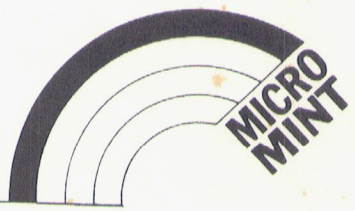
203-871-6170

Serial# 815

MICROMINT
Z8 FORTH



THE MICROMINT, INC. 4 PARK STREET, VERNON, CT 06066

Rev 2.0

203-871-6170 for technical assistance

Call Jeff at Microcomint
Bachiochi at 1 PM
Engineer

Z8/FORTH USER'S MANUAL

PRODUCT WARRANTY

Conditions of Sale

MICROMINT, INC., and the Buyer agree to the following terms and conditions of the Sale and Purchase.

1. MICROMINT, INC. extends the following warranty; a factory manufactured circuit board or assembly carries with it a 90 day warranty covering both parts and labor. Any unit which is found to have a defect in materials or workmanship shall at the option of MICROMINT, INC. be repaired or replaced.

2. For repair of units which have expired their warranty, a minimum inspection fee must be prepaid. Contact MICROMINT INC for information on current minimum charges.

3. NO WARRANTY is extended on USER ASSEMBLED systems or kits However, assembled kits will be inspected and repaired with charges based on the current minimum one hour charge. MICROMINT, INC. retains the right to refuse to repair any USER ASSEMBLED item. This right is at the sole discrertion of MICROMINT, INC.. However, in the event that repair charges would exceed a reasonable amount, the user may be consulted for a determination. Repairs on user assembled items must be PREPAID. Return authorization must be obtained prior to any return.

4. MICROMINT, INC. shall not be responsible for repair or replacement of any units which become defective through user modification, negligence, abuse and/or mishandling, or improper installation.

5. MICROMINT, INC. shall not be responsible to the Buyer for any loss or claim of special or consequential damages.

6. All units returned for repair must have prior authorization from MICROMINT, INC.. A return authorization number may be obtained by phone or letter. Please retain a record of the return authorization number as most subsequent correspondence will reference the number. Under no circumstances is any product to be returned to MICROMINT, INC. without prior authorization. MICROMINT, INC. will assume no responsibility for unauthorized returns. All returns must be shipped prepaid. Insurance is recommended as losses by a shipping carrier are not the responsibility of MICROMINT, INC Repaired units will be returned with postage paid.

7. MICROMINT, INC. reserves the right to change any feature or specification at any time as well as the minimum charges and other condition or warranty contained herein.

REVISION 12/84

EFFECTIVE DEC. 1ST 1986
MICROMINT PRODUCT WARRANTIES
ARE EXTENDED TO BE: ONE YEAR
FROM DATE OF PURCHASE...

COPYRIGHT

Z8 FORTH was written and devloped under contract by
Steve Chalmer

Steve Chalmer author

MICROMINT INC.
4 Park Street
Vernon, Connecticut 06066

716 883-6162

DISCLAIMER

THE MICROMINT INC. makes no representations or warranties with respect to the contents hereof. Further, changes are periodically made to the information contained herein. THE MICROMINT INC reserves the right to incorporate these changes in new editions of this publication without obligation to notify any person of such revision or change.

Mention in this document of specific product(s) does not constitute an endorsement of the product(s); rather, the information regarding specific products(s) is given for illustrative purposes. Description of other manufacture's interface or technical data is not intended to supercede information provided by such manufacturer.

TRADEMARK

Z8 FORTH is a trademark of CIRCUIT CELLAR INC.

Z8 is a trademark of ZILOG Corporation

Z8/FORTH USER'S MANUAL

PREFACE

Z8/FORTH is a high level language compiler on a chip. It combines the power of the Z8 microcomputer with the compactness and speed of FORTH. The basic features of the Z8 are well documented in the "Z8 Technical Manual" and the Z8 Assembly Language as it relates to FORTH Implementations have been documented in "FORTH-79" which is a publication of the FORTH Standards Team, and is distributed by the FORTH Interest Group.

In addition to these publications, We highly recommend three other books which will help you in your programming efforts. They are:

Starting FORTH, by Leo Brody
FORTH Programming, by Leo J. Scanlon
The FORTH Encyclopedia, by Mitch Derick and Linda Baker.

All of these books will be found in the bibliography section of this manual.

Throughout this document, it is presumed that you have some knowledge of the workings of the Z8 microcomputer and that you have a rudimentary knowledge of FORTH. If you are not experienced in working with FORTH, don't worry. It is very easy to pick up. The best way to learn FORTH is to use it. To that end, this manual leans very heavily towards practical examples using FORTH and approaches it in a narrative style. The first section deals with how to start talking to Z8/FORTH. The next section describes some of its main features and how to use them. This is accomplished by leading you through the programming of a full screen editor. The third section describes some of the more advanced features available. The appendix section is a brief leixicon that lists all of the words in ROM. More complete definitions of these words can be found in the Scanlon book. We have tried to point out any differences that exist.

The examples that are used presume that you have the BCC21 Z8 System/Controller board from The Micromint Inc., and that you have the SYSTEM/CONTROLLER manual.

The Micromint BCC01/02 Z8 Basic Computer/Controller may also be used but it is recommended that a Micromint 16k memory board be used for expansion beyond 4K due to address decoding constraints of the Computer/Controller board.

GETTING STARTED

Configure the jumpers on the main controller board as outlined in the Z8 system controller manual on pages 22 and 23. I recommend that you use 4K of RAM at this point. Later on you may want to experiment with autostart programs, etc.

The baud rate switch settings are the same as the ones used in BASIC/DEBUG. The chart shown here should help clarify things:

baud rate	switch # 1	2	3	4	5	6	7	8
150	.	.	.	*	*	*	*	*
19.2K	*	.	.	*	*	*	*	*
9600	.	*	.	*	*	*	*	*
4800	*	*	.	*	*	*	*	*
2400	.	.	*	*	*	*	*	*
1200	*	.	*	*	*	*	*	*
110	.	*	*	*	*	*	*	*
300	*	*	*	*	*	*	*	*

. = ON (logic low)
* = OFF (logic high)

The switches are presumed to reside as a byte at FFFD hex. In addition to these rates, any integer divisor of 19.2K may be used to get different frequencies by setting the divisor in binary and storing this byte in register F4 hex. Any value from 0-255 can be used. In general, this will only be done if you are not using the console on the serial port.

Make sure that jumper JP1 is set for the F800-FFFF range.

After setting the baud rate switches and making sure that your terminal is plugged in, turn on the power to the system. If your terminal was on before you turned on the power to the Z8 board, you will have noticed a carriage return/line feed, but you should not see any characters. Now press the return key.

If all is well, you should see the friendly greeting "OK" on the screen. This is FORTH's way of saying that everything seems to be in order, the memory is configured, there is no autostart application program to run, and FORTH is waiting for you to tell it what to do.

The way that your RAM has been configured is discussed in the section of this manual titled: word list and configuration. Right now, don't worry about it.

If for some reason you don't see "OK" then there's probably something wrong. Check the baud rate settings and the cable to your terminal, and make sure that you have the jumpers set up for the amount of RAM that you're using. Unlike Zilog's Z8 Basic/Debug 2K interpreter, Z8 FORTH uses the first 4K of memory space, so make sure that you've set the lower 2K of RAM to start at 1000H and the next 2K should start at 1800H.

Let's start by writing a program. All it will do is print the word HELLO on the screen, but it's a start.

Type in the following line exactly as you see it here. Make very sure that you put all of the spaces in. Notice that there is no space between the dot and the first quote:

```
: TEST ." HELLO" ;
```

Now press the Return key. Nothing happened! Well, it said "OK" after the semicolon, but no "HELLO". What's going on?

I said we were going to write a program to print the word "HELLO" on the screen, and we have. But we haven't run the program yet. To do that requires a command. Type in the word TEST and press the Return key.

Ha! Wrong again. That's not "HELLO", it's "HELLOOK" and where's the friendly "OK"?

Trust me. All it needs is a little tweaking. Let's try another word. This one is called NEWTEST and we type it in like this:

```
: NEWTEST CR TEST CR ;
```

Note that CR is a FORTH word not a representation for the Cariage Return key on the console.

Well, at least we got the "OK" back. Say, doesn't that word TEST look familiar? Now try typing in NEWTEST and press Return. Now we get "HELLO" on the line after our word NEWTEST and "OK" on the line after that.

Having gotten through the preliminaries, let's analyze what we've just done.

The first word, "TEST" was defined by using a colon followed by a space followed by the word TEST. (In FORTH, all words are separated by spaces. This doesn't waste any storage because the spaces are removed when the definition is compiled.) After the name of the word being defined was a curious pair of characters consisting of a period followed immediately by a quotation mark :

```
."(space)
```

This was followed by the word "HELLO" with another quotation mark at the end of the word, then a space, then a semicolon. What does it all mean? We already know what it does. It prints the word HELLOOK on the screen and loses "OK".

Nothing happens when you type characters on a line, except that the characters are put into a special place in RAM called the TIB(in long english that is Terminal Input Buffer). If you make a mistake before you type Return you can backspace and correct it, but the backspace erases the characters it goes over. If you try to backspace past the beginning of the line, you get a beep and can't go any further.

After you press the Enter or Return key, FORTH starts to look at what you're telling it to do. The first thing it came to was a colon followed by a space.

FORTH considers any character followed by a space to be a word. To find out what the word means, FORTH looks it up in a dictionary. (Isn't that what you would do?)

The word : is in the dictionary, and it means:

"Create a new word in the dictionary and name it by the next word you come to."

So, the new entry is created and the next word which FORTH comes upon is the word TEST. That's what the new entry is called. Once a colon-definition (as it's called) is started, FORTH remembers all the words between the new word's name (here, TEST) and the semicolon. These words are the definition of the new word and are executed whenever FORTH encounters the new word.

As FORTH continues to look at what you typed in it comes to the word ." and, not being bothered with our typographic prejudices, it merrily looks it up in the dictionary to find that it means:

"Get and count all of the characters you can find until you come to another quotation mark. Put the number of characters found into the dictionary followed by the characters themselves. At some point in the future, be prepared to extract these characters from the dictionary and print them to the console."

This is speedily done. Continuing, FORTH comes to the semicolon and looks it up. The semicolon means:

"Check to make sure that there are no loose ends and close the dictionary addition; then attach it formally so that it too can be referred to when required."

Having done all of that, FORTH found that there was no more that you'd typed in and so it immediately said "OK".

When you typed in TEST, FORTH looked up that word in the dictionary and found it (not surprising as it was the most recent addition). TEST says output the characters "HELLO" to the console, so FORTH does just that. Then with nothing else to do it sent "OK", and that's why TEST prints HELLOOK". You never told it to put a space after "HELLO".

NEWTEST was defined after TEST and what you said was to put out a carriage return and line feed (or CR) before TEST and another CR after TEST. This put the HELLO on its own line.

