sinclair

ZX81-FORTH

ROM

with Multi-Tasking

BY

Skywave®

SKYWAVE SOFTWARE®

Producers of High Quality Forth Products

73 CURZON ROAD, BOSCOMBE, BOURNEMOUTH, BH1 4PW, ENGLAND
TELEPHONE: (0202) 302385 (5 lines)
International +44 202 302385

Partners: D. J. HUSBAND, D. HUSBAND

Legal Notices

ZX81-FORTH Manual © Copyright 1983 by David Husband

License

Creative Commons Deed
(Human-Readable License Summary)

About this Document

This document has been created from a scan of an original paper copy. Other than the inclusion of these 'Legal Notices', no alterations have been made to the contents of the original document.

The original document was written in 1983 by David Husband of Skywave Software, Bournemouth, UK.

This version of the document was produced in 2012 by kind permission of the author and copyright holder, David Husband. It is available for download from the DibsCo website.

For further information please visit http://www.dibsco.co.uk or email forth@dibsco.co.uk

Disclaimer

--

END OF LEGAL NOTICES

13th May 2012

Table of Contents

1.1 Introduction

This Manual is intended as a guide to the use of ZX81-FORTH, and assumes that the user will use it in conjunction with a book on FORTH. We recommend the book "The Complete FORTH" by Alan Winfield, which is available from most booksellers or in case of difficulty from us for £6.95 + £1.00 postage and packing.

Where possible, ZX81-FORTH matches the fig-FORTH commands, although ZX81-FORTH is not fig-FORTH. It was not possible to include all the fig-FORTH words because of ROM space limitations. ZX81-FORTH also contains some non-standard words so that multi-tasking can be accomplished.

ZX81-FORTH is multi-tasking. This gives the programmer the ability to write real-time routines as is described in section 14.

ZX81-FORTH is, we believe, an improvement on the fig standard in some ways by making ZX81-FORTH a compiler directive language instead of interpretive. Interpretive FORTH contains a series of addresses for each word, these being linked together by an inner-interpreter. The inner interpreter consists of an address threader using about 13 bytes and some other routines taking the total to about 70 bytes. The inner-interpreter requires 170 T states of execution time (~50uS) on a Z80. This inner-interpreter is a routine which must run as overhead for every address interpreted. A compiler directive language contains a series of calls to subroutines in each command or word. Therefore, the overhead of an inner interpreter is not necessary. The result is that compiler directive FORTH is about three times faster than interpretive FORTH in most programming applications.

Since ZX81-FORTH is not an interpretive language but instead is a compiler directive language, it should not really be classified as a TIL (Threaded Interpretive Language), but instead it would be better to call the language a Threaded Compiler Language.

ZX81-FORTH contains most of the standard fig-FORTH words. The language of FORTH is a structured programming language which allows the user to manage and manipulate all of the dynamic memory addressable by the microprocessor. FORTH is also a language in which the user can link down to machine code routines and in this respect, FORTH is only a step above assembly level programming. FORTH is however, a high-level user friendly language in that it allows the user to create his own command set. The entire program set written in FORTH is a customised set of instructions and in this way approaches other high-level languages.

In addition to the standard attributes of the FORTH language, ZX81-FORTH adds extra flexibility with its multi-tasking. Multi-tasking allows the user to schedule programs to run at any time in the future. This is a feature available only on much more expensive systems. With the proper hardware, such as plenty of I/O, a multi-tasking system can be used as a real-time controller. This means that the computer can operate at a speed sufficient enough to control the environment as events occur. A multi-tasking system could be used to enter data from a real-time environment. An example would be sampling the breathing cycle of a patient in a hospital in order to determine his or her respiratory rate. A multi-tasking system also gives the user the flexibilty of allowing a program to run in the background (it is possible to run one program in the background while editing another in the foreground).

FORTH is somewhat harder to learn than BASIC, however the flexibility gained with FORTH makes it a desirable programming language for most, if not, all programming tasks.

ZX81-FORTH is also very, very fast. Try this test :

In BASIC this program takes over 5 minutes (in SLOW mode)

```
10 FOR I = 0 TO 30000
20 NEXT I
```

The nearest equivalent in ZX81-FORTH takes about 4 seconds in SLOW, and less than 1 second in AUTO. Try it yourself.

```
AUTO 30000 0 DO LOOP
```

That makes ZX81-FORTH, for this example, about 300 times faster than ZX-81 BASIC !!

2.1 Installing the EPROM

DAVID HUSBAND sells the ZX81-FORTH ROM as a 'fit it yourself' conversion or as a ready converted computer. Those with the ready converted unit should skip this section and go to section 2.2 Initial power up.

Take the cover off the computer. There are five screws in the bottom of the case. (Three are under the pads; the pads are glued on and are easily removed.) The case is shown in figure 2.1.

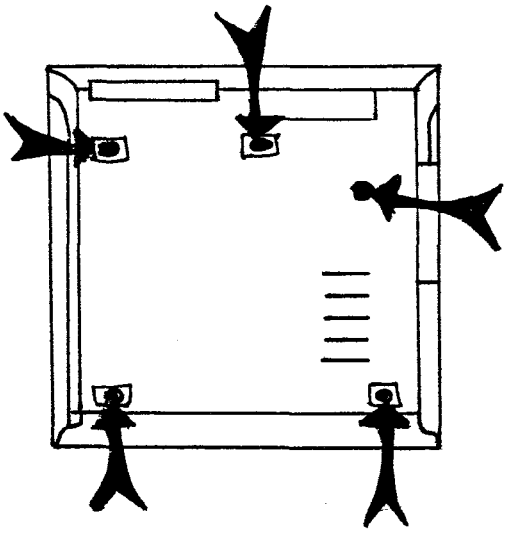
After removing the screws, the two halves of the case are easily separated. Opening the case will reveal the underside of the printed circuit board. Remove the two screws in this board as is shown in figure 2.2. Next, flip the circuit board over making sure that the ribbon cable to the keyboard does not pull out.

At this point you should be looking at the various integrated circuits on the board. Identify the BASIC ROM chip. You can find a picture of this circuit board in your Sinclair Manual. It is on either page 119 or page 162. The picture on page 162 is not of the issue 3 pcb.

Remove the BASIC ROM with great care. If it is soldered in, use a hot soldering iron and an efficient solder-sucker. Clear the unused 4 holes because we will use all 28 pins. Replace the BASIC ROM with the 28-pin I.C. Socket provided, taking care to put it in carefully. Finally plug in the EPROM supplied, with the correct orientation.

If the EPROM supplied is a 2564, it will have a couple of its pins modified to take into account the non-standard signals on the 28-pin BASIC ROM socket.

Next, turn the Sinclair circuit board over and secure it with the two screws you removed previously. Then, replace the cover and insert the five screws previously removed from the holes in the cover. At this point ZX81-FORTH will be operational (note that no BASIC will be available in this configuration.)



The Case
Fig. 2.1

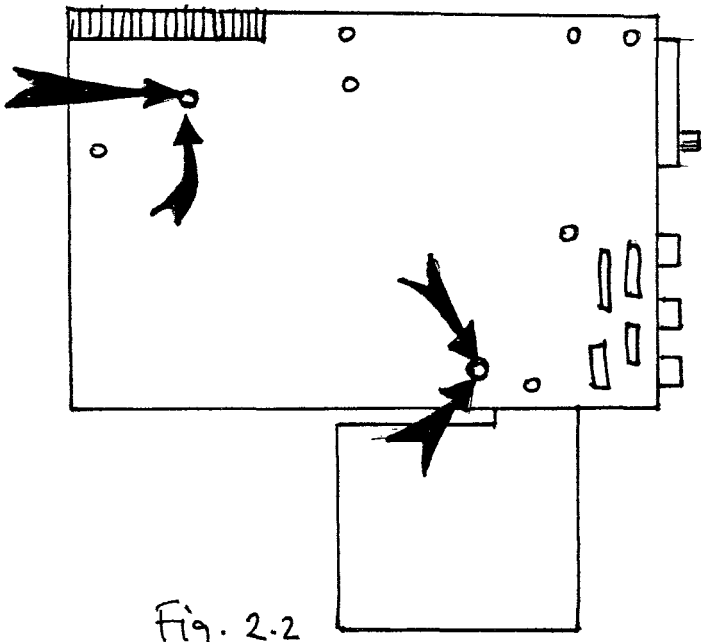


Fig. 2.2

The circuit board & screws

2.2 Initial Power-up

After completing section 2.1, or having purchased a ready converted computer, the system is ready for power-up. This section will describe for you how the various screens in ZX81-FORTH should look when you first turn the power on. After inserting the power line on the computer the screen should look like Figure 2.5. If the screen did not come up, insert the power line again. You could also try a cold restart. This is done by holding the SHIFT key and SPACE key down simultaneously for about half a second.

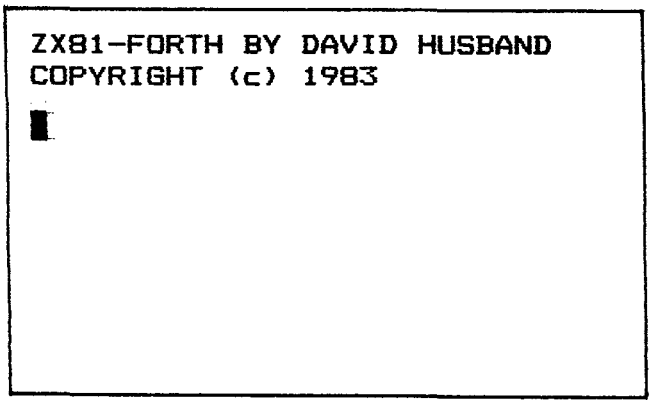
Here are some possible reasons why your screen did not come up properly:

-- Did you correctly install the 28-pin socket after removing the BASIC ROM ?

-- Did you correctly insert the ZX81-FORTH ROM in the 28-pin socket ?

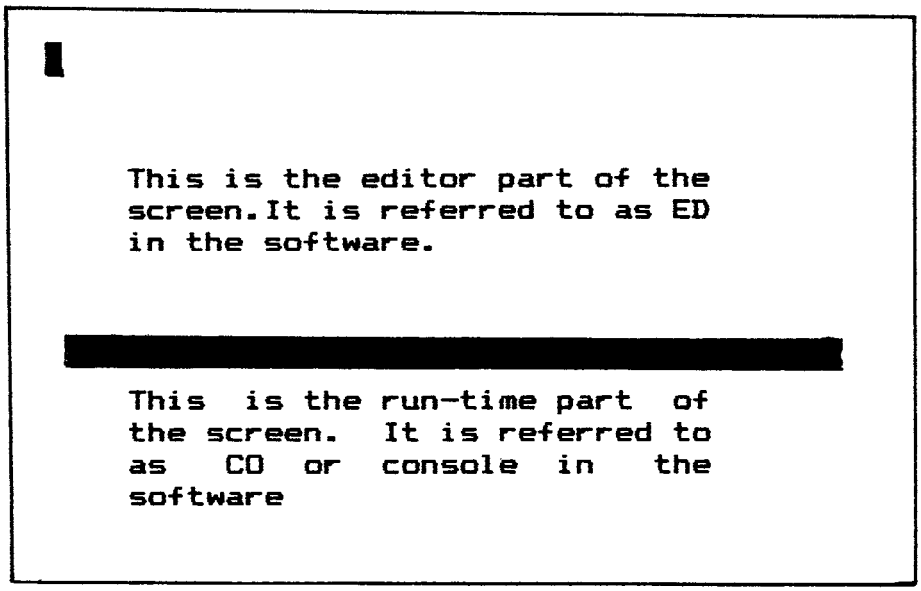
-- Did you bend any of the circuit board pins over during installation ?

