XFORTH

by A.I.M. RESEARCH

Table of Contents

Chapter 1 Preliminaries. 1-1

Chapter 2 First Steps. 2-1

Chapter 3 Manipulating the stack. 3-1

Chapter 4 Defining your own words. 4-1

Chapter 5 Loading and listing definitions and commands from disc. 5-1

Chapter 6 Control Structures. 6-1

Chapter 7 Constants and variables. 7-1

Chapter 8 Text strings and characters. 8-1

Chapter 9 Virtual memory. 9-1

Chapter 10 Interfacing with the operating system. 10-1

Chapter 11 Other topics. 11-1

Appendix A The screen editor. A-1

- A.1 Starting up the editor. A-2
- A.2 Using the editor. A-2
- A.3 Changing the key bindings. A-7
- A.4 Editor-related functions. A-7

Appendix B Using the FIG Editor. B-1

- B.1 Loading the editor B-1
- B.2 Selecting a block and input of text B-2
- B.3 Line editing B-3
- B.4 Cursor control and string editing B-4
- B.5 Block editing commands B-6

Appendix C The demonstration package and the basic examples. C-1

- C.1 Jack C-1
- C.2 Fractions C-2

C.3 Random numbers. C-2
C.4 The Sieve of Eratosthenes. C-2
C.5 The eight queens problem. C-3
C.6 Quicksort. C-4
C.7 SEARCH C-5
C.8 crypt C-5
C.9 The 'to solution'. C-5
C.10 Easter C-6
C.11 Hamurabi C-7
C.12 Exponent C-7
C.13 Life C-7
C.14 DisForth C-8
C.15 New-loop C-8
C.16 Modules C-9

Appendix D The Filing System. D-1

D.1 Random access files. D-2
D.2 Sequential input/output. D-5
D.3 Some other useful words. D-7

Appendix E The debug package. E-1

E.1 Protection. E-2
E.2 Tracing. E-3

Appendix F The assembler. F-1

Appendix G Adapting your system G-1

G.1 General G-1
G.2 Basic terminal handling. G-2
G.3 Cursor addressing G-3
G.4 The screen editor. G-4
G.5 Removing the screen editor G-4
G.6 Prompts and showing the stack G-4

Appendix H Bugs. H-1

How to use this manual.

Although this manual cannot teach you all about Forth, it will help you get started. The best way is to read Chapters 1 and 2 first, then work through Chapter 2 at the console. After that you'll be ready to read the rest of the Chapters and as much of the technical material in the Appendices as you need. Even by the time you reach Chapter 8 you'll be able to do far more than most Basic systems would allow.

The FORTH-79 and xForth reference lists summarize the meanings of all the xForth words described in the text and more besides. It may be worth sticking them on the wall near your terminal.

Studying the examples on your disc in conjunction with the Appendix describing them will probably help you get a feel for Forth programming, as long as you've worked through the basic material first. The screen editor source code in the file SEE.BLK also shouldn't be too hard to follow and is a good example of a non-trivial program coded easily in xForth. Note that the editor has been written to be simple to modify rather than with 'efficiency' in mind: this editor spends most of its time waiting for the user anyway. You should find it reasonably easy to produce your own customized version (but remember the xForth licence covers the design and original coding so you may not sell or give away the modified version). The system source code on blocks 30 to 42 is harder to follow, mainly because it uses xForth words that are only defined in the xForth Technical Manual, but you can still get some feel for what's going on.

To learn more about Forth, your best bet is to join the Forth Interest Group (FIG).

Its address is

Forth Interest Group,
c/o 30, Worsley Road,
Frimley,
Camberley,
Surrey GU16 5AU,
England

for U.K. users and

Forth Interest Group,
P.O. Box 1105,
San Carlos,

ROGER FIRTH
MEMBERSHIP SEC. FIG(UK)
7 WYNDHAM CRESCENT
WOODLEY
READING RG5 3AY

£7 per annum.

CA 94070,
USA

for others.