So, you've just written your first FORTH program. How do you get rid of it?

Forget it.

Wait- you mean to say I've got to keep this 'PROGRAM'? in the dictionary; that I can't get rid of it!!

Calm yourself. I said forget it and I meant it. The way to do it is to type in FORGET TEST and press return. FORTH will forget you ever defined TEST. It will also forget you ever defined NEWTEST. In fact, any time you say FORGET, FORTH will take the word immediately following the word FORGET and chop off the dictionary at that point. Everything after the word you gave as the argument for FORGET will be forgotten. FORTH will very placidly lombotomize all of its creative power and control if told to do so. You can turn your immensely powerful controller board into a vacuous dolt with absurd ease. All you have to do is say FORGET EXECUTE and press Return. (Go ahead, try it.)

We just made FORTH FORGET how to execute machine language programs. ALL machine language programs. Forth is a machine language program.

Now press the reset button, and all will be as it was. Of course if you had just spent an hour typing things into the dictionary, all that would be gone. The moral is that FORTH will always do exactly what you tell it to do. The power of FORTH is a double edged scalpel. Forth allows you to get right down to the control registers and turn off access to the console. This is real power and real responsibility. Just think ahead before you act and you'll do fine.

So much for fun and games. Let's jump head first (pun intended) into something more serious.

ADVANCED FEATURES

In this section, various hooks are described in the dictionary which will allow you to tailor your own error messages, add your own mass storage routines, and provide precompiled autostart applications programs.

Warnings and Errors

There is only one warning message trapped. If the word being defined is called by the same name as one already in the dictionary, the word being defined is printed out followed by ?U. This stands for not Unique. Be very careful of any words you use which begin with the three letters FOR. Any six letter word which starts with FOR will be impossible to FORGET, as will any word after it in the dictionary. This can be used to advantage if you want to redefine the word FORGET to include a fence buffer. This is a register which is loaded by the word FREEZE which is defined to take the address left by HERE and put it into the fence buffer. The new definition of FORGET will not allow any words to be forgotten unless their addresses are higher than the address in the fence buffer. Register pair 2EH has been reserved for this purpose if you wish to use it.

There are seven possible error messages trapped by the system.
These are:

CODE#	CHARACTER	MEANING
0056	?V	reaching the end of a buffer while in compilation mode. Unless you install a mass storage routine, you won't see this very often.
0143	<word> ?C	This word must only be used when system is in compilation mode.
024E	<word> ?N	This word is not in the dictionary and is not a number.
0353	; ?S	There is something wrong with the definition, most likely an unpaired conditional of some sort, eg. IF with no THEN or a CASE ... OF ...ELSE ... ENDCASE with a missing ELSE. 0444 :?D You are attempting to use : inside a definition.
0557	<word> ?W	This word is not in the dictionary so it cannot be compiled into a defini- tion.
0656	<word> ?V	This word is not in the dictionary, so it cannot be forgotten.

In general, fatal errors empty both stacks and perform the word QUIT. This puts you back into console mode waiting for input. A carriage return will give you OK.

The CODE# column refers to the numbers left on the stack when the error routine is invoked. There is a hook left here for writing your own error trap routine. The reserved register pair R26H is tested for a non-zero value at the beginning of the warning routine. If the register contains 0000, the warning is issued and if it is a fatal error, QUIT is invoked. If R26H does not contain 0000, the value is assumed to be the code field address of the error trap routine. This is put onto the stack and executed. To link your error trapping routine to this hook, write and test it and after it has been debugged type in:

```
FIND <word> 26 !
```

where <word> is the name of your error trap routine.

Interrupts

Interrupts may be used as you desire, but you cannot link interrupt routines into high-level code easily. However, as long as you preserve the values of certain working registers, you can vector to machine code interrupt routines very easily. When an interrupt occurs, provided the interrupt mask register has been modified in accordance with proper Z8 procedures (ie, DI change mask then EI), the interrupt will go to its normal ROM location. In these locations are the addresses of instructions which are JMP IRR04 to IRR0EH depending on the interrupt. If the addresses of the appropriate routines are loaded into RR04 through RR0EH, those routines will be jumped too. You must preserve the following things to successfully return from interrupt:

```
the system stack pointer   (RRFEH)
the register stack pointer  ( RFDH)
registers 10H-2FH
registers 50H-5FH
registers 70H-7FH
```

You may freely use registers 30H-4FH and registers 60H-6FH. These are never used by FORTH.

Autostart Hook

The autostart hook happens only on power-up, reset, or if you invoke the word COLD. The system is configured by initializing the control registers, then testing for contiguous ROM at 400H boundaries starting with the internal ROM space. The testing is performed by reading the byte at an address, writing the complement of the data to it, reading the byte again, restoring the original data, and finally comparing the two data reads. If they are identical, the location is assumed to contain ROM for the next 1023 bytes. When the first non-ROM location is found, it is presumed to be RAM, and the dictionary pointer is loaded with this address. RAM is tested at 1K boundaries using the same method until the test routine detects the first non-RAM location. At this point, the configuration is completed.

The Rstack pointer is loaded with the address of the top of RAM space. The Dstack pointer is loaded with the address 100H below the Rstack pointer value, and the TIB pointer (Terminal Input Buffer) is loaded with the same address as the Dstack pointer. The Dstack grows down towards the dictionary, and the terminal input buffer grows up towards the bottom of the Rstack. Once this configuration is completed, the top two bytes of lowest ROM block are tested. If they contain FFFFH, then the baud rate switches are read and QUIT is performed.

The system is set up in console mode waiting for input. If the two top bytes of ROM contain any other value than FFFFH, they are presumed to contain the code field address of the first word to execute in an applications program, and this address is EXECUTED. The system must contain at least 1K of RAM located just over the ROM space in order to function properly. Other RAM and ROM may be put anywhere, but if applications are to autostart, the address must be in non-writable memory contiguous with the Z8 ROM space.

Note that the actual code to be run can reside anywhere. It's only the pointer that must be at the top of contiguous ROM space. Thus if a 2716 EPROM were located from addresses 1000H to 17FFH, the code field address of the word to be executed on power-up must be stored in locations 17FEH and 17FFH. The most significant byte of the address is stored in 17FEH and the least significant byte is stored in 17FF. This can be easily done by compiling the program and then typing:

```
FIND <word> 17FE !
```

where <words> is the name of the program you want to have on power up.

Mass Storage Hook

The mass storage hook is provided through the use of the register named BLK. If you've studied the listing for the full screen editor in the last section, you will have seen that if this register (R16H) pair contains a number between 0 and 3FH, the number is presumed to be the address of a 1K screen residing in RAM. The absolute address is derived by multiplying the number in BLK by 400H. This is why screen addresses must be on even 1K boundaries. If the number in BLK is greater than 3FH, it is presumed to be the code field address of the mass storage routine and is EXECUTED. The mass storage routine is expected to leave the address of the buffer location in RAM where the new input stream is to be found by WORD.

Bachiochi

Z8 FORTH UTILITY EPROM

REV. 2.0 2732 EPROM

This set of utility words for the Z8 FORTH language includes a full screen editor, cassette I/O driver primitives, EPROM programmer primitives, and some words which make it easy and fast to write a HEX file loader for machine code downloading.

It is important that the EPROM which contains the utility words be located at B000 hex in the address space of the Micromint system. Make sure that you have properly configured the jumpers on the board which you are using to insure this.

The editor which is found in the utility EPROM differs somewhat from the full screen editor which was described in the Z8 FORTH manual. The utility editor is also full screen, but is much easier to modify in order to accommodate different control codes or add new features.

The cassette I/O primitives will allow you to easily dump and load screens of information. You can also dump and load machine code, dictionary pictures, register files, and virtually anything which resides in memory. The primitives are almost completely unconstrained with respect to header length and format, and are very easy to incorporate into any kind of cassette based file structure.

The EPROM programming primitives support 2716 and 2732 type EPROMS, and pre-suppose a 50 millisecond programming pulse. The adaptive programming algorithm is not employed. Verification, extraction of pre-programmed data, and programming are all easy to do using these primitives.

The HEX file loader primitives make it possible to download machine-code routines from another computer using the INTEL HEX file format.

For all of these utilities, I have provided examples of how to use them to some advantage. The actual uses to which you will put them are, of course, left as an exercise for the reader.

FULL SCREEN EDITOR

The FULL SCREEN EDITOR allows you to edit a 1024 byte screen of text which is located on any 1K boundary in available RAM. The screen format is 16 (decimal) lines of 64 characters each. The cursor may be moved to any character position in this space by means of re-definable control keys. Text is always entered in a type-over mode. That is, any text you type in will replace the text you are typing over. Insertion mode is not supported, but can be added if desired.

All of the examples in this manual will use the full screen editor as the means of entering and storing programs. Of course, you can also use the more primitive line by line entry method supported by the Z8 FORTH kernel to enter the examples if you wish.

To compile the vocabulary for the editor, make sure that the EPROM is properly installed in your system at address B000 hex, and that you have sufficient RAM available to use it. When compiled, the editor vocabulary takes up 1125 (decimal) bytes or a little over 1.1K. If you have 4K of RAM on the main controller board and 2K of RAM plus the UTILITY EPROM on the expansion board then you will have a bit less than 3K of dictionary space left, and room for two separate screens of editing space. If you have the expansion memory card in your system, you can easily partition the editing screens to be located anywhere in this space by changing the value stored in the variable SCR as described later in this section.

Compiling the editor vocabulary is done by typing in the line:

```
2C  BOOT
```

and pressing the carriage return or enter key. The word BOOT which is part of the kernel vocabulary causes the editor to be compiled by making the input stream come from a location in memory. In this case, that location is block 2C hex which translates into address B000 hex by means of the mass storage hook described in the Z8 FORTH manual. Note that the word BOOT ends by using the word QUIT which means that it cannot be used inside another definition because QUIT causes the system to go back to input mode and wait for keyboard instructions. The definition of BOOT is given in Appendix B. You can easily redefine another word which preserves the contents of >IN and allows you to redirect compilation from within another word. The source code for the word LSCR is a good example of this. It can be found in Appendix A.

When the editor is successfully loaded, you will get the OK prompt back. It takes about 11 seconds to compile. If it takes much longer than that, there is probably something wrong so check to make sure that your hardware is functioning properly and that the UTILITY EPROM is correctly installed at the right address.

Before using the editor, it is necessary that you provide a place in memory for the text to reside. A good location would be any RAM which resides on the expansion board, as this is likely to be non-contiguous with the dictionary. For example, let's say that you have 2K of RAM residing at address 8000 hex. To set up a screen which resides at this address type in the following line:

```
8000 SCR ! .CLEAR
```

and press the enter key. After a very slight hesitation, you will get the OK prompt back. What you have done is filled a 1K buffer beginning at 8000 hex with ASCII blank characters (20 hex) and set up a pointer to this buffer in a variable called SCR .