-- Is your RAM Pack attached properly ? If the connections are not good, the system will not display the video information. It is best to remove the RAM Pack (if you have a 2k system) and power-up again.



Screen upon power-up or COLD restart.
Figure 2.5

There are two sets of screens in ZX81-FORTH, the first one we have seen. The second one is a split screen. To display the split screen, hold down the SHIFT and EDIT keys simultaneously. Now the screen should appear as shown in Figure 2.6.



EDITOR SCREEN
Fig 2.6

You can toggle between the two parts of the screen by using the SHIFT/EDIT keys.

To ensure that your system is working correctly, type the following commands:-

VLIST (NEW LINE) This will display all of the ZX81-FORTH words presently in memory.

.CPU (NEW LINE) This identifies the type of processor which is presently running. (A Z80)

2.3 Warm and Cold Restarts

ZX81-FORTH can be re-initialised from software anytime without having to pull out the power supply lead.

WARM RESTART: WARM (NEW LINE) will execute a warm start.

A warm restart can also be performed by holding BREAK (SHIFT/SPACE) down for an instant.

A warm restart resets the stack pointers to the absolute bottom. The system then checks for a catastrophic error such as an over-write of the system variables. If necessary, a warm restart calls a cold restart to recover. The editor is reset to the CONSOLE screen. Finally, if the task flag is off (command TOFF) any background task is set to a null program and all tasks are LOCKED.

COLD RESTART: COLD (NEW LINE) A cold restart is performed by holding the BREAK key down for about half a second.

A cold restart reconfigures the entire system and brings up the original screen, as if the power had just been turned on. All working memory (RAM > 4000H) is erased. Cold restart also checks for the amount of RAM attached to the Sinclair and stores this value in a system variable.

2.4 System Response & Errors

ZX81-FORTH prompts the user with an OK after each successful operation in the execution screen. The operation may be as simple as putting a number onto the stack, or as complicated as an entire line of program. As long as no error is found, the system reports with an OK.

If some error should occur during an operation, an ERROR statement will be displayed on the screen followed by an error code. The error codes are as follows:-

ERROR F — This error message is displayed when the user attempts to forget a FENCEd word. If the FENCEd word appears anywhere in the list of words being forgotten this error will be displayed.

* ERROR H — This error message will be flagged if the user attempts to enter a token which cannot be interpreted as a hexadecimal number or is not found in the Dictionary.

ERROR M — This error message is displayed when the available usable memory (RAM) is almost full. An error M will occur when the user program area runs to within 32 bytes of the parameter stack.

ERROR R — Error R stands for redundant. If the user trys to define a word with the same name which belongs to a program already in the dictionary, an error R will be flagged. The word or task you are defining will still be defined but will have priority over the previous word in the dictionary and the word or task already defined will no longer be accessable to the programmer.

* ERROR S — This message is displayed if the parameter stack pointer underflows, something which should never happen since popping undefined information off the stack is a no-no.

* ERROR U — Error U stands for Undefined word. This message is displayed when a word is used in a definition and either does not convert properly into a number or is not found in the dictionary. (This is only followed by a warm restart if it is found in the middle of a definition).

* NOTE: These errors will generate a warm restart after their display.

3.0 The Editor

ZX81-FORTH uses ASCII characters. This is a deviation from the Sinclair BASIC. It uses its own, non-standard, character set. Using ASCII makes the system much more flexible in terms of communicating with existing computer systems (most of which use ASCII to communicate to modems and printers).

The visual editor is a screen editor, not a line editor. This gives the user a great deal of flexibility in writing FORTH programs.

ZX81-FORTH uses two screen areas, so don't be confused. The first screen appears when you turn on the power. (See fig 2.5). The second screen is displayed with the SHIFT/EDIT set of keys. Once you are in the second screen (Fig 2.6) you can use the top part of the screen for editing. The lower part is an execution screen. (This will be called the console.) The two parts are separated by a black band called the video pad. You can switch between edit and execution screens using the SHIFT/EDIT keys.

3.1 Editor Commands

Note: If you hold any key down for more than one second, the depressed key will repeat.

SHIFT/<-	Moves the cursor one space left.
SHIFT/↓	Moves the cursor one line down.
SHIFT/↑	Moves the cursor one line up.
SHIFT/->	Moves the cursor one space right.
SHIFT/9	Inserts a line at cursor position.
SHIFT/4	Deletes the line at cursor position.
SHIFT/0	Deletes the character at cursor position.

On the editor screen, pressing NEW LINE moves the cursor down one line and does not compile that line. Continued typing on a line will provide immediate wraparound onto the next line should you type beyond the end of the screen. However, if you go back and insert characters in full lines, they will not wraparound. Instead the characters will be lost off the right side of the screen.

The insertion mode is automatic. Typing a character in the middle of a line moves all the characters following the cursor to the right.

The following commands deal with the video pad.

SHIFT/3	Takes the cursor line and stores it in Pad.
SHIFT/2	Takes the contents of Pad and puts them at the cursor position, moving the other lines below, downwards.

3.2 Compilation of Code

The following commands deal with the compilation of code from the editor screen. There are two ways to write code in ZX81-FORTH. The first is to enter commands one by one in the console screen. (The console screen is described in Section 2). The more desirable method of writing code is to write a series of words in the editor screen and then either compile the entire screen or compile the lines one by one. This allows you the freedom to go back and change things in the editor screen and recompile.

SHIFT/Q Compiles the line of code at the cursor position, and the compiled line then appears in the execution screen.

A FORTH word may be more than one line long. In this case you will have to place the cursor on the top line, compile it, and then move the cursor to compile the rest of the line. Do not worry about the compiler, it will wait until it finds a semicolon (;) before it assumes that the end of a FORTH word is reached.

If the line is successfully compiled, an OK will appear at the end of the line. If the line does not compile properly due to a programming error, then ERROR will be displayed followed by the appropriate code. The error codes are explained in Section 2.4.

CPL Compiles the entire editor screen.

Use this command after you have filled the entire editor screen and wish to compile all the statements. This is also very useful for compiling screens after they have been downloaded from the cassette tape.

You do not have to write all of your code in the editor screen and then compile it. You can compile FORTH code line by line in either Screen 1 (fig 2.5) or in the execution portion of Screen 2 (fig 2.6). Each word that you type while on the execution screen is compiled immediately. It is for this reason that using the editor to hold uncompiled (or source) code is desireable. You can make changes to the middle of a program before it is too late to change it.

The editor screen can be turned off in order to make the present screen only an execution screen.

EOFF Turns the editor screen off.

The editor screen can be turned back on again by using the SHIFT/EDIT key.

3.3 Creating Screens

Any number of screens can be created using the SCREEN command. A screen is a portion of the video display in which characters can be placed. You can have as many screens as you wish and they can be any rectangular size, ranging from one character by one character to as large as the screen itself. If a screen is defined outside the bounds of the video display, the screen will be defined in RAM outside the bounds of FD00H to FFFFH (Video RAM).

The definition of a screen is as follows:

a b c d SCREEN name

a = column number of upper left hand corner.
b = row number of upper left hand corner.
c = column number of lower right hand corner.
d = row number of lower right hand corner.
name = this can be any screen name desired by the user.

```
0 0 15 15 SCREEN S1 (NEW LINE)
```

This command will create a screen starting in row 1, col 1 and continuing to row 16, col 16.

To display something on this screen, type:

```
" HELLO THERE " S1 .W   (NEW LINE)
OK
```

Screens are defined in the dictionary, so they can be disabled by FORGETting them just as you would forget any FORTH word.

```
FORGET S1       (NEW LINE)
```

would disable screen S1

Every screen has a name and this name serves as an identifier which must be used with certain commands that deal with different screens. Some of these words are REV, .W, .C, and will be discussed in later sections of this manual.

When you power-up, two screens are already defined. The dominant screen is the console screen or execution screen and it has an identifier of C0. The editor screen which is enabled by SHIFT/EDIT has an identifier of ED. To use words which direct output to different screens use these identifiers.

```
C0      Console screen identifier.
ED      Editor screen identifier.
```

3.4 Fast, Slow, Auto

ZX81-FORTH still accepts the SLOW and FAST commands as does Sinclair's BASIC. In FAST mode, the video display is turned off until the CPU finishes processing the program. In SLOW mode, the video display always remains on but only 20% of the processing time is used to execute the program, the other 80% is used to update the display.

The individual keys no longer initiate the SLOW and FAST commands, instead you must type them out letter by letter.

ZX81-FORTH also supports the command AUTO. AUTO will interrupt the video display if the processor requires more then 1/4 second to execute a program.

It is possible to make any screen reverse video. The word REV along with the screen identifier is used to toggle the screen from reverse video to normal or vice versa.

```
CO REV     will make the console screen reverse video.
ED REV     will make the editor screen reverse video.
```

REV Executed after a screen identifier to reverse the dominant background field. (reverse video or normal video).

4.1 Storage

ZX81-FORTH allows for the storage of screens or a series of screens on magnetic cassette tape. The whole editor screen will be saved, therefore make sure that only the information you want to save appears on that screen.

Both LOAD and STORE will temporarily stop video output to the screen. The timing required to store or load screens requires all of the processor time, and because no interrupts are issued during the cassette routines, all tasking is suspended.

Storage takes place from the editor screen. (This is the portion of the video display above the black band as shown in fig.2.6). Each screen is loaded with an identifying number. You should take care to remember which number a specific screen is, so that if a large number of screens are stored on a tape each one will not have to be viewed to find the information you want.

Simplest case:

Fill the editor screen with any information which you desire. After you are finished go to the execution screen (type SHIFT/EDIT).

Such a screen might look like this:

```
THIS  SCREEN WILL BE AN  EXAMPLE
SHOWING   HOW  ONE  MIGHT  STORE
INFORMATION ONTO THE TAPE.  NOTE
THAT  THIS SCREEN DOES NOT  HAVE
TO  BE A PROGRAM.   A SCREEN  IS
512 BYTES  OF  INFORMATION.  YOU
CAN  STORE ANY SECTION OF MEMORY
BY MOVING IT TO THE VIDEO RAM OF
THE  EDITOR SCREEN (WHICH STARTS
AT FD00 HEX.)
```

```
10 STORE  (NEW LINE)
```

Remember that there is actually a five second leader on most cassette tapes which cannot be taped over. Therefore, advance the tape at least ten seconds before storing information.

STORE Takes a number off the parameter stack, in this case ten, and stores the editor screen with the number as an identifier.

You can only store information from the editor screen. You can not store information from the execution screen (also referred to as the console screen). This should not be a problem, because for most large programs, you will be working in the editor screen.

4.2 Retrieval

To retrieve information from the tape, the LOAD command is used. As an example, to retrieve the same screen we just loaded in the last section, the command is:

```
10 LOAD   (NEW LINE)
```

LOAD This command takes the number on the parameter stack as the screen number to be loaded from the tape. The routine will continue to look for the screen until it is found or until it is interrupted by hitting the space key.

After typing 10 LOAD, rewind the tape, then press PLAY and wait for the computer to read the tape. When the screen is read from the tape, the editor area will contain the same information that it contained when it was loaded onto the tape.

Typing 0 STORE will ensure that the screen will be the first loaded no matter what number is specified with LOAD.

Typing 0 LOAD will load the first screen found on the tape regardless of screen number. Also, all subsequent screens (after the first screen) can also be loaded using the --> command described in the next section.

ZX81-FORTH allows any information on the editor screen to be stored and loaded. The contents of the screen do not have to be FORTH words or definitions. ZX81 BASIC does not allow this. With ZX81-FORTH you have a way to store any information that you wish (letters, etc.).

4.3 Loading and Compiling Screens

When loading any program from the tape to the editor screen, all of the code on that screen can be compiled as soon as it is loaded.