The best teaching book is usually considered to be 'Starting Forth' by Leo Brodie, published by Prentice-Hall in 1981. It is likely that the BBC book will also be good; even though it is aimed at Forth for the BBC microcomputer, most of it will be applicable to xForth because xForth obeys the Forth79 standard.

Watch for magazine articles; most of the computer magazines have had articles on Forth; for example, Computing Today had a whole series in 1981-82 and Byte had a special issue on Forth in August 1980.

Chapter 1

Preliminaries.

Forth is a computing language that at first sight is very different from most others. It's best if you temporarily forget what you already know about computing and read this introduction with an open mind.

To use xForth, you type 'words' at a terminal. The words needn't be English words: a word is in fact any sequence of non- blank ASCII characters, up to 31 characters long, so

++ + dog Dog DOG +dog dog+ dogs 1 . , !

are all (different) possible xForth words. The xForth system reads the first word - that is, after you type a carriage return to tell it you're ready, it takes the first sequence of non-blank characters. It looks up the word in a dictionary that is built into the system. If it finds the word, it performs an action that has previously been defined for that word. Then it goes on to the next word, and so on until there's nothing left to read, when it waits for you to type in more words. This process of looking up words and obeying their definitions is called interpreting, and the part of xForth that does the job is called the interpreter.

It may be helpful to think of xForth words as like English verbs, which also describe actions. Some verbs in English don't need a noun after them ('yawn') while others do need a noun after them ('do something'). Similarly, some xForth words can act on their own, as in BELL which sounds the terminal's noise-maker, while others operate on words that follow them and that the xForth interpreter hasn't seen yet. An example is VARIABLE which takes the next word and adds it to the dictionary with an appropriate definition so that VARIABLE x acts in much the same way as you'd expect it to in any language.

Writing programs in Forth consists of defining sequences of actions in terms of existing words, and assigning the sequences to new words which are added to the dictionary. You'll learn how to do this in Section 4, but the idea is to end up being able to type, say,

3 frames

to do three frames of your animation program, or

Saturn-probe

to run your video game. The process of word building is helped by the controlling words already in the dictionary that let you handle conditionals, loops and so on.

Of course, the different words will have to communicate somehow and in Forth most messages are sent simply by being left on a stack for the next word to pick up. The stack is just like a tidy pile of papers: most words pick messages off the top of the pile and leave others there, rather than making an untidy mess by scrabbling around lower down the pile. To help beginners (and experts who admit they sometimes make mistakes) xForth is initially set up to show what's on the stack every time control is returned to the terminal. A side effect of having a stack is that arithmetic can be done using 'reverse Polish' notation very easily, so Forth systems aren't usually supplied with a direct means of understanding the usual algebraic type of notation. One could be supplied, but most programmers find the rewards of programming directly with the stack - once they've got used to it - are so great that they never get round to writing an algebraic expression parser.

Even if you haven't followed everything that's been said so far, go on and read the next section and try out the examples. You'll find it's really easy.

Chapter 2

First Steps.

You are now ready to start using xForth, though it's worth skimming through this chapter before actually sitting down at a terminal with it.

You'll certainly make the odd typing mistake when using xForth, so make some now and correct them: unless you chose some other character during the CONFIG process, xForth responds to backspace (control/H if you haven't a backspace key) by backing up and removing the last character typed. It responds to control/X (or whatever you chose for yourself) by removing the whole line typed so far. If control/P is typed, a copy of everything that appears at the terminal will go to the printer until such time as you type another control/P. If you have a teletype or a VDU that can't backspace, you should have let the CONFIG program know. If you didn't, you should now type

```
LOAD-FILE TTY-RUB.BLK
```

followed by a carriage return, and then backspace and control/X will behave in a more suitable way for your terminal.