You can now begin to use the editor. Type in the line:

```
 EDIT
```

and press the enter key. What you should see is:

```
8000                                                          |
00<|_                                                         |
01<|                                                          |
02<|                                                          |
03<|                                                          |
04<|                                                          |
05<|                                                          |
06<|                                                          |
07<|                                                          |
08<|                                                          |
09<|                                                          |
0A<|                                                          |
0B<|                                                          |
0C<|                                                          |
0D<|                                                          |
0E<|                                                          |
0F<|                                                      ;S  |
```

If your screen does not look like that, then you must modify the control codes used by the editor. See the section below on modifications.

The cursor controls (usually shown by arrows on your terminal) should be able to move the cursor around the screen. When you reach either a line boundary or a column boundary your terminal should beep and you should not be able to cross the boundary. Note that the cursor will stop in the left most character position, but will stop one position to the right of the right most character position. This is done to allow normal backspacing. Note also that backspacing is non-destructive and that use of the rubout (or DEL) key will cause a non-printing character to be inserted into the buffer which will make a line appear shorter than it actually is when the line is listed.

If your cursor controls do not move the cursor properly, then you must modify the control codes used by the editor. See the section below on modifications.

To leave the editor, press the ESCape key. This should cause the cursor to go to the line immediately following line 0F and print the OK prompt followed by a carriage return/line feed sequence. If your escape key does not cause this to happen, then you must modify the control codes used by the editor. See the section below on modifications.

To use the editor, type in the program (I'll give a sample below) and make whatever changes you wish. Remember that spaces must be left between all words. Pay particular attention to the ends of lines. The end of one line and the beginning of the line immediately below it are right next to each other in memory. If you are cramped for space, there is nothing wrong with beginning a word on one line and ending it on the next line. The compiler will see that as one word. This is important to remember when entering strings. If you are not using the full screen editor, no string that you enter can be longer than will fit on a single 64 byte line including the ." word. In the full screen editor, you could start a string on a line and finish it 4 lines later. The maximum length is 256 bytes between the first character and the closing " .

Before I show the sample program, you will note that the current address in memory is shown as the top line on the terminal, the cursor is indicated by the underscore (_) character, and that the line numbers are shown in hex. If you had been in decimal base prior to stating the word EDIT then the address and line numbers would have been in decimal. The word ;S which appears at the end of the screen is always placed in that location by the editor. Its purpose is to insure that the screen stops loading at that point. It will always be placed there, even if you write over it so to prevent possible problems it's a good idea to avoid putting anything either one space before it or one space after it.

Here are the screen images of a sample program:

```
8000
00<|: ( 29 WORD 1 >IN +! ; IMMEDIATE  ( THIS ALLOWS COMMENTS)  |
01<|: VARIABLE <BUILDS , DOES> ;                                |
02<|0 VARIABLE OFFSET  0 VARIABLE CHECKSUM                      |
03<|: CHECKIT DUP CHECKSUM +! ;                                |
04<|: get B5A8 EXECUTE ;                                       |
05<|: *10 B5AC EXECUTE ;                                       |
06<|: *100 B5AA EXECUTE ;                                      |
07<|: 2GET  get SWAP *10 SWAP get ROT OR                       |
08<|        IF   OR CHECKIT                                     |
09<|        ELSE ." DATA ERROR"                                 |
0A<|        THEN                                               |
0B<|;                                                          |
0C<|: PROGRESS CR ." YOU HAVE LOADED SCREEN: " SCR @ 0 D. CR ;  |
0D<|: 4GET 2GET *100 2GET OR ;                                 |
0E<|: ESCAPE BEGIN KEY 1B = UNTIL CR ." OK" QUIT ;             |
0F<|PROGRESS _                                          ;S    |
```

```
8400
00<|: GETHEX  BEGIN KEY 3A = UNTIL                                    |
01<|  2GET 4GET OFFSET @ + SWAP 2GET                                   |
02<|  CASE 0 OF 1 DO 2GET OVER I + 1 - C! LOOP DROP                     |
03<|            2GET DROP 0                                            |
04<|       ELSE                                                        |
05<|       1 OF DROP 2GET DROP 1  ELSE                                 |
06<|       SWAP DROP 0 D. ."  IS NOT A RECOGNIZED RECORD TYPE"         |
07<|       CR ." PRESS  <esc> TO EXIT LOADING"                         |
08<|       ESCAPE                                                      |
09<|  ENDCASE                                                          |
0A<|;                                                                 |
0B<|: HEXLOAD 0 CHECKSUM ! CR                                         |
0C<|  BEGIN CR GETHEX CHECKSUM @ FF AND                                |
0D<|        IF ." CHECKSUM ERROR - <esc> TO QUIT" ESCAPE THEN          |
0E<|  UNTIL                                                            |
0F<|;  PROGRESS _                                              ;S     |
```

To type in this sample program, start by loading SCR with 8000 hex as shown above. Then enter the first screen full of data.

When the first screen is properly typed in, press escape and type in the line:

```
8400 SCR ! .CLEAR EDIT
```

which will put you back into editing at the top of the next screen.

When the second screen is typed in, press escape to get back to console mode.

To view the screens you have entered, use the word LIST which will scroll the listing but not put you into editing mode. Remember to change the address in SCR to view the other screen.

To load the screens, type in the following line and press enter:

```
8000 SCR ! LSCR 8400 SCR ! LSCR
```

When the OK prompt returns you have sucessfully loaded the program. You will have been notified of your progress along the way if you typed in exactly what I have shown.

(Incidentally, the program allows you to download INTEL HEX format data files from another computer. Each line is accepted with a CR/LF sequence, and each line is tested for a valid checksum and record type. It is an interesting project to rewrite this program to use XON/XOFF type protocols and interface it with an originating program such as PC-TALK. The words get *10 and *100 are defined in Appendix B.)

Modifications

In the event that your terminal doesn't work with the full screen editor as it resides in the EPROM it will be necessary to modify the control codes which the editor uses to correspond to those which your terminal uses.

To begin this process, make sure that you have your terminal manual in front of you so that you can refer to the proper codes. Note that all of the numbers I am using in the following procedures are in hex unless specifically stated otherwise.

If you look at the listing of the editor source code which is reproduced in Appendix A you will notice that the first set of 16 (decimal) colon definitions consist of one number which is left on the stack, and in the case of CRETURN one other number which is emited. These 16 definitions are divided into three groups:

1. The words SCR POS and LN# are used to define registers for variable storage;

2. The words HOME CLEFT CDOWN CUP CRIGHT CRETURN and BELL which are used to control cursor movements in the terminal;

3. The words ESC LC DC UC RC and DR which are used to identify the control codes which the terminal sends out when you press the cursor control keys.

Before making the modifications, you should determine which words you want to modify. If you don't need to use registers 70 - 75 then you need not modify any words in the first group. (If you have been following the recommendations in the Z8 FORTH manual then you should not be using the registers from 70 - 7F anyway, but these words are there should you need to modify them.)

If you tried to EDIT a screen, and the listing did not start in the upper left hand corner, then you must modify the word HOME.

If you could not exit from the editor without pressing the reset button then you must modify the word ESC .

If you did not hear the 'bell' on your terminal when you tried to advance the cursor past an editing screen boundary, then you must modify the word BELL .

If your cursor control keys did not move the cursor properly, then either the words in group two or the words in group three or both must be modified. It can be tricky to find out which option to take without experimentation unless your terminal manual tells you what character sequences are output by the cursor control keys, and what character sequences must be input to the terminal in order to control cursor movement.

The following listing shows what the editor expects to see from the terminal in order to control the cursor, what the editor outputs to the terminal to move the cursor, and which words are involved in each case (all values are in hex):

FUNCTION	VALUE FROM TERMINAL		WORD USED
left cursor	08	(^H)	LC
down cursor	0A	(^J)	DC
up cursor	0B	(^K)	UC
right cursor	0C	(^L)	RC
return (ENTER)	0D	(^M)	DR
escape edit	1B	(<esc>)	ESC

FUNCTION	VALUE TO TERMINAL	WORD USED
left cursor	08	CLEFT
down cursor	0A	CDOWN
up cursor	0B	CUP
right cursor	0C	CRIGHT
cr/lf	0D 0A	CRETURN
bell	07	BELL

If the values in this table do not correspond to those which your terminal needs, you must modify the words which don't match. It is important that only single numbers be sent FROM the terminal as cursor control keys. Values sent TO the terminal by the editor can be any number of characters long.

In the example which I will give, I will show how to modify the cursor control keys sent FROM the terminal so that they correspond to the ones used by WORDSTAR (reg. MICROPRO), and how to modify the word BELL so that it automatically performs a cursor wrapping function. This example should give you the necessary techniques to make your own modifications.

The WORDSTAR (reg. MICROPRO) control keys are shown in this table:

FUNCTION	VALUE FROM TERMINAL		WORD USED
left cursor	13	(^S)	lc
down cursor	18	(^X)	dc
up cursor	05	(^E)	uc
right cursor	04	(^D)	rc

The simplest method for modifying the editor requires the definition of four new words. If we use lower case for the names of the words, it will be easy to tell which words are ours. The definitions are:

```
: lc  13 ;
: dc  18 ;
: uc  05 ;
: rc  04 ;
```

Once these definitions are typed in, type the following lines:

```
MODIFY LC lc
MODIFY DC dc
MODIFY UC uc
MODIFY RC rc
```

This amends the editor by replacing the old compiled words with the new words which have just been defined. This is a one way procedure. Once the original words have been modified there is no way to restore the original values unless you FORGET SCR and re- BOOT . You may redefine the new words after you have MODIFYed the old ones, but you must re-MODIFY using the new definitions. If you have deleted the new definitions by FORGETting them, any attempt to use the editor will result in a crash unless you define new words and re-MODIFY .

This is not good programming practice because you are patching code which has already been compiled, but it does allow easy modification of ROMed source code.

The use of the word MODIFY make things easy, but the new definitions do take up room in the dictionary. It is also possible to redefine the group three words only by replacing their compiled literal number values with the new ones. For instance:

```
13   FIND   LC   4   +   !
```

will replace the literal number 08 in the original definition with the literal number 13 . This method is bad practice, and can only be used with group three words which consist of a single number compiled in a colon definition. The word MODIFY allows whole new control structures to be defined, not just literal number replacement. However, this method has the advantage of not taking up dictionary space. Be careful.

Another example of the word MODIFY is to redefine the word BELL so that the cursor will wrap around instead of being stuck.

If you look at the editor source code listing, you will see that the word BELL is used in the words ?CTL and EDIT . In both cases, it is invoked when the cursor is detected to be at a screen boundary, and in ?CTL it is also invoked when the control character is not recognized. In order to wrap the cursor, our new definition which we will call CWRAP must determine at which boundary if any the cursor is, and must both move the cursor to the new location and update POS for the new column number and LN# for the new line number. If CWRAP does not find the cursor at a boundary, it must ring the bell to satisfy that constraint. For whatever function it performs, CWRAP must leave a value on the stack which is EMITted by the word EDIT as part of the overall editing program logic.