CON The command to turn on the automatic screen compiler.
COFF The command to turn off the automatic screen compiler.

As an example, create a simple program on the editor screen and store it on the tape as screen 1. Now use the following commands:

CON 1 LOAD (NEW LINE) This will automatically compile the screen which has been loaded from the tape.

The screen compiler defaults to off on initial power-up, or on a COLD restart.

4.4 Sequential Screen Loading

There may be many times when your FORTH program is longer than one screen. When this happens each screen must be loaded and compiled before the next screen can be loaded. It is important that you store your screens in increasing sequential order if you want to load and compile them in sequence.

To store screens onto the tape in sequential order, you may use the following command:

<-- This command stores the present editor screen with a screen number which is one larger than the last screen stored.

In order to load screens sequentially, the command is:

--> This symbol, when placed at the bottom of the editor screen and compiled, increments the screen count and loads the next screen.

PAGE This is an INTEGER variable which contains the most recently accessed STOREd or LOADed page number.

BLK This INTEGER contains the address for the address to which the tape will download and from which the tape is loaded. The default value is the origin of the display buffer.

Warning: If you are compiling each screen as it is loaded sequentially, you must give the compiler enough time to compile each screen.

There are two ways to do this:

1. Stop the tape after each screen has been loaded and is compiling. When the computer is ready to load another screen (horizontal lines appear) restart the tape.

2. If large blank spaces are left on the tape when you are saving sequential screens, there should be enough time to compile each screen before the next one is to be loaded.

Either of these methods should ensure that the next screen on tape will not be "played" before the computer is ready to receive it.

5.1 Stack Structure

FORTH is different from most other computer languages in that it uses a stack. A stack is a data structure which stores things in the order in which they were entered. Items can be removed with the last item first. Here is an example:

Configure your screen so that you are on SCREEN 1, or the execution part of SCREEN 2. Enter three numbers:

```
0 1 2  (NEW LINE)    The stack looks like this:

2     top
1
0     bottom
```

To display the top item on the stack, simply type :

```
.     (NEW LINE)
```

This will remove 2 from the stack and display it. Typing :

```
.     (NEW LINE)      a second time will display the 1.
```

All the mathematical operations are also performed on the stack. To examine this, type the following:

```
0 2 3      (NEW LINE)      The stack looks like :

3
2
0
```

Now type :

```
*     (NEW LINE)      This command takes the top two items off
```

of the stack, multiplies them, and puts the result back on the stack. The result is :

```
6
0
```

If another multiplication were performed, then :

```
*     (NEW LINE)      Would leave a :

0                     ( 6 * 0 = 0 )
```

Most of the commands in ZX81-FORTH use a stack. ZX81-FORTH has a separate 8-bit character stack and a 16-bit number stack. You will be using the number (or parameter) stack most of the time. The character stack is discussed in more detail in the section on character input and output.

5.2 Dictionary and its Use

After reading the last section you should have a feel for how the number stack works. ZX81-FORTH words are stored in another place in memory, the dictionary. The dictionary grows upwards from 4000H. Every FORTH word is stored in memory with a header. The header contains the number of characters in the word plus the characters of the word itself. The number of characters plus the characters themselves are used by the outer-interpreter when a search of the dictionary is made for a word.

The programmer can create new words in the dictionary using various compiling words. These words are described in detail in Section 14.

Here is an example of how a new word would be defined using the COLON and SEMI-COLON compiling words. (Known as a Colon Definition). To create a word which takes the average of two numbers on the stack and displays the result, type :

```
: AVG  + 2 / . ;     (NEW LINE)      on the execution screen.
```

or

```
: AVG  + 2 / . ;     (SHIFT/E)       on the editor screen.
```

The above program computes averages by adding the two values on the stack and then pushing 2 on to the stack and performing a divide. The dot (.) then takes the value off the stack and displays it.

The word AVG can now be used to take the average of two numbers. Find the average of 86 and 46 by typing :

```
86 46 AVG       (NEW LINE)
66 OK
```

The answer of 66 is displayed to the screen.

VARIABLEs and INTEGERs (constants) can also be created in the dictionary. This is covered in Section 10.

5.3 Command Format

Many of the descriptions of the ZX81-FORTH words will be of the following form :

```
top --> stack                    stack <-- top
        before      COMMAND      after
      execution                execution
```

Each word is described by an example. The state of the stack is shown before and after the word is executed. The words are first described in a generic format and then an example of each one is given.

What the symbols mean :

```
    n = 16 bit number (n1, n2, n3 etc.)
    d = 32 bit number (d1, d2, d3 etc.) Sometimes nlow and nhigh
              are used to describe how double numbers appear on
               the stack or in the dictionary.
    u = unsigned 16 bit number.
 addr = represents an address in memory.
    b = byte.
    c = character.
    f = boolean flag (0= false, 1= true).
```

6.0 Mathematical Commands

ZX81-FORTH uses integer arithmetic. For some this may be inconvenient at first. However, one of the commodities a computer has is speed. Often it is desired that operations be performed as quickly as possible, perhaps because the calculation is done many times per second, and integer arithmetic is much faster than floating point arithmetic. If you need more accuracy in your values, the values can be scaled by a factor of 100 or 1000. Scaling by 100 would allow you to include pennies in calculations based in the pound and pence system.

Most 16 bit arithmetic is signed arithmetic in ZX81-FORTH. However, most 32 bit, and all 64 bit arithmetic is unsigned. This may seem to present a problem if you are not keeping track of the approximate magnitude of your calculations.

Here is the difference between signed and unsigned arithmetic. Below is listed a chart showing the difference, with the Binary and Hex formats of the numbers shown. This can be extended to 32 bit and 64 bit numbers.

UNSIGNED	BINARY	HEX	SIGNED
65535	1111111111111111	FFFF	-1
65534	1111111111111110	FFFE	-2
..	....	..	..
..	....	..	..
32768	1000000000000000	8000	-32768
32767	0111111111111111	7FFF	32767
..	....	..	..
..	....	..	..
0	0000000000000000	0000	0

Table 6.1
Unsigned, Signed, Binary, and Hex Numbers

+ (Addition) Adds the top two stack items n1 and n2 leaving the sum on the stack.

n1 + n
n2

```
1 1 + . (NEW LINE)
2 OK
```

- (Subtraction) Subtracts the top item, n1 from the second item, n2 and leaves the result on the stack.

```
n1 - n                          2 1 - . (NEW LINE)
n2                              1 OK
```

* (Multiplication) Multiplies the top two items, n1 and n2, and leaves the product on top of the stack.

```
n1 * n                          2 4 * . (NEW LINE)
n2                              8 OK
```

/ (Division) Divides the second item, n2 by the top item, n1, and leaves the result on top.

```
n1 / n                          6 2 / . (NEW LINE)
n2                              3 OK
```

2* (Multiply by 2) This multiplies the top stack item by two.

```
n1 2* n                         3 2* . (NEW LINE)
                                6 OK
```

2/ (Divide by 2) This divides the top item by two.

```
n1 2/ n                         4 2/ . (NEW LINE)
                                2 OK
```

ABS (Absolute Value) Leaves the absolute value of the top item on top of the stack.

```
n1 ABS n                        -12 ABS . (NEW LINE)
                                12 OK
```

MAX (Maximum) Finds the larger of the two top stack items and leaves it on top of the stack.

```
n1 MAX n                        9 4 MAX . (NEW LINE)
n2                              9 OK
```

MIN (Minimum) Finds the smaller of the top two stack items and leaves it on top of the stack.

```
n1 MIN n                        9 4 MIN . (NEW LINE)
n2                              4 OK
```

MINUS (Unary minus) Changes the sign of the top stack item.

```
n1 MINUS n                    31 MINUS . (NEW LINE)
                              -31 OK
```

+- (Swap Sign) Applies the sign of the top item, n1, to the second item, n2 and leaves the second item at the top of the stack.

```
n1 +- n2 (signed)             2 -3 +- . (NEW LINE)
n2                            -2 OK
```

MOD Performs the division, n2/n1, and leaves the 16 bit remainder on the stack.

```
n1 MOD nr                     15 4 MOD . (NEW LINE)
n2                            3 OK
```

/MOD Performs the division, n2/n1, and leaves the remainder n1 on top, and the quotient, n2, as the second item.

```
n1 /MOD n1                    5 2 /MOD . (NEW LINE)
n2      n2                    1 OK     . (NEW LINE)
                              2 OK
```

*/MOD Multiplies the second, n2, and third, n3, items and divides by the first, n1, leaving the remainder on the top of the stack with the quotient below it. (Signed arithmetic).

```
n1 */MOD nr                   3 3 2 */MOD . (NEW LINE)
n2       nq                   1 OK        . (NEW LINE)
n3                            4 OK
```

M* This multiplies two 16 bit numbers, n1 and n2, and leaves a 32 bit result, d.

```
n1 M* d                       20000 20000 M* D. (NEW LINE)
n2                            400000000 OK
```

M/ This divides a 32 bit number, d, by a 16 bit number, n, leaving a 16 bit remainder, nr, on top of the stack and a 16 bit quotient, nq, as the second item.

```
n M/ nr                     400000001. 20000 M/ . (NEW LINE)
d    nq                       1 OK    . (NEW LINE)
                              20000 OK
```

MD* This multiplies two 32 bit numbers on the stack leaving a 64 bit result.

```
d1 MD* dlow
d2     dhigh
```

MD/ This divides a 64 bit number by a 32 bit number, d1, leaving a 32 bit remainder, dr, and a 32 bit result, dq.

```
d1    MD/ dr
dlow      dq
dhigh
```

D* This multiplies two signed 32 bit integer numbers together and leaves a 32 bit signed result on the stack.

```
d1 D* d                    -30000. 25000. D* D. (NEW LINE)
d2                         -750000000 OK
```

D/ This takes a 32 bit unsigned number and divides it by a 32 bit unsigned number generating a 32 bit remainder and a 32 bit quotient.

*/ Multiplies the second item, n2, and the third item, n3, and then divides by the first, n1, leaving the result on the stack.

```
n1 */ n                    4 6 3 */ . (NEW LINE)
n2                         8 OK
n3
```

D+ (32 bit add) This is a double precision add which adds the top 32 bit numbers found on the stack.

```
d1 D+ d1                   400000. 40000. D+ D. (NEW LINE)
d2                         440000 OK
```

D- (32 bit subtract) Performs a double precision subtraction of the top 32 bit item from the second 32 bit item.

```
d1 D- d1
d2
```

DABS (32 bit ABS) This operation takes the absolute value of the 32 bit number on the stack.

```
d1 DABS d2
```

DMINUS (32 bit MINUS) Changes the sign on a 32 bit number.

```
d1 DMINUS d2
```

```
40000. DMINUS D. (NEW LINE)
-40000 OK
```

U* (Unsigned *) This multiplies the two unsigned numbers found on the stack, leaving an unsigned result.

```
u1 U* u
u2
```

```
6000 6 U* U. (NEW LINE)
36000 OK
```

UMOD (Unsigned MOD) The second unsigned 32 bit item, ud2, is divided by the top number, u1, leaving the unsigned remainder.

```
u1 UMOD u
ud2
```

U/MOD (Unsigned /MOD) The second unsigned 32 bit item, ud2, is divided by the top number, u1, leaving the remainder on the top of the stack and the quotient as the second item.

```
u1 U/MOD ur
ud2      uq
```

7.1 Logical Operators

AND Performs a bitwise AND of the two 16 bit items on the stack.

```
13    00001101        (8 bit example)
 7    00000111
AND
 5    00000101        result
```

OR Performs a bitwise OR of the two 16 bit items on the stack.

```
5     00000101        (8 bit example)
9     00001001
OR
13    00001101        result  (13= 0D hex)
```

