, but hold in PAD.

M — n ---
Move cursor by a signed amount and print its line.

T n ---
Type line n and save in PAD.

L ---
List the current screen.

R n ---
Replace line n with the text in PAD.

P n ---
Put the following text on line n.

I n ---
Spread at line n and insert text from PAD.

TOP ---
Position editing cursor at top of screen.

CLEAR n ---
Clear screen n, can be used to select screen n for editing.

FLUSH ---
Write all updated buffers to disc. This has been modified to cope with an error in the Micropolis CPM disc drivers.

COPY n1 n2 ---
Copy screen n1 to screen n2.

-TEXT addr1 count addr2 -- boolean
True if strings exactly match.

MATCH cursor-addr bytes-left-till-EOL str-addr str-count
--- tf cursor-advance-till-end-of-matching-text
--- ff bytes-left-till-EOL
Match the string at str-addr with all strings on the cursor line forward from the cursor. The arguments left allow the cursor R# to be updated either to the end of the matching text or to the start of the next line.

LLINE --- f
Scan the cursor line for a match to PAD text. Return flag and update the cursor R# to the end of matching text, or to the start of the next line if no match is found.

FIND ---
Search for a match to the string at PAD, from the cursor position till the end of screen. If no match found, issue an error message and reposition the cursor at the top of screen.

DELETE	n --- Delete n characters prior to the cursor.
N	--- Find next occurrence of PAD text.
F	--- Input following text to PAD and search for match from cursor position till end of screen.
B	--- Backup cursor by text in PAD.
X	--- Delete next occurrence of following text.
TILL	--- Delete on cursor line from cursor to end of the following text.
C	--- Spread at cursor and copy the following text into the cursor line.

BUGS, Etc.

We hope that there are no bugs in this program. However, if any turn up, please let us know about them. We will attempt to send you a fix. We cannot guarantee a "fix" with a phone call, so please write, don't call. If we must return a phone call, we will call collect.

We sell the program as-is with no guarantees as to its workability or suitability. ARMADILLO INT'L SOFTWARE or its employees shall have no liability or responsibility for any liability, loss, or damage, (including consequential) resulting from the use of this software.

If the cassette is found to be defective within 30 days of purchase, return it to ARMADILLO, postpaid, and we shall replace it free of charge.

ERROR MESSAGES

Message # (hex)	(dec)		Meaning
0	0		Word not found in dictionary and not a valid number
1	1		Stack underflow
2	2		Dictionary full
4	4		Definition not unique (warning only)
6	6		Disk range error
7	7		Stack overflow or dictionary too full
11	17	?COMP	Can't use this word unless compiling
12	18	?EXEC	Can't use this word unless executing
13	19	?PAIRS	Conditionals not paired correctly
14	20	?CSP	Stack not left clean
15	21	FORGET	In protected dictionary
16	22	?LOADING	Can't use this word unless loading
17	23		Line number? (line not on this screen)
18	24	FORGET	CONTEXT and CURRENT must be the same
19	25	R/W	Disk error