With this in mind, here is a definition of CWRAP :

```
: CWRAP
   0
    ?L IF 1 + THEN
    ?R IF 2 + THEN
    ?F IF 4 + THEN
    ?0 IF 8 + THEN
    CASE
         0 OF  7  THEN
         1 OF  3E 0 DO CRIGHT LOOP 40 POS ! CRIGHT ELSE
         2 OF  3E 0 DO CLEFT  LOOP  0 POS ! CLEFT  ELSE
         4 OF   D 0 DO CUP    LOOP  0 LN# ! CUP    ELSE
         8 OF   D 0 DO CDOWN  LOOP  F LN# ! CDOWN  ELSE
         5 OF  3F 0 DO CRIGHT LOOP 40 POS !
                D 0 DO CUP    LOOP  0 LN# ! CUP    ELSE
         6 OF  3F 0 DO CLEFT  LOOP  0 POS !
                D 0 DO CUP    LOOP  0 LN# ! CUP    ELSE
         9 OF  3F 0 DO CRIGHT LOOP 40 POS !
                D 0 DO CDOWN  LOOP  0 LN# ! CDOWN  ELSE
         A OF  3F 0 DO CLEFT  LOOP  0 POS !
                D 0 DO CDOWN  LOOP  0 LN# ! CDOWN  ELSE
         DROP 7
    ENDCASE
;
```

For each test performed, a separate bit is set. This allows us to determine what to do in the corners. I have decided that when a valid control character is received in a corner, the cursor will go to the diagonally opposite corner. If the cursor is not in a corner position, it will wrap around the same column or the same line. This definition takes up another 540 decimal bytes of dictionary space (which is one reason why I left it out in the first place).

To install CWRAP load in the definition (which should be done using the full screen editor for greatest ease) and then type in:

```
MODIFY  BELL  CWRAP
```

From this point on until you FORGET SCR or reset the system, the cursor will behave according to the modifications which you made.

To make the modifications permanent, you could save your definitions on tape using the cassette utilities described in this manual, or you could blow another EPROM which contains a completely modified EDITor program and which resides at an address other than B000. If you opt for the latter choice, you can determine which number to use for the word BOOT by taking the address you would like to use and dividing it by 400 hex. This will result in a number of 3F or less. As long as the remainder of the division is zero, the EPROM will load properly there.

CASSETTE UTILITY

The cassette utility provides the primitive I/O driver for the cassette port on the system expansion board. All file organization and data structuring is left to the discretion of the programmer.

To use the cassette utility, you should define this word:

```
: CASSETTE  B5A4  EXECUTE  ;
```

This allows you to invoke the utility easily without having to remember the execution address of the code in the EPROM.

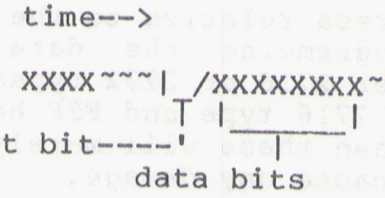
The word CASSETTE takes three arguments on the stack:

	address	count	i/o-flag
stack depth:	2	1	TOP

where address is the starting address of where the data is either coming from or going to; count is the number of bytes of data to be transmitted or received; and i/o-flag tells the utility routine whether the data is being transmitted or received.

When the i/o-flag is equal to 0 it means that the data is being received from the cassette port and placed into memory at address. After count number of bytes have been received, the utility exits with nothing on the stack and the received data stored in memory.

When the i/o-flag is non-0 it means that the data is being transmitted to the cassette port using 8 cycles of 2400Hz for a one-bit and 4 cycles of 1200Hz for a zero-bit. The transmission format for each byte is:




```
     XXXX~~~| /XXXXXXXX~~~
            | |        | |
1 start bit-'  |        `2 stop bits
          data bits
```

and count number of bytes are transmitted contiguously.

Additionally, the non-zero i/o-flag is used as a counter for the number of one-bits which are output before the first byte is transmitted. Thus, if the i/o-flag equals 100 hex, there will be 256 (decimal) one-bits output before the first byte of data. This translates into 1024 (decimal) cycles of 1200Hz or about .85 seconds of header tone which will allow the input circuitry to synchronize. This header tone can be varied at will to allow the program to catch-up to the data, etc. Any number from 1 to FFFF hex is valid, but 0 is reserved for the receive function as stated above. Bear in mind that FFFF hex would output 218 seconds of 1200Hz.

This unconstrained approach to the cassette primitives allows you to tailor your storage needs to any given application. For instance, storing and loading an editing screen full of program can be as simple as the words:

```
: STORE  SCR @ 400 4B0 CASSETTE ;
: LOAD   SCR @ 400   0 CASSETTE ;
```

Of course, you will probably want to add words which tell you to start the cassette machine in record mode, and stop the machine when the data is finished, etc. You also may want to add checksums or other data integrity checks. Remember that these must be added into the data which you are outputting. The easiest way to do that sort of thing would be to designate a separate buffer area for formatted data to which you move the information to be output, and from which you test the data being input.

Both the input and output routines for the cassette will abort if you press the ESC key on your terminal (provided this outputs the ASCII escape code which is 1B hex).

EPROM PROGRAMMING PRIMITIVES

There are three primitives in the utility ROM which are helpful in driving the EPROM programmer board. They should be invoked by defining the words:

```
: PROGRAM  B5B0  EXECUTE  ;
: READDR   B5A6  EXECUTE  ;
: MOVEIT   B5AE  EXECUTE  ;
```

The word PROGRAM takes four arguments on the stack. These are:

EPROM offset address source address count type flag

stack depth:	3	2	1	TOP

The EPROM offset address is the address relative to the start of the EPROM where you want to begin programming the data. The lowest EPROM address is 000 for either 2716 or 2732 types. The highest offset address is 7FF hex for a 2716 type and FFF hex for a 2732 type. Offset addresses higher than these will merely wrap around the address space, but will not cause any damage.

The source address is the address in system memory from which the data will be taken in order to program the EPROM.

The count is the number of bytes which will be programmed into the EPROM. These bytes are contiguous, starting from the source address and the EPROM offset address, and continuing towards increasing addresses.

The type flag is an indicator of which EPROM type is being programmed. The two types supported are:

flag	type
0	2732
1	2716

Attempting to program the wrong device could result in permanent damage to the EPROM. Make sure that the EPROM is correctly installed and that the switch on the EPROM board is set to the correct position before invoking the word PROGRAM . Remember also that 2732A type EPROMS require you to adjust the programming voltage to 21V from its normal 25V level. Contact the MICROMINT Inc. on how to make this adjustment on your board.

The word PROGRAM has no constraints on the number, source, or offset address used for programming. If the number is greater than the size of the EPROM you will be over-writing previously programmed data when the EPROM address wraps around and this will almost certainly result in garbage. You are responsible for doing your own bounds checking.

You may program only one byte if you desire, or any size block of bytes. Each byte takes ~50 milliseconds to program, so a 2716 will require about 102 seconds and a 2732 will require about 205 seconds. There is no method for aborting PROGRAM once it has been invoked short of pressing the reset button and this could very easily destroy the device. Be careful. If this distresses you, write a word which programs one byte at a time and updates the source and offset addresses by itself while checking for keyboard entries between each byte cycle. The speed overhead is quite minimal for such a word.

The word PROGRAM does not check for a blank EPROM prior to programming, nor does it perform a verification of the data after it has finished. In order to do these important tasks, you can use the word READDR.

READDR takes one argument on the stack and returns a byte:

EPROM_offset_address --- data

The EPROM offset address is the address relative to the lowest address in the EPROM (000) . The data which is returned is the contents of that address from the EPROM.

This allows us to define two useful words:

```
: ?BLANK  IF 7FF ELSE FFF THEN  0
          DO I READDR FF -
             IF ." NOT "  LEAVE  THEN
          LOOP
          ." BLANK" CR
;

: ?COMPARE IF 7FF ELSE FFF THEN 0
           DO I READDR OVER I + C@ -
              IF ." NOT "  LEAVE  THEN
           LOOP  DROP
           ." EQUAL" CR
;
```

?BLANK takes the EPROM type flag on the stack and returns a message of either BLANK or NOT BLANK .

?COMPARE takes the source address and the EPROM type flag from the stack and returns a message of either EQUAL or NOT EQUAL .

These two words allow for the verify blank and verify program functions, and both are easily modifiable for incorporation into more advanced programs.

The third useful utility is the word MOVEIT . This word takes three arguments from the stack and performs a function similar to the kernel word CMOVE .

	source	destination	count
stack depth:	2	1	TOP

Source is a memory address from which the data is taken.

Destination is a memory address to which the memory is moved.

Count is the number of contiguous bytes moved.

MOVEIT differs from CMOVE in that MOVEIT only addresses memory space, whereas CMOVE treats addresses 0000 to 00FF as register space. Furthermore, MOVEIT will perform non-destructive data moves if destination is greater than source even if source plus count is greater than destination. That is, if the data being moved upwards in memory is longer than the distance it is being moved, CMOVE will overwrite some of the data before it is moved while MOVEIT will move the data properly. If you are moving data upwards in memory, use MOVEIT . If you are moving data downwards in memory, use CMOVE but remember the constraint about register space. As you ordinarily don't attempt to move things around in ROM space, this should not pose too much of a problem.

BIBLIOGRAPHY

FORTH Encyclopedia; Mitch Derick and Linda Baker, Mountain View Press, Inc. 1982

fig-FORTH Installation Manual; Forth Interest Group 1980

FORTH-79, A Publication of the FORTH Standards Team; FORTH Standards Team, 1980

FORTH Programming; Leo J Scanlon, Howard W. Sams & Co., Inc. 1982

Starting FORTH; Leo Brodie, FORTH Inc., 1980

Z8 Microcomputer Preliminary Technical Manual; Zilog Inc., 1978

Z8 PLZ/ASM Assembly Language Programming Manual; Zilog Inc., 1980

WORD LIST AND CONFIGURATION

This section contains the words available in the Z8/FORTH ROM. In general, regular words which have similar functions are represented by only one form of the word. For example, only U* and U/ are included for multiplication and division. U/ can serve as a base for implementing the MOD function as it leaves the remainder second on the stack with the quotient on the top of the stack. Likewise, D. is available but . is not. This is because the word S->D is present which makes double numbers out of single numbers while preserving the sign. So the word . can be defined by you as : . S->D D. ; if you need it.

With the exception of the words >IN BLK and BASE all other registers which are used are not given names in the dictionary. They are defined below in terms of their location and function, should you wish to define them as constants for use in programs.