XOR Performs a bitwise exclusive-or, XOR, of the two 16 bit items on the stack.

```
5     00000101        (8 bit example)
7     00000111
XOR
2     00000010
```

7.2 Comparison Operators

< If the second stack item is less than the first, the operation leaves a 1, otherwise it will leave a 0.

```
u1 < f              0 2 < . (NEW LINE)
u2                  1 OK      true
```

> If the second stack item is greater than the first, the operation leaves a 1, otherwise it will leave a 0.

```
u1 > f              0 2 > . (NEW LINE)
u2                  0 OK      false
```

0= Tests whether the top item is 0. If it is, then the operation leaves a 1, otherwise it will leave a 0.

```
u1 0= f             13 0= . (NEW LINE)
                    0 OK      false
```

0> Tests whether the stack item is positive. If it is, then the operation leaves a 1, otherwise it leaves a 0.

```
u1 0> f
```

0< Tests whether the stack item is negative. If it is, then the operation leaves a 1, otherwise it leaves a 0.

```
u1 0< f
```

= Tests whether the top two stack items are equal. If they are, the operation leaves a 1, otherwise it leaves a 0.

```
u1 = f              37 DUP = . (NEW LINE)
u2                  1 OK           true
```

C= Just like = but it only checks the least significant byte of the two 16 bit stack items.

```
c1 C= f             259 3 C= . (NEW LINE)
c2                  1 OK           true  (259 mod 256 = 3)
```

D< Checks if the second 32 bit item is larger than the first 32 bit item. If the operation is true, then a 1 is left, otherwise a 0 is left. This is an unsigned comparison.

```
d1 D< f
d2
```

D> Checks if the second 32 bit item is smaller than the first 32 bit item. If the operation is true, then a 1 is left, otherwise a 0 is left. This is an unsigned comparison.

```
d1 D> f
d2
```

D= Checks if the two 32 bit items on the stack are equal. If they are, a 1 is left, otherwise a 0 is left.

```
d1 D= f
d2
```

DMAX Leaves the larger of the two 32 bit items on the stack. This is an unsigned operator, so -2. will be greater than 2.

```
d1 DMAX d
d2
```

DMIN Leaves the smaller of the two 32 bit items on the stack. This is an unsigned operator.

```
d1 DMIN d
d2
```

D0= Checks whether the 32 bit item on the stack is equal to 0. If it is, a 1 is left, otherwise a 0 is left.

```
d D0= f
```

U< This is an unsigned less than comparison of the two 16 bit items on the stack. If u2 is less than u1, then a 1 is left, otherwise a 0 is left.

```
u1 U< f
u2
```

8.1 Number Bases

FORTH is capable of working in any number base. This is not so difficult to achieve, however, as the microprocessor can only work in Binary. This means that a conversion process must be done to work in Decimal or Hex. Once you have that conversion process, it is not difficult to extend it to cover all number bases. FORTH refers to a variable called BASE during numerical conversion, and changing base is as simple as changing the contents of the variable BASE.

The default base on initial power-up is Decimal. (Base 10, 0 to 9)

DECIMAL — Sets the current base to decimal.

HEX — Sets the current base to hexadecimal. (Base 16)

BASE — An INTEGER variable used to contain the current base of the system.

```
n1 TO BASE (NEW LINE)      Makes the current base n1.

BASE . (NEW LINE)          Places current base on to the stack
n1 OK

3 TO BASE (NEW LINE)       Changes the base to 3
DECIMAL 532 3 TO BASE . (NEW LINE)
201201 OK                  (201201 is 532 in base 3)
```

After DECIMAL and HEX, the most useful number base is BINARY and a definition of BINARY would be :

```
: BINARY  2 TO BASE ;
```

8.2 Stack Manipulation

. A dot "." prints the top 16 bit item on the output device (video screen).

```
1 2 3 . (NEW LINE)
3 OK      prints the top item
```

D. (Double number display) A double number (32 bit item) is taken off of the top of the stack and displayed on the screen.

U. (Unsigned number display) An unsigned number is taken off of the top of the stack and displayed on the screen. An unsigned 16 bit number ranges from 0-65535 whereas a signed 16 bit number ranges from -32768 to +32767.

S->D This is a 16 to 32 bit sign extension word.

```
n1 S->D d1          45 S->D D. (NEW LINE)
                    45 OK
```

D->Q This is a 32 to 64 bit sign extension word.

DROP Drops the top stack item.

```
n1 DROP n2
n2
```

2DROP Drops the top two stack items, or a double number.

DUP Copies the top stack item.

```
n1 DUP n1
       n1
```

?DUP Duplicates the top item only if it is non-zero.

OVER Copies the second stack item to the top.

```
n1 OVER n2
n2      n1
        n2
```

PICK Copies the stack item indexed by the top stack item, and places it on top of the stack.

```
n1 3 PICK n3
n2        n1
n3        n2
          n3   (a 2 PICK is the same as an OVER)
```

SWAP This word interchanges the top two stack items.

```
n1 SWAP n2
n2      n1
```

DSWAP This word interchanges the top two 32 bit stack items.

```
d1 DSWAP d2
d2       d1
```

ROT This word rotates the top three stack items. Item 3 goes to the top, and the remaining two items are pushed down the stack.

```
n1 ROT n3
n2     n1
n3     n2
```

9.0 Memory Commands & Memory Manipulation

FC56H is an address containing the present memory size connected to the ZX81. To display the memory size type the following :

```
HEX FC56 @ DECIMAL . (NEW LINE)
16384  OK          (This is the memory size of a
                    system with a 16k RAM-Pack)
```

MEM — This word places the amount of memory currently available to the system onto the stack.

```
DECIMAL MEM . (NEW LINE)
14976  OK          (This is the memory available to a
                    16k system at power-up)
```

VLIST — This will display all FORTH words currently found in memory.

SP@ — This will put on to the stack the current address of the stack pointer.

ALLOT — This word takes a number from the stack and reserves that many bytes in the dictionary.

```
0 VARIABLE V1 22 ALLOT  (NEW LINE)
OK          (When executed, V1 will now place the
low address on the stack, of a 24 byte block of
RAM in the dictionary which could be used for
arrays or character strings, etc. 22 bytes were
reserved by ALLOT and 2 were reserved by VARIABLE
to give 24 in total.)
```

@ (SHIFT/E) — This fetches the value at the memory location addressed by the top stack item, and places it on the stack.

addr @ n1 — A practical example might be :

```
30 +ORG @       (30 +ORG references the
address of the system variable that stores the
start address of the display buffer, which is
usually FD00H)
```

! (SHIFT/W) — This word stores the second stack item in the memory address specified by the top stack item.

n1 ! addr

```
HEX 32 FD00 ! (NEW LINE)
OK
```

This puts an "R" in the upper left hand corner of the video display in a 32k or less system.

X81-FORTH does not guard against you storing values in dangerous areas, such as the system variables, so be careful to store only in free memory.

?

Fetches and displays the contents of the address on top of the stack.

V1 ? (NEW LINE) (If V1 is a variable, ? will print its contents.)

+!

Increments the contents of the memory location addressed by the top stack item, by the second stack item.

addr +! 25 V1 +! (This will add 25 to V1)
n

C!

Stores a one byte item into the location addressed by the second item on the stack.

b addr C!

C@

Fetches a one byte item from the location addressed by the top stack item and places it onto the stack.

addr C@ b

COPY

This copies one screen of information (512 Bytes) to the address on top of the stack from the address found as the second item on the stack.

addr1 COPY
addr2

```
0 VARIABLE SCR1 510 ALLOT  (NEW LINE)
FBUF SCR1 COPY      (NEW LINE)
```

This will store the contents of the editor screen into a memory buffer called SCR1. To recall that information all you have to do is type SCR1 FBUF COPY (NEW LINE)

MOVE

This word is used to move blocks of memory around the system. It will take 3 items from the stack. The first is the number of words (2 bytes) you want to move, the second is the destination address, and the third is the source address. A routine which will do the same thing as the example for COPY above is :

FBUF SCR1 256 MOVE (NEW LINE) This takes the contents of the editor screen and moves it to SCR1. 256 words is 512 bytes.

FILL — This word is used to fill areas of memory with a specified byte. The FILL word takes three items from the stack. The first is the byte which is to fill memory, the second is the number of bytes to be filled, and the third is the starting address of the area to be filled.

FBUF 512 14 FILL (NEW LINE) Will fill the edit screen with dots. (Using Sinclair's non-ASCII character codes.)

BLANKS — This word is just like FILL but it fills memory with 0's and only uses two items from the stack.

FBUF 512 BLANKS Will blank out the editor screen. (Using Sinclair's non-ASCII codes.)

FBUF — This is an INTEGER value which contains the base address of the display buffer. To access the display buffer all one needs to do is type FBUF and the address of the display buffer will be placed on the stack. To change FBUF all you need to do is put the new buffer address onto the stack and type TO FBUF.

+ORG — Adds the item found on the stack to the address of the beginning of the system variables and is most commonly used to access the system variables.

D! — Stores a 32 bit number at the address found at the top of the stack.

```
addr D!
nlow
nhigh
```

D@ — Fetches the 32 bit number found in the location addressed by the item at the top of the stack.

```
addr D@ nhigh
        nlow
```

PAD — Execution of this word places the address of a 64 byte scratch-pad on to the stack. This pad may be used for temporary storage by the user. The PAD will overlay other parts of ZX81-FORTH if the 64 byte limit is exceeded, so be careful!

10.1 Data Types

ZX81-FORTH as you probably realise by now operates on either 16 bit or 32 bit integers. There are operators for both types of numbers. Integers are one type of data, while another type is character data. ZX81-FORTH characters are standard ASCII characters and can be found in Section 12.5. On some systems floating-point numbers are included. For space reasons, ZX81-FORTH does not include floating-point arithmetic, but an extension ROM will be available.

10.2 Variables

VARIABLE is used to create a variable which references a memory location used to store two bytes of information, usually a value. When ever the variable so created is executed, the address of that variable is placed on the stack. An initial value is always assigned to the variable when it is created, and this value can be changed at will.

```
value VARIABLE varname

     value = the initial contents of the variable.
     varname = is a name chosen by the programmer.

0 VARIABLE AVG       (NEW LINE)
OK     (0 is the initial value and AVG is the name)
AVG @ .    (NEW LINE)
0 OK  (AVG  places the address of AVG on the stack,  @  gets
       the contents and . prints it to the screen)
```

The user can also create 32 bit variables using the word 2VAR. This creates a 4 byte variable, and behaves the same way as the other variable.

10.3 Integers

INTEGER This word is very much like variable. It creates a 2 byte variable in the dictionary, but instead of placing the address of the variable on the stack, it places the actual value.

value INTEGER intname

```
100 INTEGER INT1     (NEW LINE)
OK          (Here we have created an integer variable
                with an initial value of 100)
```

In ZX81-FORTH, the word INTEGER behaves identically to the word CONSTANT in other FORTH's. It has been used to keep variables because of the space saving such a method allows.

TO Allows the user to change an INTEGER value.

3 TO BASE changes BASE's contents to 3

10.4 Arrays

Arrays can be created in FORTH just as they can in BASIC, FORTRAN, or other languages. First, space must be allocated in the dictionary.

> 0 VARIABLE VAL 22 ALLOT (NEW LINE) This will create a variable called VAL. It then gives VAL an initial value of 0 and reserves 22 additional bytes for it in the dictionary. This gives VAL the capacity to hold 12 16 bit numbers (24 bytes).

To access any 2 byte value in the array put the array item you wish to access on to the stack and use the following commands.

> 2* VAL + @ (NEW LINE) This will access the proper array value by doubling the index, adding it to the address, and fetching the proper number from that address.

This is not the only way to construct arrays. A more efficient and elegant way is to use the <BUILDS ... DOES> construct. This method will be shown in Section 14.2. As data structures, arrays are of fundamental importance in implementing solutions to programming problems.

11.0 Control Structures

Unlike other FORTH versions, ZX81-FORTH allows the user to use the IF .. ELSE .. THEN and the ... DO ... LOOP and the other statements outside of a colon-definition. It does this by creating a headerless word which executes immediately.

IF .. ELSE .. THEN This is a special structure used to create logical branches. IF checks the top entry on the stack. If the top stack entry is non-zero, (true) the code between the IF and ELSE is executed. If the top entry is zero, (false) the code between the ELSE and the THEN is executed. For example :

```
1 IF ." TRUE " ELSE ." FALSE " THEN  (NEW LINE)
TRUE OK  (The words between IF and ELSE is executed.)

0 IF ." TRUE " ELSE ." FALSE " THEN  (NEW LINE)
FALSE OK  (The words between ELSE and THEN is executed.)
```

IF .. THEN This is a simpler construct than the IF .. ELSE .. THEN construct. This statement allows the execution of code if the value on the stack is non-zero (true). Another example :

```
1 IF 1 1 + . THEN (NEW LINE)
2  OK
```

This displays the addition of 1 and 1, because the test is true, having found a zero value on the stack before the IF. If the initial stack value were zero the code would not be executed.

DO .. LOOP The DO LOOP uses the top two indicies on the stack followed by executable code within the DO and LOOP words.

```
limit initial DO .. code .. LOOP
```

limit = the upper limit of the loop count.
initial = the lower limit of the loop count.

The index is incremented by one from the initial value to one less than the limit. The value of the index is accessable via the word I.

For example :

```
9 0 DO I . LOOP  (NEW LINE)
0 1 2 3 4 5 6 7 8 OK
```

should be displayed. Note that the upper limit 9 does not get executed.

DO ... +LOOP This construct allows the user to increment or decrement the count by any value and it looks like this :

```
limit initial DO .. code increment +LOOP

limit = is the upper or lower limit for the loop count.
initial = the value where the count is started.
code = any FORTH word or words.
increment = any positive or negative value.

-5 0 DO I . -1 +LOOP   (NEW LINE)
 0 -1 -2 -3 -4 OK
```

LEAVE This terminates the loop at the next LOOP or +LOOP. It could be used in an IF THEN clause.

For nested loops a second index is available, the index J. For further nested loops, the NI index can be used.

```
0 NI corresponds to I
1 NI corresponds to J
2 NI would correspond to the next index and so on.
```

CASE is most often used in a definition, however it can also be used interactively on the execution screen. The command format is as follows :

```
CASE e0 e1 e2 e3 e4 ;     (2 or more statements may be used)

STACK VALUE          EXECUTION PRIORITY
     0                    e0
     1                    e1
     2                    e2
     3                    e3
     etc.                 etc.
```

The statement at e0 is executed when the stack value before the CASE has a value of 0, e1 when the stack value is 1, etc. e1 may be a FORTH definition, or any mathematical expression. To test a CASE type :

```
2 3 (NEW LINE)
OK
1 CASE * + ; . (NEW LINE)      (this will execute the +)
5 OK
```

BEGIN .. code .. AGAIN This statement will execute any code found between the BEGIN and AGAIN words. When AGAIN is reached control is transferred to BEGIN and the code is executed again thus creating an infinite loop.

```
BEGIN ." HELLO " CR AGAIN  (NEW LINE)   and will print out
HELLO
HELLO
HELLO
HELLO
..
..
```

Hello's will continue to be printed to the bottom of the screen, and then the print will continue by scrolling.

On most machines that will be the end of the matter, as the only way to restart FORTH would be to switch off the machine and reload, but in ZX81-FORTH a SHIFT/SPACE (break) will return you to the normal keyboard. This because the keyboard is a system task!!

```
BEGIN .. code .. flag UNTIL

code = any FORTH word or words.
flag = a logical operation which leaves a true or false
       value on the stack which is tested by UNTIL. If the
       flag is true, (non-zero) the loop is terminated,
       otherwise the execution flow returns to BEGIN and
       carries on through the loop again. For example :

0 BEGIN 1 + DUP DUP . 9 = UNTIL  (NEW LINE)
1 2 3 4 5 6 7 8 9 OK
```

This routine takes the top stack value, (initially a 0) increments it by one, duplicates it twice in order to save the value before displaying it, and then it performs the logical operation, a comparison to 9. In this case the 9 is printed out. This is done because the FORTH statements are executed before the UNTIL checks the top value of the stack against the 9.

BEGIN .. WHILE .. REPEAT The construction of this word is as follows :

```
BEGIN .. words .. test WHILE .. words .. REPEAT

words = can be any FORTH word or words.
test = is a logical operator which leaves a true or false
       value for WHILE to test.
```

For example :

```
BEGIN 1 + DUP DUP . 5 > WHILE ." END " REPEAT  (NEW LINE)
0 1 2 3 4 5 6 END OK
```

12.1 Character Stack

ZX81-FORTH is unique in that it has both number and character stacks.

The character stack store bytes of ASCII code and provides a more efficient and convenient method for storage and manipulation than a parameter stack on its own. The stack pointer for the character stack can be found in the system variable located at address FC84 hex. ZX81-FORTH uses the IY register of the CPU to hold the parameter stack pointer. These stacks are independant of each other, but in order to make use of the character handling routines of the system, character strings must maintain a certain format. A character string consists of two parts : the string of ASCII characters that reside on the character stack which actually makes up the string, and a number which sits on the parameter stack and is a count of how many characters that are stored in the character string on the character stack. Manipulating character strings is done through manipulation of the numbers on the parameter stack which represent the length of the character strings. For instance, to concantenate two character strings into a single character string, all one needs to do is add the two numbers on the parameter stack together to generate a number which represents one composite character string. Or simply put, a + concantenates strings.

12.2 Character Commands

String I/O

." Defines the beginning of a string of characters to be output to the screen. Any characters found between a ." and a " will be placed into a character string and output to the console device (the execution screen). For instance :

: MESS ." THIS IS A MESSAGE " ; Will print THIS IS A MESSAGE to the console each time MESS is encountered in a program or typed on the execution screen.

" This works just like ." except instead of taking the character string and outputing it to the console device, the " will leave the character string on the character stack to be manipulated either by another routine or directed to another screen. The length of the string will be found on the parameter stack immediately after the " is executed. The parser expects a space immediately after tne first " and does not count it as a character. Both the ." and " use the " as a delimiter to mark the end of the input character string. Example :

." THIS IS A STRING " When placed in a definition or in the execution screen will display the string between the quotes.

" THIS IS A STRING " This will insert this string in the character buffer with the number of characters in the string placed on the parameter stack.

ABORT" Checks a flag taken from the parameter stack and if the flag is true (non-zero) then a user defined error message is placed between the final " in ABORT" and a warm restart is executed. This command could be useful for displaying user defined error text in a program. Example :

```
1 ABORT" ERROR 10 " (NEW LINE)
ERROR 10 OK

0 ABORT" ERROR 10 " (NEW LINE)
OK
```

CR Is used to print a carriage return, line feed on the console screen.

SP Will print a space to the console screen.

CLS Will clear the console screen.

EMIT This word is used to take an ASCII character from the parameter stack and output it to the console screen.

KEY Calls a routine which will get a value from the keyboard and put the ASCII value of that key onto the parameter stack.

S@ This word gets a word from the keyboard (ending with a space or CR) and puts that token or string onto the character stack.

W! This will take a character string and store it onto the address found above the character count of the string on the parameter stack.

" I WILL PUT THIS IN THE PAD " PAD W! will put the text into PAD. After the string is transferred to PAD, what the memory image looks like is a single byte character count followed by the character string. In other words, after executing the string above, typing a PAD C@ . will print out the number of characters in that string. (27 in this case).

W@ Will take an address from the parameter stack and fetch the character string stored at that address. It places the character string itself onto the character stack and the number of characters in the character string onto the parameter stack.

PAD W@ CO .W (NEW LINE) will print the character string found in PAD out to the screen console.

.W Allows output of a string to the console screen, editor screen, or any other user defined screen. The command expects to find the string on the character stack with the number of characters on the parameter stack and must be preceded by a valid screen identifier.

On the execution screen enter the following :

" THIS IS A STRING " (NEW LINE)

To display this to the console screen (the execution screen) type :

CO .W (NEW LINE)

To display this to the editor screen, type :

ED .W (NEW LINE) and to display to any other user defined screen type :

screen-identifier .W (NEW LINE)

.C This word follows the same format as .W but it works like EMIT. It uses two numbers from the parameter stack, the first is the screen identifier, the second is the ASCII value of the character that you want to output to the identified screen.

.CN This word is just like .W except that it always directs output the console screen and so it needs no screen identifier. .CN will then take a character string and output it to the console. A ." TEXT " is just like a " TEXT " .CN

CDROP This word drops a character string off of the character stack. It assumes that a character count is on the parameter stack as is usual.

CDUP This word will duplicate a character string on the character stack much the same as DUP does to the parameter stack.

.CO Is used to take a character string from the character stack and direct it to the keyboard input buffer just as though that character string had been typed in on the keyboard. This is used for a variety of things; one of which comes to mind is the dynamic self re-scheduling of tasks. A simple example of how .CO works is :

" VLIST " .CO (NEW LINE)
OK
VLIST etc. OK will take the character string VLIST and direct it to the keyboard just as if you had typed it.

12.3 Character/Number Stack

There are a group of words in ZX81-FORTH which use both the number/parameter stack and the character stack. These word types are described in this section. Remember that a character string always consists of two things; a number on the parameter stack which describes the length of the character string, and the character string itself which resides on the character stack.

C>N This word removes one character from the character stack and places that character's ASCII value onto the parameter stack. It will reduce the character count which is on the parameter stack by one and place the ASCII value of the character taken from the character stack on top of the character count. If the character string is empty this routine will leave the null character count at zero and return a 0 ASCII value.

N>C This does just the opposite of C>N. It takes an ASCII character value off of the parameter stack and places it onto the character stack. The character count for the character string that the ASCII character will be appened to, should be under the ASCII value on the parameter stack. The character count will be incremented by one to reflect the extra character on the character stack.

W> This word is used to format character strings for output. If you have a character string that is only 5 characters long and you want it to fill up eight spaces when it is printed, you could do this simply by the command 8 W>. This will take any character string of seven characters or less and append enough spaces to the beginning of it to make it eight characters long. VLIST uses this to get all the words into columns. A definition which will convert a number on the parameter stack to a formatted three character long character string and print it out could look something like this:

```
: .F  # 3 W> CO .W ;
```

If you use .F instead of . all your numbers will be printed out in at least 3 character long strings.

```
: TBL  CLS 11 1 DO 11 1 DO I J * .F LOOP CR LOOP ;
```

TBL is a definition which will generate a 10 by 10 multiplication table.

H>A Is useful when transforming HEX nibbles into ASCII equivalent characters. H>A takes a number off the parameter stack in the range 0-15 decimal and converts it into its appropriate ASCII equivalent and leaves the character on the parameter stack. A 13 H>A EMIT will echo a D onto the execution screen.

A>H Does just the opposite of H>A, it removes an ASCII value in the range 0-9, A-F and leaves on the parameter stack its HEX equivalent.

># This is a word which is used to attempt to convert characters in a character string to a number on the parameter stack. Let's say that the character string which represents 100 is found on the character stack, by executing a ># the character string will be converted to a 16 bit integer with a value of 100 and it will be placed onto the parameter stack. This word leaves a flag of 1 on the parameter stack above the converted number if the conversion is successful, otherwise the flag will be 0. If the conversion is unsuccessful the character the character string will be left unchanged and can be used to prompt the user for a correct character string. A definition which would use the full capabilities of this word follows :

```
: READ BEGIN ." ? " S@ ># WHILE
CR CO .W ." IS NOT GOOD TRY AGAIN " REPEAT ;
```

This word will prompt the user for input with a ? and leave the input number on the stack if the conversion is good, otherwise it will print out the original string and ask for further input until a good string is input.