Register Pair	Name
RR04	interrupt vector address for interrupt 0
RR06	interrupt vector address for interrupt 1
RR08	interrupt vector address for interrupt 2
RR0A	interrupt vector address for interrupt 3
RR0C	interrupt vector address for interrupt 4
RR0E	interrupt vector address for interrupt 5
RR10	DPL (holds number of places to right of (radix point)
RR12	H (this is the dictionary pointer)
RR14	>IN (Input buffer index pointer)
RR16	BLK (hook to mass storage and editor)
RR18	BASE (holds the current number base)
RR1A	TIB (pointer to terminal input buffer)
RR1C	STATE (holds 0 if executing or C0H if) (compiling)
RR1E	HLD (points to current output number) (conversion)
RR20	S0 (holds the start value for the Dstack)
RR22	R0 (holds the start value for the Rstack)
RR24	WIDTH (holds the number of characters) (used in dictionary entry)
RR26	WARNING (holds 0 or the code field) (address of error trap)
RR28	CONTEXT (holds a pointer to link to) (context vocab)
RR2A	CURRENT (holds a pointer to link to) (current vocab)
RR2C	VOCLINK (points to the length byte of) (the latest dictionary entry)
RR2E	FENCE (reserved for lowest address to) (allow FORGETting)
RR50-RR5E	temporary registers used as working registers by FORTH. !!DO NOT EVER MESS WITH THESE!!
RR70-RR7E RESERVED FOR SYSTEM USE!!	/ RR78 is used for stack balancing / RR7A is the FORTH Rstack pointer \ RR7C is the FORTH instruction pointer \ RR7E is the FORTH execution vector

The control registers are loaded with the following default values on power up (the meaning of these values can be found in the Z8 Technical Manual):

RF0	RF1	RF2	RF3	RF4	RF5	RF6	RF7	RF8	RF9	RFA	RFB	RFC	RFD	RFE	RFF
00	0F	F0	FF	<a>	0F	FF	41	92	2B	00	00	00	50	<stack>	

\ value of baud rate switches

/ values depend on your memory configuration

Word list:

Words are shown below in ASCII order. Each word has a preferred pronunciation which is given in capital letters immediately following the word. Following this is the stack parameter representation which is explained below. After the stack representation is a brief explanation of what the word does, and some possible uses if it is generally used with other words, eg. CASE...OF...ELSE...ENDCASE, etc.

The stack representation consists of a set of characters separated by dashes and enclosed in parentheses:

```
( n2 n1 --- n )
```

The top of the stack is always the right most number if more than one number listed on one side of the dashes. On the left of the dashes, the characters represent the values which must be passed to the word. To the right of the dashes, the characters represent the values remaining on the stack after the word is finished executing. If there are no parameters remaining, the right side is blank. In general, the letter n will be used to represent a single number (ie. a number from 0000 to FFFF in hex) while the letter d will be used to represent a double number (from 00000000 to FFFFFFFF hex). Double numbers are stored so that the most significant 16 bits are the top of stack.

Certain words do not take parameters but do require that they be followed by another word. Compiling words are of this catagory. Certain other words may take both a value on the stack and require another word to follow them. The word CONSTANT is a good example of this. In both of these cases,the stack representation will take two lines eg.:

```
( n --- )
CONSTANT <word>
```

In this example, the <word> string is used to represent any word you choose. The < and > are not meant to be included in the word unless you wish them to be part of the spelling of the word.

For further information on the working of this vocabulary, it is strongly recommend that you consult one of the excellent books listed in the bibliography.

! STORE (n a ---)
The number second on the stack is stored into the memory location pointed to by the address on the top of stack. If the address is less than 100 hex, the address is treated as a Z8 register It is stored with the most significant byte in the lower numbered register or memory address. Register pair boundaries are not respected. Registers 80-EF hex are not used in the Z8, and writing to them effectively loses the data.

SHARP (d --- d)
This word only makes sense to use within the expressions <# and #>. Its function is to convert one digit of a number to ASCII and place the converted digit into a string which is growing downwards towards the dictionary from the value returned by PAD.

#> SHARP-GREATER THAN (d --- a n)
This word is used to terminate the pictured numeric output sequence. Its function is to drop the double number being converted, and place the address and number of characters that have been converted on the top of stack for use by another word, such as TYPE. #S SHARP-S (d --- 0 0) This word converts digits to ASCII until the entire number is converted. This leaves a double precision zero on the top of the stack. This is only used in the pictured numeric output sequence between the words.
<# and #>.

' TICK (--- a) if not compiling
(---) if compiling
' <word> in either case
This word has two actions. If used in a colon definition,it compiles the parameter address of the word which follows it as a literal. That is, when the definition executes, the parameter address of the word which followed TICK will be left on the stack rather than being executed. If TICK is used when the system is in console mode, the parameter address of the word which follows it is left on the stack. You can achieve the same result as stating a word by using the phrase:

' <word> 2 - EXECUTE
You can also change the value of a constant by using TICK. For example, if HOOHAH is a constant which was leaving the value of 13 on the stack, after executing the phrase: 68 ' HOOHAH !

the value left by the execution of HOOHAH will be 68.

+ PLUS (n n --- n)
The top two values are added and the sum is left on the stack.

+! PLUS-STORE (n a ---)
The number second on the stack is added to the number stored at the location pointed to by the top of stack.

+LOOP PLUS-LOOP (n ---)
This is one of the words used to end a DO ... +LOOP or DO ... LOOP structure. The number on the top of stack is added to the loop index. When the loop exceeds the loop limit, the loop is exited. Be careful. DO uses signed values for both the limit and the index. If either number is bigger than 8000 hex you may be in for a surprise. If you need longer loops you should either try nesting loops or use another branching structure.

, COMMA (n ---)
The number on the stack is compiled into the dictionary at the location referenced by the dictionary pointer. The dictionary pointer is incremented by 2.

- MINUS (n n --- n)
The number on the top of the stack is subtracted from the number second on the stack and the result is left as the top of stack.

." DOT-QUOTE-SPACE (---)
This word takes no arguments from the stack. It may only be used inside a definition. Its function is to take a string which follows it up to the first " character and compiles it into the definition. When the word that ." is used in is executed, the string is output to the terminal device at the current baud rate.

0= ZERO-EQUALS (n --- f) where f is 0 or 1
This word leaves a flag corresponding to the truth value of the question: is the top of stack equal to zero? If n were 0, f would be 1. Otherwise, f will be 0. If n were a truth value left from a previous logical operation, the effect of 0= would be to reverse that truth value. Used in this sense, it becomes the equivalent of the NOT function.

1+ ONE-PLUS (n --- n+1)
The number on the top of the stack is incremented.

2DROP TWO-DROP (d ---)
The double number on the top of the stack is dropped. Of course, the compiler can't tell whether the top two numbers are a double number or not, so this word will also drop the top two single numbers from the stack.

: COLON (---)
: <word>
This is the chief compiling word in FORTH. Its function is to put the system into compilation mode and save the value of the stack to check for possible compiling imbalances in structures.

; SEMI-COLON (---)

This is the word used to end colon definitions. Its function is to put the system back into console mode, toggle the smudge bit so that the interpreter will be able to find the word in the dictionary, and to check to make sure that the stack is balanced and that structures are complete.

<# LESS-THAN-SHARP (d --- d)

This word is used to open the pictured numeric output phrase. It doesn't do anything with the stack, but subsequent words will convert the double number on the stack to start a string of ASCII characters. Its function is to put the value left by PAD into the HLD register. This sets up the storage needed to buffer the output string.

<BUILDS (---)

This word is used to start the <BUILDS ... DOES> structure. The word <BUILDS can only be used inside a colon definition. When the word which contains <BUILDS executes,the word immediately following is compiled into the dictionary. When that word is executed, the words which follow the DOES> part of the defining word are executed, with the parameter address of the defined word left on the stack.

How about a couple of examples to clarify things.

Let's say we define the word VARIABLE. What this word will do is define other words in the dictionary which will allow us to refer to variables by name. Let me emphasize that:the word VARIABLE will define other words which we can use to store single numbers in. When we want to use a word which has been defined by VARIABLE,we do so by storing or fetching the value. That means that we want the words defined by VARIABLE to leave their addresses on the stack. The words ! and @ take a value and an address as their arguments. If the words defined by VARIABLE leave their addresses on the stack, all we have to supply are the values.

Now that we have a firm grip on what we want VARIABLE to do for us,we can define it. Remember that <BUILDS will create a dictionary entry when it executes, and when the word which <BUILDS built is executed it will leave its address on the stack. So we should define the word VARIABLE as

```
: VARIABLE <BUILDS , DOES> ;
```

This is a colon definition, so it starts with a colon and will end with a semi-colon. It uses the word <BUILDS and the word DOES> so we know that it must take another word immediately following it as an argument.

Right after the word <BUILDS is the word , which takes a number off the stack and compiles it into the dictionary. That means that we must have a single number on the stack before we use the word VARIABLE. There are no other words between DOES> and ; so that we know that when a word which is defined by VARIABLE executes it leaves its own address on the stack and since the only word between <BUILDS and DOES> is the word , we know that what is initially stored at that address is the number which we had on the stack when the defined word was built. Putting all of this together, we can define a variable named WINSTON which has an initial value of 1984 by stating

```
1984 VARIABLE WINSTON
```

and we can obtain the value of WINSTON by stating

```
WINSTON @
```

If we wish to change the value of WINSTON we can easily do so by stating, for example

```
1776 WINSTON !
```

In the next example, we'll make up a word which will define functional words. The words defined will take a value from the stack and add this value to another value taken from the stack when the defined word is invoked.

```
: OFFSET <BUILDS , DOES> @ + ;
```

The word OFFSET, takes a value from the stack and compiles it as a parameter for the words which it defines. When the defined words are invoked, they add their initialized value to whatever happens to be on the top of the stack at that time. Thus, if we define the following words

```
2 OFFSET 2PLUS   5 OFFSET 5PLUS
```

then stating 15 2PLUS will leave the value 17 on the top of stack. Stating 10 5PLUS will leave the value 15 on the top of stack. Words defined by OFFSET could have some application in table lookups, etc. Try some examples of your own. Remember that there is a substantial savings in dictionary space by using the <BUILDS ... DOES> construct. Words defined by the defining words using this construct take up only enough room for the parameter storage necessetated by the <BUILDS part, plus one word which points to the DOES> part. If these words do something fairly complicated, you can save a great deal of room. The proper use of <BUILDS and DOES> also makes the program very readable because you can refer to each function by a descriptive name.

<null> (---)
This word does not have a name, and cannot be invoked under normal circumstances. The word itself consists of the ASCII null character which is a byte with the value of 0 hex. The sole function of this word is to be the indicator of the end of a character input buffer such as the terminal input buffer or a buffer such as the one described in the full screen editor called SCR. What <null> does is relatively simple, but a detailed explanation is rather beyond the scope of this document. What I'll do here is give you the definition, and if you are sufficiently motivated by a need to know you can consult one of the excellent FORTH books listed in the bibliography.

Please note that although the word null is used as a place holder in this colon definition, the actual dictionary header consists of a length byte of C1 hex followed by the ASCII character 00 hex. In the dictionary, the last letter actually compiled into the name field has the most significant bit set. This means that the complete name field for the word <null> in hex looks like C180. Its definition is

```
: <null>  BLK  @
          IF   1  BLK  +!
               >IN  @
               0  >IN  !
               3FF  =
               IF    state  @  CO  =
                     IF   0056  error
                     THEN
               THEN
          ELSE  R>  DROP
          THEN
;
```

The word STATE (in line 6) leaves the value 1C hex on the stack. It is shown in lowercase because it is a virtual word that cannot be invoked unless defined. Register pair 1C is used to hold the value of the flag which indicates whether or not the system is compiling. The word ERROR (in line 7) is similarly a virtual word.

= EQUAL (n1 n2 --- f)
If the top two numbers on the stack are equal, the value of f is one. If the top two numbers are not equal, the value of f is 0. The numbers are treated as unsigned values, so the comparison is valid for both positive and negative numbers. That is, +1 and -1 are not equal.

>= GREATER-EQUAL (n1 n2 --- f)
If n1 is greater than n2, f will be equal to 1. If n1 is equal to n2, f will be equal to one. If n1 is less than n2, f will be equal to zero. The comparison is done on UNSIGNED quantities. This means that -1 (FFFFH) is greater than 0 (000H). Be careful.

>IN GREATER-IN (--- a)
This word is the name of a variable that is used to point to the offset location in an input buffer. Being a variable, it leaves an address on the stack that can be examined by the use of the words ! and @ . The address left by >IN is 14 hex which indicates that this variable is kept in one of the Z8 register pairs, namely RR14. If you refer to the definitions in the full screen editor you'll see that >IN is used as the location pointer for the characters stored in SCR .

>R TO-R (n ---)
This word takes the number on the stack and pushes it onto the R stack. Be extremely careful in your use of this word. Every use of the >R must be balanced by a use of R> or you will end up crashing the system. The Rstack is used to hold the return parameters for threading code, as well as loop indices and temporary values.

@ FETCH (a --- n)
This word leaves on the stack the value of the number stored at the address on the top of the stack.

AGAIN (---)
This is only used in conjunction with the word BEGIN to form an endlessly repeating loop. A word which has the structure

BEGIN ... AGAIN

in it will never exit unless somewhere between BEGIN and AGAIN the word <null> is encountered or the same effect as the word <null> is produced.

ALLOT (n ---)
This word advances the dictionary pointer by n bytes, where n is the number on the top of the stack. Its function is to reserve space in the dictionary. Remember that each number you wish to leave room for takes up two bytes. Double numbers take up four bytes. Strings take up one length byte plus one byte per character. In general, the word ALLOT is used in defining words to make room for tables and other structures for which the values are not known until execution time.

AND (n1 n2 --- n3)
This word performs a 16-bit logical AND operation between the two numbers on the top of the stack and leaves the result on the top of the stack.

BASE (--- a)

This word is the name of a variable where the current number base is stored. The address left on the stack is 18 hex, so the value is stored in the Z8 register pair RR18. If you state BASE @ you will always leave the value 10 on the stack. If you print out base, it will always print out as 10 unless you use the word H. which prints out the number on the top of the stack in hex regardless of the value in BASE . The system will default to the hexidecimal number base (base 16 decimal) on powerup unless you tell it to do otherwise in an autostart program. Any number base from 2 up to 7F hex is valid, but there is no checking on this. If you state 1 BASE ! you will have some difficulty obtaining meaningful numeric responses. Remember that the value in BASE represents only how you are communicating with the system. All values are treated internally as binary numbers.

BEGIN (---)

This is the opening word of various conditional and non-conditional loop structures. These can be used only inside colon definitions. The structures are listed here. For more detailed explanations of each type, see the definition of the other words used in each type.

BEGIN ... AGAIN
endless loop

BEGIN ... UNTIL
loop until a nonzero value is found on the stack when UNTIL is encountered.

BEGIN ... WHILE ... REPEAT
if a non-zero value is found on the stack when WHILE is encountered, perform the words between WHILE and REPEAT , then branch back to BEGIN . When a zero is found on the stack when WHILE is encountered, branch to the word immediately following REPEAT.

BLK B-L-K (--- a)

This is the name of a variable used to determine where the input stream is coming from. The address left on the stack is 16 hex, which means that the value is stored in the Z8 register pair RR16. See the section "Mass Storage Hooks" for more details.

C, C-COMMA (n ---)

This word compiles the least significant byte of the number on the top of the stack into the dictionary, and increments the dictionary pointer by 1. It can be used to compile byte tables, etc.

C! C-STORE (n a ---)

This word stores the least significant byte of the number second from the top of the stack (n) into the address pointed to by the number on the top of the stack (a). Only the byte at the address pointed to is affected. This is the primary method for writing a value to PORT 2, for example. The expression 1 2 C! will cause bit 1 of port 2 to go high if port two has been set up as an output port for this bit. Remember that anything under 100 hex is treated as the address of a Z8 register.

C@ C-FETCH (a --- n)

The byte stored at the address pointed to by the number at the top of the stack is fetched and left as the least significant byte of the number on the top of the stack; the most significant byte is set to 00 hex. Because numbers on the stack are always 16 bit quantities, this is the primary method of obtaining byte values from ports and other register addresses. Remember that anything under 100 hex is treated as the address of a Z8 register. You can easily change the system control registers this way, but be careful !!!

CASE (---)

This is the opening word of a multiple value testing structure. The number on the top of the stack is used to determine which of the statements contained in the structure will execute. In general, this is the most powerful programming tool that is built into the FORTH language. The CASE structure is made up of four words. These are

CASE ... n OF ... ELSE ... ENDCASE

The word ELSE is the same word used in the IF ... ELSE ... THEN structure for a good reason: you can think of a CASE structure as being a set of nested IF structures in which all of the housekeeping is done for you.

The word CASE does nothing but save the value of the stack pointer temporarily. Every time the word OF is encountered, there are assumed to be two values on the stack. The top of the stack is the number defined in the case structure (shown as n in the structure description above), and the number second on the stack is the number you are using as the index for the case table.

When OF executes, it compares the top two numbers on the stack without destroying your index value. If the numbers are equal, the index value is dropped and the words between that particular OF and the ELSE that follows it are executed. When the ELSE is encountered, the program will branch to the words that immediately follow the word ENDCASE at the end of the case table. If the numbers are not equal, the program skips to the words which immediately follow the ELSE associated with the OF that detected the nonmatch. In most cases, this word will be the number used to test for another OF ... Else pair. In the last case in the table, the words which follow the ELSE are those words which will be executed in the event that none of the OF cases found a match. When the words between the last ELSE and the word ENDCASE execute, the number used as the case table index is still on the stack. This is to aid you in the determination of why the other tests failed. If you don't need the index value, be sure to drop it. A good example of the usefulness of the case table is found in the definitions for the full screen editor. Case tables can be used more easily than IF ... THEN ... ELSE statements if you are nesting more than two deep.

C-MOVE (a1 a2 n ---)

The byte found at the location pointed to by address a1 is copied to the address pointed to by address a1; then both a1 and a2 are incremented and n is decremented. The process repeats until n decrements 1 to 0. All three numbers are unsigned single numbers, and the move is performed destructively. That is, if a1+n is greater than a2 some of the bytes moved will have been previously overwritten by earlier iterations. This can be used to advantage when filling a buffer with a set value. For a good example of this, see the definition of .CLEAR in the full screen editor descriptions.

COLD (---)

This word causes the execution of the system powerup sequence. It is the same in effect as pressing the reset button and has the same implications. The entire start up procedure will be followed, including memory sizing and baud-rate setting if there is no autostart program. It provides a very handy way of doing system development work. If a portion of RAM contiguous with the Z8 ROM space can be write-protected by means of an external switch, doing some definitions and following the procedure outlined in the " Autostart Hook " section will compile the definitions into the dictionary located in the RAM space.

At this point, assuming there is still some RAM in the system which is not write-protected to allow for stack usage, throwing the switch and invoking COLD will allow the testing of a ROM based application without the trouble of blowing an EPROM. This is particularly easy to do if you have a "soft ROM" or ROM simulator which looks like a 2732 type or 2716 type, as these can plug into the Micromint board directly.

COMPILE (---)

COMPILE <word>

This word causes the word immediately following it to be compiled into the definition of the word which is currently being compiled when COMPILE executes, not when COMPILE is being compiled.

Under most circumstances, you won't need to use this word. Its primary function is to add to the compiler by building words which are used in the creation of other words. The same kind of power is available by using the <BUILDS ... DOES> construct, and by the judicious use of the word IMMEDIATE.

CONSTANT (n ---)

<n> CONSTANT <word>

This word compiles a new word in the dictionary which leaves the value <n> when invoked. The main reason for using a constant rather than a literal number is that a constant only uses up two bytes every time it's compiled into another word while a literal uses four bytes. Also it is possible to change the value of a constant retroactively by using the sentence

<new-value> ' <constant> !

where <new-value> is the number you want the constant to leave on the stack, and <constant> represents the name of a previously defined constant. This does not make any constant into a variable. Variables in FORTH leave their addresses on the stack but constants leave their values on the stack directly.

COUNT (a --- a n)

This word presupposes that the number on the stack is the address of a string in which the first byte contains the length of the string and the subsequent bytes are the characters contained in the string. COUNT leaves on the stack the address of the first character of the string, with the number of characters on the top of the stack. This sets the string values for use by the word TYPE but can also be used for general purposes for any kind of indexed array.

CR (---)

This word causes the output of a D hex followed by an A hex to the serial output port. This is an ASCII carriage return followed by an ASCII line feed. If your terminal is the type that automatically inserts a line feed after the receipt of a carriage return and you can't disable this function you're going to have to get used to double spacing for certain things. Of course, you can always redefine CR to suit your purposes, but FORTH itself uses CR as defined here and you can't change those applications. Fortunately you can always get around them as they happen only in console mode, never in applications programs.

CREATE (---)
CREATE <word>
This is the basic method for putting together something in the dictionary. The header for the word <word> which must immediately follow the word CREATE in the input stream is compiled into the dictionary. The system is not put into compile mode, and although the link pointer is updated so that this new word can be both located and forgotten if necessary, the smudge bit is set. This means that the word cannot be executed. That's a good thing, because CREATE puts a zero in the execution address as a place holder, and trying to execute an address that contains a zero will result in a system crash. Most of the time you'll use the <BUILDS ... DOES> construct to put new defining words into the dictionary and the : ... ; construct to define new words. CREATE is there for advanced uses if you need it, however.

D+ (d1 d2 --- d3)
This word adds the two double numbers on the stack and leaves the double number sum on the stack.