Does the opposite of >#, it removes a character from the parameter stack and converts it into a character string in the current base. The dot "." word uses # and is defined as : . # CO .W ;

U# Is just like # but performs an unsigned conversion from a 16 bit number to a character string. U. is defined as : U. U# CO .W ;

D# Is the double number version of #. D. is then defined as : D. D# CO .W ;

12.4 Character Comparison

W= Expects two numbers from the parameter stack, both should be addresses of character strings which are to be compared character by character for equality. If the two strings which are pointed to are equal, a 1 will be placed on the parameter stack; if the two are not equal, a 0 will be placed on the stack. Both of the addresses will be removed before the flag is left.

S= Takes an address off of the parameter stack which points to a string which is to be compared to the character string on the character stack. This is much the same as W= except that here one of the character strings is on the character stack already. This routine removes the address but leaves the character string on the stack intact before it leaves the flag.

12.5 Keyboard Allocations

Key	Shifted Key	Shifted Key Function
1	EDIT	Toggles between screens
2	AND	Fetches a line from PAD
3	THEN	Puts a line into PAD
4	TO	Deletes a line from the editor screen
5	<-	Moves cursor left
6	↓	Moves cursor down
7	↑	Moves cursor up
8	->	Moves cursor right
9	GRAPHICS	Inserts a line on the editor screen
0	RUBOUT	Deletes one character
Q	""	Compiles an editor screen line
W	OR	Store word. Displayed as !
E	STEP	Fetch word. Displayed as @
R	<=	[character
T	<>	_ character
Y	>=	] character
U	$	$ character
I	(	(character
O	)	) character
P	"	" character
A	STOP	Clears the present screen
S	LPRINT	% character
D	SLOW	' character
F	FAST	\ character
G	LLIST	^ character
H	**	# character
J	-	- character
K	+	+ character
L	=	= character
NEW LINE	FUNCTION	Home cursor to top left corner.
Z	:	: character
X	;	; character
C	?	? character
V	/	/ character
B	*	* character
N	<	< character
M	>	> character
.	,	, character
SPACE	BREAK	WARM Restart, if held for 1/2 second COLD restart.

13.0 The Printer

P — Toggles the printer on or off. If the printer routine is on, any information that goes to the video display will also go to the printer. As an example, to get a listing of all the FORTH words presently in the dictionary, type :

P VLIST (NEW LINE) This prints to both the printer and the console)

PRTR — PRTR is to the printer device as EMIT is to the console device. By putting an ASCII value onto the parameter stack you can output that character to the printer by using PRTR. This works regardless of whether the P toggle is on or off.

.P — This word is written for the ZX-Printer or any other compatible printer which is made explicitly for the ZX81. What it does is look for a character string in PAD which is 32 characters long or less, and print that line out on the ZX-Printer. The word will pad the rest of an empty line out with spaces and print one complete line out to the printer. It will not do anything with a user defined printer routine.

" THIS IS A TEST " PAD W! .P (NEW LINE) will print

THIS IS A TEST out to your ZX-Printer.

PRINT — Is used when you have some other ASCII compatible device that you want your printer output to be directed to rather than the ZX-Printer. The routine which you write to interface into must remove the ASCII value from off of the parameter stack and use it to output to your printer device. As an example, let us say we have an RS-232 card attached to our system and we have written two routines to interface to that card. One of them is a routine which will return only if the RS-232 card is ready to accept another character for output; let us call this routine RS_READY. The other routine is simply the routine which will place the address of the RS-232 output port onto the stack, we will call this one RS_ADDRESS. Now we can use these in the following manner to output characters through the RS-232 port instead of the defaulted ZX-Printer.

: RS_OUT RS_READY RS_ADDRESS C@ ; This will output one character taken from off of the parameter stack to the RS-232 port.

PRINT RS_OUT This reassigns the printer output to the routine RS_OUT instead of the default routine.

14.1 Colon / Semicolon

FORTH is different from many other languages in that it allows the user to define his or her own words to extend the language. The user can completely customise a set of words which can then be used in any program.

The basic construct of colon definitions is

```
: wordname  program ;
```

In ZX81-FORTH 'wordname' is compiled into the dictionary as a word with a specific operation as defined after wordname and before semi-colon. Try ...

```
: AVG + 2 / ;  (NEW LINE)
OK
```

Successfully typing the above word will define a word which adds the top two stack items and divides by two. In other words, AVG finds the average of two numbers. To execute this word, type:

```
2 4 AVG .       (NEW LINE)
3 OK
```

The above statement will put 2 on the stack, then put 4 on the stack, then execute the commands as defined by AVG, and finally display the top stack item left by AVG. We will now define a word that takes the average of two sets of numbers by using AVG and then check to see if the averages are equal. It will also print an appropriate response.

```
: EQUAL  AVG  ROT ROT  AVG = IF
." EQUAL " ELSE ." NOT EQUAL "
THEN ;

24 24 12 6 EQUAL     (NEW LINE)
NOT EQUAL OK
```

The two ROT commands here are included to put the value calculated by AVG on the bottom of the stack and the next two numbers to be averaged on the top.

FORGET — You can forget any word in the dictionary with FORGET providing it is not protected by the FENCE value. Simply type :

```
FORGET word     (NEW LINE)
```

and the word along with any dictionary entries compiled after 'word' will be removed from the dictionary.

FENCE — You can protect any word from FORGET by typing :

```
FENCE word      (NEW LINE)
```

14.2 : word .. <BUILDS ... DOES> .. ;

This is one of the most important and powerful FORTH structures. With it you can define new defining words. What this means is that you can create new types of defining words, and using these words new types of data structures can be produced and great power can be given to the programmer. The format for this word is :

: new-defining-word <BUILDS definition code DOES> run-time code ;

defining-word = the name of the new defining word.
definition code = the code which is executed when the defining-word is used to create a new word.
run-time code = this code is executed when the new word is used as a command word.

It is possible in a <BUILDS DOES> construct to have no definition code or run-time code. As an example:

: ARRAY <BUILDS 20 ALLOT DOES> ; Here is an example of the construct with no run-time code. This statement will allow the user to create arrays of ten words (20 ALLOT sets aside twenty bytes in the dictionary, enough for ten 16-bit variables). ARRAY is now a compiling word which is a lot like VARIABLE except that it reserves twenty bytes in the dictionary for user variables instead of just two and also uses no initialiser. An example of how to use ARRAY follows:

```
ARRAY NUMBER   (NEW LINE)
```

This creates an array called NUMBER which will reserve twenty bytes for number storage in the dictionary. To access these twenty bytes we need some way to reference them, perhaps by placing the address of where they are found in memory onto the stack. It just so happens that this is exactly what executing NUMBER will do for us. NUMBER places the address of the first byte of the twenty bytes ALLOTed to NUMBER onto the stack.

The program can be expanded as shown below. We will create a one dimension array and will allow the user to access any number in the array by placing an index on the stack.

```
: ARRAY1 <BUILDS 40 ALLOT  DOES> SWAP 2* + ;
```

ARRAY1 is now a defining word which when used will create a twenty word array. Let's make one called XYZ

```
ARRAY1 XYZ      (NEW LINE)
```

We have now created in the dictionary an array called XYZ. We can insert a number, say 123, into the 11th word in this array by typing:

```
123 11 XYZ !    (NEW LINE)
```

What has happened up to this point ? First, a 123 was placed onto the stack. Second, the index 11 was placed on the stack, and third, XYZ is encountered. XYZ first places the address of memory in the dictionary where the array is located and then initiates the execution of the code following DOES>. In this case the top two stack items are swapped (putting the index on the top and the address below it). Next, we double the index with 2* because we are dealing with 16-bit values and address memory in 8-bit bytes. The next thing we do is add the offset of the index to the address already on the stack. After this is done, the stack contains the address of the indexed array member and the value to be stored there. A store (!) will finally put the value in the array.

Another routine could be written to fetch values from the array and would look something like this :

```
11 XYZ @  (NEW LINE)
```

Self Modifying Data Structures

A remarkable consequence of FORTH's ability to define new defining words is that we may build 'intelligent' data structures ; for example, arrays that automatically maintain averages, or lists that re-order themselves whenever any entry is altered.

To take the first of these examples, suppose we have a 10 element array 'READINGS' defined using a word similar to XYZ of the last example. To compute the arithmetic average of the contents of this array requires adding together all 10 entries and dividing by 10. A special definition could easily be written to do this as follows:

```
: AVERAGE                  (take average of array 'READINGS' )
        0
        11 0 DO
                I READINGS @ +
           LOOP
        10 / ;
```

If our FORTH application needed us to calculate an average like this often and for many different arrays then, to simplify the overall program, we should define a new defining word *ARRAY with the averaging function built into the DOES> part of the definition:

```
: *ARRAY                ( 'special' array with running average )
        <BUILDS
                DUP ,       ( save array size )
                0 ,         ( set 'average' to zero )
                0 DO        ( step through elements )
                    0 ,     ( defining and zeroing  )
                  LOOP
        DOES>
             DUP DUP @      ( get array size          )
             SWAP 4 +       ( point to start of array )
             OVER 0 SWAP
             0 DO           ( step through array )
                  OVER @ +        ( add up    )
                  SWAP 2 + SWAP   ( bump up pointer )
                LOOP
             SWAP DROP SWAP /     ( divide by array size )
             OVER 2 + !     ( store average in element 0 )
             2 + SWAP 2 * + ;     ( calculate address )
```

Arrays defined by *ARRAY may be used just like those defined by XYZ, for example :

```
10 *ARRAY READINGS

10 1 READINGS !             ( readings(1)=10 )

20 2 READINGS !             ( readings(2)=20 )

1000 10 READINGS !          ( readings(10)=1000 )


2 READINGS ?                ( print contents of readings(2) )
20 OK
```

Which is exactly how we would expect a 10 element array, with elements numbered from 1 to 10 to behave. But typing :

```
0 READINGS ?          103 OK
```

will print the average of the values currently contained in the array ((10+20+1000)/10 = 103). This average will be calculated afresh every time the name of the array 'READINGS' is executed and will always be true however many times we might have altered the values stored in the array.
For example :

```
870 10 READINGS !               ( alter readings(10) to 870 )

50 6 READINGS !                 ( set readings(6) to 50 )

0 READINGS ?                    ( new average is 95 )
95 OK
```

and, of course, <u>all</u> arrays defined by *ARRAY will have this function built in !!

14.3 Operating System Words

[_] This word is used to suppress the execution of an immediate word in a definition. The immediate word which follows [_] will, if in a definition, be compiled to execute when the word being defined is executed and not during the compilation of the word itself. (Functionally equivalent to the FIG word [COMPILE])

(Any words placed in brackets will not be compiled and will act as comments in your program. Anything entered up to a ")" will be entered as a comment.

, Stores the 16-bit number found on the parameter stack into the dictionary at the next available location.

C, This is like , but stores a byte into the dictionary rather than 2 bytes (16-bits).

HERE This word places the address of the next free dictionary space onto the stack.

H This places the address of the memory location which contains the address of the next free dictionary space onto the stack.

T Places the address of the memory location which contains the tail pointer of the dictionary onto the parameter stack.

HEAD Is used in creating new defining words. HEAD creates the dictionary header of a word and links it into the dictionary. HEAD generates no code field and thus if a word is created with HEAD and no attempt is made to place behind it a code field, execution of that word will crash the system. (Functionally equivalent to the FIG word CREATE).