D. (d ---)
This word prints all significant characters of the double number on the stack according to the current base. If the number is negative, a minus sign is output before the first character. In order to print single numbers, use the word S->D , which transforms single numbers into double numbers while preserving the sign. A very useful word you might build in this regard is the word . which prints single numbers. It can be defined as

```
: . S->D D. ;
```

DABS (d1 --- d2)
This word takes the double number on the stack and tests it as a two's compliment number. If the double number is negative, DABS negates it. Otherwise it is left positive. In short, it leaves the absolute value of the double number on the stack. One of its uses is in the formation of pictured numeric output using the <# # # ... #> structure. This works only with positive valued double numbers. See SIGN , # , and HOLD for more information.

DIGIT (n1 --- n2 1)
(n --- 0)
This word takes the number on the top of the stack and attempts to convert it to a digit according to the current base. If it succeeds, it leaves the digit as a single number on the stack and leaves a flag equal to the number one on the top of the stack. If it fails, it leaves a zero on the stack but does not leave anything else.

DLITERAL (d ---) or
(d --- d)
If the system is in compilation mode, DLITERAL takes the double number from the stack and compiles it into the current position in the dictionary as a double literal. This takes six bytes, two for the double literal primative address and four for the number itself. If the system is in console mode, DLITERAL merely leaves the double number on the stack without taking any other action.

DNEGATE (dl --- d2)
This takes the two's compliment of the double number on the stack.

DO (l i ---)
This is the beginning of the indexed loop construct. There are two variants of the loop

DO ... LOOP
DO ... +LOOP

which differ only in that LOOP adds one to the index while +LOOP (qv.) adds whatever value is on the stack when it executes. The value of the limit should be on the stack with the value of the initial index on the top of stack before DO executes. DO removes these values from the stack and puts them on the Rstack for the duration of the structure. The index is kept topmost on the Rstack.

The index of the innermost loop is accessed by the word I while the index of the next outer loop is accessed by the word J . In general, loops can be nested as deep as you wish, although you only have easy access to the two innermost indices at any given point in time. This structure can only be used inside a colon definition.

DOES> (---)
This is the companion word to <BUILDS (qv.).

DROP (nl n2 --- nl)
This word removes and discards the top of the stack.

DUP (n --- n n)
This word duplicates the number on the top of the stack.

ELSE (---)
This word is used as part of the IF ... ELSE ... THEN structure and also as part of the CASE ... OF ... ELSE ... ENDCASE structure. See IF and CASE for more information.

EMIT (c --)

This word causes the character on the top of the stack to be output to the serial port. Remember that the top of the stack is treated as a byte character by EMIT even though it is actually a 16-bit number. Only the least significant 8 bits are output, and all eight bits are output. This makes it very easy to transmit control codes or anything else in the way of binary data.

ENDCASE (---)

This is the end of the CASE structure. See CASE.

EXECUTE (a ---)

This word takes the address on the top of the stack and assumes that it contains a pointer to a machine-code routine. FORTH gets the value of that pointer and executes the code at that address.Let me stress that the address on the top of the stack is NOT the address of the machine code which will be executed, it is the address of a POINTER that contains the address of the machine code. This indirection allows you to define a set of words, then build a table of execution addresses. The addresses of the entries in the table can then be used in another word that calls out the addresses and EXECUTEs them. The value of doing this is that it allows the activity of the executing word to be redefined by changing the values stored in the table to point to other words, some of which may have been defined after the table was compiled. It is possible to define the operating system of a smart machine this way so that the system hooks are always easily accessible to upgrades without total recompilation.

EXPECT (a n ---)

This word takes the number on the top of the stack and obtains characters from the serial input port, storing them in order from the address on the stack in increasing memory. The characters are echoed to the serial output port. The character 08 hex is treated as a backspace character and deletes entries as it moves backwards over them. You are not allowed to backspace past the address given. The attempt causes the output of the character 07 hex, which is the ASCII bell character. The value stored at the address left by the variable >IN is the current offset of the input.

FIND (--- a) or (--- 0)

FIND <word>

The execution address of <word> is left on the stack if it has been previously compiled. Otherwise, the value zero is left on the stack.

FORGET (---)

FORGET <word>

If <word> exists in the dictionary, it is forgotten, as are all the words that were compiled after it. If <word> does not exist in the dictionary, an error is issued. Be very careful with this word.

H. (n ---)
This word translates the number on the top of the stack to a four digit hex number and outputs the characters to the serial port. The value of the number stored in BASE is not changed. This word has the advantage of running much faster than D. if all you need is speed and information.

HERE (--- a)
This word leaves the address of the next location in the dictionary that will be used when you compile something In effect, this is the same as stating 12 @ because because register pair 12 hex contains the dictionary pointer.

HOLD (c ---)
This word is useful only inside the <# ... #> structure, which is used for pictured numeric output. Its effect is to take the character on the top of the stack and store it in the output string which was initialized by <# and which is growing downwards towards the dictonary. For example, the expression 2E HOLD when used inside the pictured numeric output structure will cause the printing of a radix point (a period) at that location.

I (--- n)
This word leaves the current index of the innermost loop on the stack. It has a secondary use as well, which is that it copies the value of the top of the Rstack on to the top of the stack without otherwise affecting the Rstack. This can be most helpful when using the Rstack as temporary storage.

IF (f ---)
This is the opening word in the structures

```
IF ... THEN
IF ... ELSE ... THEN
```

In general, if the number on the top of the stack is non-zero, the statements immediately following the IF are executed. If the number on the top of the stack is zero, the statements immediately following the THEN are executed. If ELSE is used and the number on the top of the stack is zero, the statements between the ELSE and the THEN are executed. Note that these structures can only be used inside colon definitions. It is interesting to observe that the words IF , ELSE , and THEN do not really exist in compiled code. Their function is to compile branch instructions into the words being defined. The word THEN has the sole function of resolving the address offsets which are setup by IF and ELSE. The class of words which perform in this way (that is, words which execute while another word is being defined) is called IMMEDIATE. See the definition of IMMEDIATE for more information on this class.

IMMEDIATE (---)

This word sets a bit in the dictionary header of the word most recently defined. The bit set is called the precedence bit. Words that have their precedence bits set will execute even when the system is compiling. That is, a word that has its definition followed by the word IMMEDIATE will not be compiled into other words under normal circumstances. Instead, when the word is used, it will execute whether or not the system is compiling.

Words of this type are called IMMEDIATE WORDS. They are most often used to increase the power of the compiler by allowing new structures to be defined. A good example of immediate words is the definition of the CASE structure words.

The word CASE is defined as

```
: CASE FE @ 76 ! ; IMMEDIATE
```

The present value of the stack pointer, which is kept in register pair FE hex, is saved temporarily in register pair 76 hex, which has been reserved for this purpose. Remember that this occurs while the word in which the case structure is used is being compiled, not when this word executes. The word CASE merely sets up a condition. The word OF is defined as

```
: OF COMPILE OVER COMPILE = [COMPILE] IF COMPILE
  DROP ; IMMEDIATE
```

(Note: the word [COMPILE] is not part of the kernel dictionary. Its purpose is to enable the compilation of immediate words. Its definition is described below, and [COMPILE] is itself an immediate word.)

OF compiles a conditional test into the word being defined. That test is to see if the number that is on the stack when the word being defined executes is equal to the number that was compiled into the definition just before the word OF was used. If they are equal, the number being tested is dropped and the statements between OF and ELSE are executed. Note that the word OF contains the word IF in order to set up the test. IF is an immediate word, so special action had to be taken to compile it into the definition of the word OF ; otherwise, the IF would have executed instead of being compiled. The special word [COMPILE] was used to do this. This is defined as

```
: [COMPILE] FIND , ; IMMEDIATE
```

What this does is to find the dictionary entry for the word that follows it and compile its execution address into the currently compiling definition. [COMPILE] is itself an immediate word and that it does not end up in the definition of the word in which it appears.

The last word in the CASE structure is the word ENDCASE. This word is defined as

```
: ENDCASE BEGIN FE @ 76 @ -
     WHILE [COMPILE] THEN
     REPEAT
; IMMEDIATE
```

Remember that the word [COMPILE] is used only to compile the word THEN into the definition of ENDCASE. What happens here is that the words OF and ELSE have left addresses on the stack which need to have branch offset values stored in them to resolve the conditional branches. Since we have preserved the value of the stack pointer that the structure opened, we now use that value to test to see that all of the branches are resolved. The word THEN is used to resolve the branching set up by the words IF and ELSE , and this word is repeatedly executed when the word ENDCASE executes. I realize that all of this is rather confusing, but I have presented it in the hope of showing the kind of power which is present even in short definitions . Immediate words can greatly extend the power of the compiler. In turn, an expanded compiler can be used to make very sophisticated data structures and manipulate them easily.

J (---- n)

This word leaves the index of the second outer loop on the stack. That is, if you have nested loops, J leaves the current index of the loop in which the present loop is operating. Loops may be nested as deep as you have Rstack space for, which could be as many as 100 decimal times, but it is only easy to get at the indices of the two innermost loops through the use of I and J . Of course, you can always pass outer indices to inner loops by using I or J before beginning the next inner loop.

KEY (--- c)

This is the basic serial input port word. A character is obtained from the serial input port at the present baud rate and is ANDed with 7F hex, then put on the stack. Note that this restricts the use of KEY to only ASCII values. The actual value of the byte received is present in the serial input port buffer (Z8 register F0 hex) and the phrase F0 C@ will put all eight bits on the stack for processing if you desire. Remember that the word EMIT outputs all eight bits of the character. KEY will wait until a character is received, and the character is not echoed to the output port. If you are making a full duplex link, you must explicitly output the character using the word EMIT . Remember to DUP the character before you EMIT it or you'll lose it.

LEAVE (---)
This word causes the index of the innermost loop to be made equal to the limit of the loop. When the word LOOP or +LOOP is next encountered, the loop will exit. The loop does not exit until the word LOOP or +LOOP is encountered.

LITERAL (n --- n) if executing or in console mode
(n ---) if compiling
This word behaves differently whether the system is compiling or executing. If LITERAL is used inside a colon definition, it compiles the number on the stack as a literal number in the definition. If it is used when the system is executing or in console mode, it merely leaves the number on the stack. One of the uses of LITERAL is to compile numbers left on the stack before the definition is started.

LOOP (---)
This is one of the words used to close a DO ... LOOP structure. See the word DO .

MIN (n1 n2 --- n)
This word leaves the smaller of the two top numbers on the stack. The numbers are unsigned. That means that -1 (FFFF hex) is much bigger than 1 (0001 hex). Be careful.

NEGATE (n1 --- n2)
This replaces the number on the top of the stack with its two's complement.

NUMBER (--- d)
This takes a string that has been left at HERE by WORD and attempts to make a double number out of it according to the current base. If all of the characters are valid, the conversion is successful; otherwise, an error is issued. The characters "." and "-" are permitted. If the radix point is found, register pair 10 hex will contain the number of digits to the right of it after the conversion. Remember to define register 10 as a constant before attempting to fetch the place number value from it.

If you simply state 10 @ you'll find that the value will always be -1 (FFFF hex) because the number 10 will have been converted and it has no radix point.

OF (n --- n) or (n ---)
This word is part of the CASE structure. If the value being tested is equal to the number on the stack, the words which are between the OF and its associated ELSE are executed and the number remains for testing. See CASE for more information.

OR (n1 n2 --- n3)
The top two numbers on the stack are bitwise ORed logically, and the result is left on the stack.

OVER (n1 n2 --- n1 n2 n1)
The number second on the stack is copied and pushed onto the top of the stack. All other stack entries are pushed down.

PAD (--- a)
This leaves the address of HERE plus 44 hex on the stack. It is used most often as the starting point for pictured numeric output which grows from PAD down towards HERE .

QUIT (---)

This is the default startup word if there is no autostart program or if there is there is an error that has not been trapped by a user-defined trap. The Rstack is initialized to the value contained in register pair 22 hex; the stack is initialized to the value contained in register pair 20 hex; BLK is initialized to 0, which means that the input stream must come from the serial input port; finally, a carriage return and line feed are output. There is no "OK" until a carriage return is received from the input stream. Of course, if words are input before the carriage return they are interpreted first. Eventually, unless some form of endless loop is encountered first, the system will always return to the endless loop in QUIT.

R> R-FROM (--- n)
This word pops a number from the Rstack and pushes it onto the stack. If this has not been balanced by a previous >R there is a great likelihood that the system will crash. Be careful.

REPEAT (---)
This word is used to end a BEGIN ... WHILE ... REPEAT construct. When it executes, its function is to resolve the branch addresses set up by BEGIN and WHILE . Its effect is to cause the branching of the program to the word which immediately follows BEGIN .

ROT (n1 n2 n3 --- n2 n3 n1)
This word takes the third number down in the stack and brings it to the top of the stack. The top two numbers are pushed down.

S->D (n --- d)
This word takes the number on the top of the stack and converts it into a double number while preserving the sign. Remember that the word D. only works on double numbers, and that pictured numeric output only works on positive double numbers. To output a single number using the pictured numeric ouput including the sign, use the following phrase

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

This will output a five digit number with a leading minus sign if it is negative.

SIGN (n2 d --- d)
This word means something only when used in a pictured numeric output structure. Its function is to place a minus sign into the output string if the number third on the stack is negative. In this case, negative means any number greater than or equal to the number 8000 hex.

SMUDGE (---)
This word is used to clear a bit in the dictionary header of the word most recently defined. The cleared bit is called the smudge bit. Its function is to allow the interpreter to execute the word. The smudge bit is automatically reset by the word ; but if you're using CREATE to make a word you'll have to keep track of these things yourself. A word that is not completely defined would crash the system if executed, so when CREATE defines the dictionary header for the word and links it into the word list, it sets the smudge bit to prevent the word from being executed. It can be found using FIND and can be forgotten using the word FORGET without problems even if the smudge bit is set. This is because most often the reason that a word's definition is not finished is due to some error which prevents the ; from executing. If you don't FORGET the word, you'll get non-unique warnings when you try to redefine it, and you'll clutter up the dictionary with useless code.

SPACE (---)
This outputs an ASCII 20 hex, a space, to the serial output port.

SWAP (n1 n2 --- n2 n1)
This exchanges the top two numbers on the stack.

THEN (---)
This is the terminating word for IF ... THEN and IF ... ELSE ... THEN structures. See IF for more information.

TYPE (a n ---)
This takes n characters from the address second on the stack and emits them to the serial output port. The characters need not be ASCII, and all eight bits are output.

U* (n1 n2 --- d)
This takes the top two single numbers on the stack and performs an unsigned multiply on them, leaving a double-number unsigned product.

U/ (d n1 --- n2 n3)
This takes the double number second on the stack and performs an unsigned divide by the single number on the top of the stack. The unsigned remainder is left as the second on stack with the unsigned quotient left as the top of stack. If the divisor is 0, both the remainder and the quotient will be equal to the value FFFF hex as an indication of this. No error is issued.

UNTIL (f ---)
This is the termination word of the BEGIN ... UNTIL structure. If the number on the top of stack is zero, the program branches back to the word which immediately follows BEGIN . Otherwise, the program continues with the word which follows UNTIL .

WHILE (f ---)
This is the word which tests the condition in a BEGIN ... WHILE ... REPEAT structure. If the value on the stack is nonzero, the words that fall between WHILE and REPEAT are executed and the program branches back to the words between BEGIN and WHILE . If the value on the stack is zero when WHILE is encountered, the program branches to the words that follow REPEAT .

WORD (c --- a)
This searches the input buffer for the first occurrence of the character C that was on the stack when WORD was executed. Leading occurrences of the character are ignored, until the first occurrence of a character that doesn't match or the end of the buffer. From this point, all characters until the next occurrence of the target character are transferred with the length byte located at HERE . The address equal to HERE is left on the stack by WORD .

XOR (n1 n2 --- n3)
This word performs a bitwise xor (exclusive or) (operation or) the top two numbers on the stack and replaces them with the result.

[LEFT-BRACKET (---)
This changes the system from compilation mode to execution or console mode. It is automatically invoked by the word ; but can also be used to compute values during the compilation of a word that can then be compiled into the word as a literal. For example, the definition

```
: DO-IT 1234. [ FIND D. ] LITERAL EXECUTE CR
;
```

would type out the value 1234 when DO-IT is invoked. This can also be used to input values into active data structures by invoking words which obtain input from the terminal and using the input to affect the compilation of the word.

] RIGHT-BRACKET (---)
This word puts the system into compilation mode. It is invoked automatically by the word : but can also be used in conjunction with the word [as shown above.

APPENDIX B

The following is the source code for the full screen editor. Note that in most cases extra spaces have been added to enhance the clarity of the code.

```
: SCR      74 ;
: POS      72 ;
: LN#      70 ;
: HOME     1E ;
: CLEFT     8 ;
: CDOWN     A ;
: CUP       B ;
: CRIGHT    C ;
: CRETURN D EMIT A ;
: ESC      1B ;
: BELL      7 ;
: LC        8 ;
: DC        A ;
: UC        B ;
: RC        C ;
: DR        D ;
: ;S BLK @ IF R> DROP THEN ; IMMEDIATE
: LSCR  >IN  @  POS  !  0  >IN  !
        SCR  @  0  400  U/  SWAP  DROP  BLK  !
        662  EXECUTE
        POS  @  >IN  !
        0  BLK  !
;
: .CLEAR   SCR  @  20  OVER  C!
           DUP
           DUP  1+  3FF  CMOVE
           3FD  +   3B53  SWAP  !
           0  POS  !
           0  LN#  !
;
: <L>  S->D  <# # # #>  TYPE  ." <|"  ;
: LIST  SCR  @
        F  0  DO
                CRETURN  EMIT
                I DUP <L> 40 U* DROP OVER + 3F OVER + SWAP
                DO
                  I  C@  EMIT
                LOOP
                ." |"
              LOOP
              CRETURN  EMIT  DROP
;
: ?L POS @ 0= ;
: ?R POS @ 40 = ;
: ?0 LN# @ 0= ;
: ?F LN# @ F = ;
```

```
: ?CTL
  CASE
     LC OF ?L IF
                  BELL
                  0
               ELSE
                   CLEFT
                   -1
               THEN
        ELSE
     RC OF ?R IF
                  BELL
                  0
               ELSE
                   CRIGHT
                   1
               THEN
        ELSE
     DC OF ?F IF
                  BELL
                  0
               ELSE
                   CDOWN
                   0
                   1 LN# +!
               THEN
        ELSE
     UC OF ?0 IF
                  BELL
                  0
               ELSE
                   CUP
                   0
                   -1 LN# +!
               THEN
        ELSE
     DR OF ?F IF
                  BELL
                  0
               ELSE
                   CRETURN EMIT
                   0 POS !
                   1 LN# +!
                   LN# @ <L>
                   CLEFT EMIT
                   CRIGHT
                   0
               THEN
         ELSE
         DROP
         BELL
         0
   ENDCASE
  ;
```

```
: ?CHR 1F OVER >= ;
: CHRADR LN# @ 40 U* DROP POS @ + SCR @ + ;
: EDIT
  LN# @ F AND LN# !
  HOME EMIT
  SCR @ 0 D.
  LIST HOME EMIT CRETURN EMIT
  LN# @ DUP
  IF
    DUP 1
    DO
      CDOWN EMIT
    LOOP
  THEN
  <L>
  POS @ 3F AND DUP
  IF
    1
    DO
      CRIGHT EMIT
    LOOP
  ELSE
      DROP
  THEN
  BEGIN
       KEY DUP ESC -
  WHILE
       ?CHR
       IF
         ?CTL
       ELSE
            ?R
            IF
              DROP
              BELL
              0
            ELSE
                1
            THEN
       THEN
       SWAP DUP EMIT
       ?CHR
       IF
         DROP
       ELSE
           CHRADR C!
       THEN
       POS +!
  REPEAT
  DROP 3B53 SCR @ 3FD + !
  F LN# @
  DO
    CRETURN EMIT
  LOOP
;
```

```
: MODIFY
  FIND 2 +
  FIND
  OVER !
  600 SWAP 2 + !
;
4F EMIT
4B EMIT
CR
;S
```

```
: get  B5A8 EXECUTE ;
```

The word get obtains one ASCII character from the terminal port. If the character is a valid ASCII hexadecimal character, then the value in hex is left on the stack with a non-zero flag as top of stack indicating that a valid hex value was received. If the character was not a valid hex character, then the character itself is left on the stack and a 0 flag is left as top of stack. (Valid ASCII hex characters are: 0123456789ABCDEF . If any of these are received, the byte value is left on stack with a non-zero flag on top.)

```
: *10 B5AC EXECUTE ;
```

The word *10 supposes that there is a valid hex byte on the stack and that its value is from 0 to F . *10 swaps the nibbles of the low order byte of the top of stack thus effectively multiplying a valid hex value by 10 hex. Note that only the lower order byte has its nibbles swapped. This word is meant to be used in conjunction with get to perform a value shift function.

```
: *100 B5AA EXECUTE ;
```

The word *100 supposes that there is a valid hex number on the stack and that its value is from 00 to FF . *100 swaps the low and high order bytes of the top of stack thus effectively multiplying a valid hex value by 100 hex. Note that this word only swaps the high and low order bytes and does not perform any other arithmetic function. It could be used to some advantage in certain text editing applications, etc.

```
: BOOT  0  >IN  !  BLK  !  662  EXECUTE  QUIT  ;
```

This word is part of the kernel vocabulary. Its purpose is to allow the compilation of source code routines which reside in ROM or RAM and which begin on some even 400 hex boundary. This word can be executed from within a colon definition provided the source code which it is loading contains a directly executed program word. If the end of the source code is reached either by the use of a word such as ;S (see the full screen editor vocabulary) or by the inclusion of a 00 byte (null) in the source code listing, then the word QUIT will be executed and control will be returned to the terminal.