IMM When embedded in a definition, IMM makes that word an immediate word and that word will execute during compile time, i.e. in a colon definition. IMM is used to customise compiler words which generate code of modify the dictionary without creating a header. (Functionally equivalent to the FIG word IMMEDIATE).

' Attempts to find the word following the character in the dictionary. When found, the address of that word is placed onto the stack.

```
' M* HEX .  (NEW LINE)
B00 OK
```

The machine code which makes up M* start at 0B00 in memory. The ' is SHIFT/D on the ZX81.

15.0 Time & the System Clock

Computers, as you are no doubt finding, are very useful and versatile tools which can do a surprising number of things. If you have been around people who do not know a great deal about these tools, you may have been asked : "What can your computer really do ? Can it cook dinner or vacuum the carpet ? What good is it if all you can do is stare at it ?" And these are good questions. Well, of course it can help with balancing the cheque-book, organising business information, generating mailing lists, calculating taxes, writing out cheques, playing games, and any number of other things, but there are a whole lot more things that a computer cannot do at all well because computers, at least in their simplest configurations, do not have eyes, hands, and/or a sense of time. In short, computers cannot be told to do what humans can do because they do not have the receptors and manipulators that we humans have. Most small computer systems also lack the ability to keep track of time.

It would be possible to give your ZX81 some "sensory" devices or transducers which would enable it to, in a limited way, perceive some things in its environment by attaching to it input ports or A/D converters etc. You could also give it hands, so to speak, by attaching to it output ports which could control something in its environment.

After having given your computer these things, it would still be necessary to give it a sense of time in order to link it to the way that the real world does things. For instance, if you were collecting data in your house by monitoring the temperature and using that information to control your boiler better, your computer would, in all probability because of its speed, have the ability to take a temperature reading once every 1/1000th of a second. Now it is obvious that gathering this much information would be useless and wasteful, but if you had a method of restricting the data-gathering process to read a temperature once a minute, the data you collected could be analysed more rationally and the whole project could be given a sense of orderliness.

The easiest way to give your computer a sense of time is to give it a clock that it can look at every time you tell it to and so enable it to make decisions about what to do and when to do it. ZX81-FORTH has just such a clock. It is made up of a system variable that is incremented every 1/50th of a second and counts from zero up to a limit that can be set to anything from 1/50th of a second to over two years. Both the clock itself (the system variable that is incremented) and the period (the changable limit) are made up of 32-bit integers which can be accessed by the two words :-

TIME Which will place the address of the system variable which contains the clock count for the system. This variable is a 32-bit integer value which is incremented each clock tick (1/50th sec) and continues until it reaches the limit set by the system variable accessed by the word PER. Upon power-up, this variable will default to zero. In order to see the number of ticks since the computer was switched on type :

```
TIME D@ D.
```

PER Will place the address of the 32-bit system variable containing the limiting value to which the system clock counts. It is through this variable that the system clock is given its overall period. This variable defaults to a count that represents 24 hours or one day.

Two examples of how to use the clock are given here. The word SET is used to set the clock with the current time and the word RTIME will enable you to display the time of day.

```
: READ BEGIN ." ?" S@ ># WHILE CR CO .W ."  BAD" REPEAT ;

: SET ." HOUR" READ TH ." MIN"  READ TM ." SEC" READ TS D+ D+
          TIME D! ." DONE" ;

: RD 60. D/ SWAP DROP ROT ROT ;

: CPT TIME D@ RD RD RD SWAP DROP ;

: COL # C>N DROP 2 W> " : " + ;

: RTIME CPT COL C>N DROP .CN  COL .CN COL .CN DROP ;

:  TIME-DIS  CLS BEGIN 13  EMIT RTIME AGAIN ;
```

The word READ is used to input one number to the stack much like an INPUT would be used in BASIC. The ." ?" outputs a prompt to the screen. The S@ reads a character from the input buffer (the keyboard), and the ># attempts a conversion. If it converts to a number ok, control will then pass out of the READ word and return with a valid number on the stack. If not the CR CO .W etc., will echo the rejected character and return to the start of the loop. The SET word prompts for input with an HOUR? and after a number has been input it runs TH which multiplies the 16-bit number on the stack by the number of ticks in an hour and leaves the result as a double number on the stack. The same is done for the minutes and seconds leaving three double numbers on the stack which are then added together with the D+'s and deposited in the master clock variable with the TIME D!. After it is all done it tells you by saying DONE.

The RD routine reduces a double number by dividing it by 60. It leaves the double quotient on the top of the stack and a 16-bit remainder under it. The CPT routine gets the 32-bit time from the computer master clock and runs it through the RD program to leave the ticks, seconds, and minutes on the stack. It then reduces what is left, the hours, to a single number, so CPT reduces the time to hours, minutes, seconds, and ticks on the stack. COL converts a number to a character string three characters long with a colon in the first location. RTIME puts the other routines together and displays the time in the format HH:MM:SS to the console. TIME-DIS just displays the time over and over in an infinite loop.

16.0 Tasking

ZX81-FORTH is fundamentally different from most other small computer operating systems in that it allows the user to task programs. Tasking is the act of scheduling a program to execute at some time in the future. Any program can be scheduled in a task, you can run approximately ten tasks simultaneously in the background before the system will slow down so much as to be useless in editing new programs. (Tasks use valuable processor time which is usually spent in editing new programs). How much the system slows down depends on what and how often tasks are run.

Tasks are set up in this way :

```
TASK task-name  program-name
```

where program-name is any word which exists in the dictionary and task-name becomes the name associated with the task you are defining. At this stage the task has been defined, but has not yet been scheduled to execute.

Scheduling Tasks

The user can schedule a task to run using the IN, EVERY, AT words. The time interval used can be :

```
TT   Task Ticks (1/50th second)
TS   Task Seconds
TM   Task Minutes
TH   Task Hours
TD   Task Days
TW   Task Weeks
TY   Task Years
```

IN — Task identifier used to schedule a task to execute after the specified time has elapsed.

EVERY — This word is used by the task scheduler to schedule a task to execute repetively using the period specified.

AT — Task identifier used in conjunction with the system clock to calculate the time existing between the current time and the time specified during scheduling so that the task will execute IN the appropriate time.

START — This word directs the task scheduler to clear the task overflow flag, the task execution flag, and the task execution queue to allow scheduling to continue.

STOP This word will set the task schedule overflow bit which in effect stops the task's execution and scheduling.

RUN Will increment the task execution queue if the overflow bit has not been set. The net effect is to schedule the task to execute the next clock tick if no other higher priority task is executing.

The format for task scheduling is :

```
command number time-type task-name

TASK   TASK1   program-name

EVERY 5 TS TASK1
```

This schedules the task TASK1 to execute every 5 seconds.

```
IN 10 TM TASK1
```

This schedules the task TASK1 to run in ten minutes. So now TASK1 is scheduled to execute every 5 seconds after ten minutes. The system will automatically start scheduling this task upon the execution of this command.

To terminate the execution of a task the STOP command is used. The format of this command is :

```
STOP task-name      (NEW LINE)
```

To restart a task you must reschedule it, or if it has already been scheduled you can use the START command.

Tasks may also be forgotten just as any other FORTH word. This is possible because every task is a word in the dictionary. This means that FORGET task-name (NEW LINE) could be used to stop the task as well, however, the task could not subsequently be rescheduled because the task definition would no longer exist in the dictionary.

ZX81-FORTH also allows a task to be run without scheduling it. This would be useful in debugging a task to ensure that it is running properly. The command is :

```
RUN task-name  (NEW LINE)
```

Before we deal with the word START, lets ask a question. What would happen if your task was extremely long ? That is, say, the task took longer than one second to execute and yet was scheduled to execute one per second. In a case like this the task would be rescheduled before it was completed, and the system

would eventually lock-up. A single task can be "back-scheduled" 63 times before the system would lock-up.

Now back to START. START clears the task register of "back-scheduled" tasks and will unlock a lock-up task. This has to be done for each task at a time and the command format is :

START task-name (NEW LINE)

LOCK Prevents all tasks from running. Tasks are still being scheduled to execute during a LOCK condition, but whatever program is being executed when LOCK is executed will gain second to highest priority in the system (second only to the master 1/50th second task) and will not be interrupted by any other task.

UNLOCK Allows tasks which have been LOCKed out to begin executing.

LOCK should be run only for a very short period of time. This word locks all lower priority tasks from running. If the LOCK to UNLOCK time was longer than the time the lower priority task was scheduled to execute in, then the lower priority task would be queued-up. LOCK does not stop the scheduling of tasks to be run, it only stops their actual execution. Therefore, the possibility exists for a task to completely fill it's queue buffer (63 scheduled executions), and upon UNLOCKing the tasks, the task with the overflowed queue would be blocked from running and could only be released by a START.

TOFF This word resets a system flag so that upon the execution of a WARM restart the following items occur :

- Causes a LOCK all tasks
- Sets the background task to a null task
- Forces the display to SLOW mode

TOFF is the default state upon power-up.

TON Disables TOFF and allows scheduled tasks to execute after a WARM restart.

It is also possible to link in a short program to run continuously in the background. A program linked in such a way will execute any time that there is nothing else going on in the system and in effect has the lowest priority of any program in the system. Programs which are put into the background must not output any information to the console or request any input from the keyboard buffer. If a background task does, there is a high likelihood that the system will not work properly. Background routines can schedule higher priority tasks to run and can access any of the system variables just as other routines in the system can, but the background routine must execute quickly, in the

order of 1/10th second or less, or the system's overall performance will deteriorate. Remember, though, that if the system slows down by 50% it will still be many, many times faster than ZX81 BASIC !!

BACK Is used to link in a user routine into the background. BACK program-name (NEW LINE) will make 'program-name' part of the background activity.

NUL Is a program which does nothing. It is used to swap out non-empty tasks from the background. BACK NUL will put the default null task into the background.

Try this :

```
: A ." THIS IS A TEST " CR ;
TASK B A
: C  RUN B ;
BACK C
```

This program will take A, a program which prints out the line "THIS IS A TEST" and attach a task "B" to it. The program "C", when executed, schedules B to execute immediately. C is then put into the background so that any time the system is doing nothing it schedules it to do something, namely execute A. To stop the message from printing continuously to the screen just hit SHIFT/SPACE momentarily. This will execute a WARM reset which automatically resets the background task to NUL.

One of the most useful tasks to run is a set of routines to print out the stack contents on the bottom line of the screen. The entire program should be typed on the editor screen. Then switch to the console screen (SHIFT/EDIT) and type CPL to compile the entire editor screen.

```
HEX CLS 16 C0 5 + C!
0 17 1F 17 SCREEN ST ST REV
: SCL 0 C N>C ST .W ;
: STD # 5 W> ST .W ;
: ST4 4 0 DO 4 I - PICK STD LOOP ;
: STE SCL ST4 FA7E SP@ - 2/
"  SP = " ST .W STD ;
TASK STK STE
EVERY 1 TS STK    DECIMAL
```

What is happening ? The first line clears the console screen. The second line creates the reverse video display line at the bottom of the screen. The screen name is ST (for stack screen). The SCL word is a screen clear command. STD is a screen display word. Finally, ST4 is the word which displays the top 4 stack values. STE stands for stack execute, this being the execution part of the code. The routine clears the screen (SCL), then displays the top 4 items (ST4), then gets the stack pointer value and displays it. Lastly, the final statement is a task which schedules the word STE as a task called STK, once per second.

The most useful applications of multi-tasking are in conjuction with various types of I/O (Input/Output), where the power of the computer can be used to control things and events in the outside world. In fact you could say that the whole future of computers as controllers is in that sort of role.

Any demonstration of tasking without employing I/O can tend to be trivial, and an impressive demonstration of tasking can be carried out using the console screen as follows :

If you set bit 8 of the screen byte it will reverse the video of that particular part of the screen. The word BYTE will do this and it also incorporates an offset so that an index can be applied.

```
: BYTE  DUP FBUF + @ 128 XOR SWAP FBUF + ! ;

: CURSOR1  0 BYTE ;       : CURSOR2  2 BYTE ;

: CURSOR3  4 BYTE ;       : CURSOR4  6 BYTE ;

: CURSOR5  8 BYTE ;       : CURSOR6 10 BYTE ;

TASK  TASK1 CURSOR1       EVERY  25 TT  TASK1

TASK  TASK2 CURSOR2       EVERY  25 TT  TASK2

TASK  TASK3 CURSOR3       EVERY  25 TT  TASK3

TASK  TASK4 CURSOR4       EVERY  25 TT  TASK4

TASK  TASK5 CURSOR5       EVERY  25 TT  TASK5

TASK  TASK6 CURSOR6       EVERY  25 TT  TASK6
```

This produces a very graphic illustration of the multi-tasking capabilities of ZX81-FORTH effectively giving a number of flashing cursors on the top line of the console screen.

17.0 The CODE compiler

This section describes how machine code can be compiled. The advantages of writing in machine code are an increased execution speed and more compact programs in terms of memory space used.

Machine code is a term referring to the type of numbers the Z80 microprocessor inside the ZX81 will recognise. In reality, it will only see combinations of 0's and 1's organised into data and instruction codes. It would be easier to deal with these numbers in terms of HEXADECIMAL, which is method by which each group of 4 bits is assigned a character between 0 to 9 and A to F. Now, it would be very, very difficult to remember what every HEX number did inside of the Z80, so every operation is represented symbolically by a nmemonic. In is common practice to write source code in these nmemonics and then convert the nmemonic to the proper HEX number either by hand or with the aid of an assembler.

First, the CODE compiler will be described, and then an example of its use will be given. The CODE compiler has the following format :

```
CODE ... hex code ... ;C
```

The above example shows the CODE compiler outside of a definition. It could also be used inside a colon definition. The CODE compiler places code at the current head pointer in the dictionary. Inside a definition you can have as many words as you want before CODE and after the ;C. Here is an example :

A machine code routine can be created to add the three numbers found on the parameter stack and to put the result back onto the stack. We will use the HL and DE registers for this, but first a little background information.

ZX81-FORTH supports commands for putting numbers onto the parameter stack and removing numbers from the parameter stack. To remove a number from the stack all you have to do is execute a Restart 2 instruction (D7 hex). This instruction takes the number off the parameter stack and places it into the HL register pair. From there on you can use it in a machine level CODE definition. To take a number from the HL register and place it on the parameter stack you must execute a Restart 1 instruction (CF hex). Now let's write the routine :-

```
E5    PUSH HL       ; This saves the contents of HL and DE
D5    PUSH DE       ; by placing them on the processor stack
D7    UPOP          ; Takes the top stack item and puts it
                    ; into HL
EB    EX DE,HL      ; swaps the contents of HL and DE
D7    UPOP          ; Puts the 2nd item into HL
19    ADD HL,DE     ; Adds HL & DE and puts the result in HL
EB    EX DE,HL      ; Move answer of first add
```

```
D7    UPOP            ; Gets third number from stack
19    ADD HL,DE       ; Adds the 3rd number with the sum of
                      ; first two
CF    UPUSH           ; Puts result back onto the stack
D1    POP DE
E1    POP HL          ; Restore registers
```

No return instruction is required as this is automatically inserted by the outer-interpreter or the ";" at the end of a colon definition.

In order to enter this code in the dictionary under the word "3+" the correct programming format would be :

```
: 3+  CODE E5 D5 D7 EB D7 19 EB D7 19 CF D1 E1 ;C ;
```

To execute 3+ place 3 numbers on the stack and use the new word.

```
1 2 3 3+ .      (NEW LINE)
6 OK
```

3+ adds the three numbers and replaces the sum, 6, onto the stack. "." prints the top stack item.

18.0 Applications

A definition of the FIG word SP! to reset the stack would be :

```
: SP!  CODE  FD 2A 90 FC  ;C ;
```

A very useful routine follows and it this does the same thing as the READ A$ statement in BASIC.

```
: READ ." INPUT REQUEST " S@ ;
```

This gets a string from the keyboard and places it on the character stack. To display the result, type :-

```
CO .W      (NEW LINE)
```

Here is a program to convert degrees Fahrenheit to degrees Centigrade. First, let us read in a variable :

```
: READ ." ENTER DEGREES FAHRENHEIT " CR ." ? " S@ ;
```

Next a word to convert the string to a number:

```
: INPUT READ ># DROP ;
```

># converts the top of the character stack to a number. We must drop a number because the conversion leaves a flag. Next is the actual conversion routine.

```
: CEL 32 - 100 * 9 / 5 * 100 / ;
```

Note that the value must be scaled by 100 before performing the division. A later division by 100 is needed to bring the result back to its original scale.

```
: PRNT ." DEGREES CENTIGRADE " . ;

: CELS BEGIN INPUT CEL PRNT CR CR AGAIN ;
```

To run the whole program type "CELS" and respond to the prompts.

19.1 Any problems ?

We are very anxious to ensure that you are satisfied with your FORTH software, so we hope you will feel free to contact us should you have any problems or queries.

We would prefer you to ring us on Bournemouth (0202) 302385 between the hours of 5pm and 6pm, Monday to Saturday so as to allow our work to continue un-interrupted.

19.2 Acknowledgements

The FORTH language was originally publicised by the

FORTH Interest Group
P.O. Box 1105
San Carlos
California
CA. 94070 USA

FIG UK can be found at

C/O Honorary Secretary
15 St Albans Mansion
Kensington Court Place
LONDON W8 5QH

ZX81-FORTH is based on TREE-FORTH by Bob Alsum of Tree Systems Inc of the USA.

19.3 Copies

ZX81-FORTH is the copyright property of David Husband trading as Skywave Software and all rights are reserved. ZX81-FORTH is supplied on an "as is" basis, with no warranty, specific or implied, attaching. No liability will be accepted for consequential loss or error. Any faulty media will be replaced free of charge.

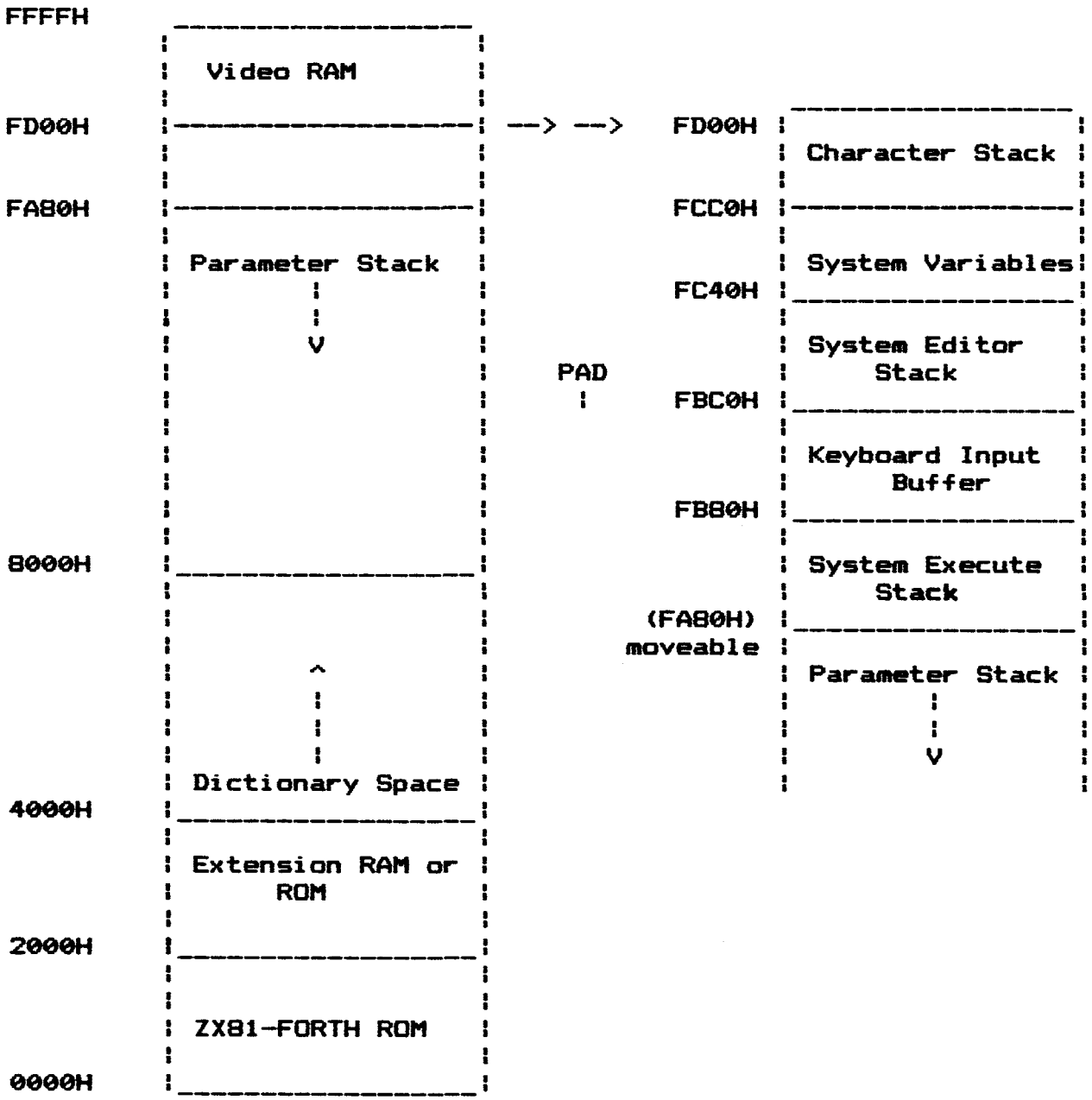
This does not however affect any consumer rights under existing legislation.

Copyright of all software remains with the original authors. "Skywave Software", "Skywave", and the Skywave logo are Registered Trademarks.

We would ask you, therefore, not to make, or permit to be made, copies or give copies to any third party (your friends, etc.) or to sell copies.

We hope you will agree with us when we way that it is only by Software Vendors, such as ourselves, making a reasonable return on our efforts, that the quality of the software marketed will improve and prosper. You must realise that it if piracy is rife the best software will never be put onto the market and prices will remain high. We ask your co-operation in ensuring that this product is not abused in this way.

20.0 Memory Map



Quick Keyboard Reference

Key	Shifted Key	Shifted Key Function
1	EDIT	Toggles between screens
2	AND	Fetches a line from PAD
3	THEN	Puts a line into PAD
4	TO	Deletes a line from the editor screen
5	<-	Moves cursor left
6	↓	Moves cursor down
7	↑	Moves cursor up
8	->	Moves cursor right
9	GRAPHICS	Inserts a line on the editor screen
0	RUBOUT	Deletes one character
Q	""	Compiles an editor screen line
W	OR	Store word. Displayed as !
E	STEP	Fetch word. Displayed as @
R	<=	[character
T	<>	_ character
Y	>=	] character
U	$	$ character
I	(	(character
O	)	) character
P	"	" character
A	STOP	Clears the present screen
S	LPRINT	% character
D	SLOW	' character
F	FAST	\ character
G	LLIST	^ character
H	**	# character
J	-	- character
K	+	+ character
L	=	= character
NEW LINE	FUNCTION	Home cursor to top left corner.
Z	:	: character
X	;	; character
C	?	? character
V	/	/ character
B	*	* character
N	<	< character
M	>	> character
.	,	, character
SPACE	BREAK	WARM Restart, if held for 1/2 second COLD restart.