Type

```
." Hello "
```

followed by a carriage return. Make sure there's a space after the first quote, and make sure the dot is there. xForth will reply (on the same line)

```
Hello ok
```

and then type on a new line

```
Stack empty
```

What has happened is this. The interpreter read the word ." (dot-quote) from the input. As explained earlier, anything not

containing a blank is a word as far as Forth is concerned.[1] This particular word is defined to read all text up to the next double-quote character " and type out that text. Having done this, the interpreter finds there is nothing left to read so it returns control to you, typing ok to let you know there were no errors, and then prompting you for more input. The prompt is initially set to show the present state of the stack. You'll find later that the prompt and the ok print can be changed: for example, if your terminal has a separate status line you can put the stack picture there, out of the way. Now try

```
" Hello " TYPE
```

You get the same as before. The word " has read and stored the text appearing the input up to the next " then has left a message on the stack saying where it stored the text and how much text there is, which is exactly what TYPE needs to type out stored text.

Type 1 2 3 4 and get

```
ok
Stack base 1 2 3 4
```

When xForth's interpreter sees a word it can't recognize, it tries to decode it as a number and if it succeeds, it leaves that number on the stack. If it fails, it issues an error message (a question mark) indicating that it can't find the word in its dictionary. Try typing AAA and see this in action.

Type 1 2 3 4 + and the stack will contain 1 2 7. The word + takes the top two numbers off the stack and puts back their sum. Type . and the 7 will be typed out, leaving 1 2 on the stack, since the word . removes and prints the number on the stack top. Type - and get -1 on the stack. Type . to print it and the stack will be empty. Type . again and see what happens.

The words + - * / MOD = < > <= >= <> AND OR XOR all take two items from the stack and put back the result of the appropriate operation. For instance, * is integer multiplication and >= is 'greater-than or equal to'. Note that / is integer division and MOD is remainder, so 1 2 / gives 0

---

1. Sometimes, we will distinguish an xForth word from normal text or punctuation by putting it between a reverse and a forward single quote. For example, we could have written `."' for dot-quote. As a reverse quote is never used in xForth, this should always make it clear what's meant.

and 1 2 MOD gives 1 . You can get both at once with /MOD which leaves the result of division on top and the remainder underneath. The word = means 'equal' and leaves TRUE (i.e. 1) on the stack if the top two items were equal, and FALSE (i.e. 0) if they weren't. The word <> means 'not equal'. The logical operators work bitwise so 1 2 OR is 3 . You can see this better by working in base 2 arithmetic: type

```
BINARY 1 10 OR
```

and see 11 on the stack. Now type DECIMAL and the 11 will be typed as 3 . This facility to change base is very handy. xForth has the words BINARY DECIMAL and HEX which arrange for all further numerical input and output to be in the appropriate base. Actually, all bases are possible within the limitations of the ASCII character set to represent them. You'll learn later how to get an arbitrary base.

Internally, numbers are 16 bit signed binary integers and so they lie in the (decimal) range -32768 to 32767. Double precision (32 bit) integers and unsigned 16 bit integers are also available: see the 'Handy Reference'. If you ordered the floating point package, you will also have floating point numbers with 10 digits of precision, and all the appropriate operators and functions to go with them.

The words NEGATE and ABS take one item from the stack top and replace it by its negative or its absolute value respectively. The word 0= leaves a logical true (1 in fact) if the stack top was 0 and a logical false (0) otherwise. NOT reverses the logical state of the stack top (and you can see it's equivalent to 0=). The word 0< leaves true if the stack top was negative so gives the same result as typing the two words 0 < . Try it for yourself. Try 0> too.

Now you know how to do simple arithmetic and how to print messages. Try this out for a while. You'll discover that the stack is cleared after errors, but if you want to clear the stack explicitly, use the word SP! which throws away everything.

Chapter 3

Manipulating the stack.

It doesn't take long to get used to stack arithmetic. Even if it seems a pain at first, you'll discover that the versatility of Forth's simple message passing system makes more advanced work easy, so bear with it. You can always write an algebraic notation converter later. Notice that brackets are never necessary. The algebraic notation (1+2)*(3-4) becomes

```
1 2 +  3 4 -  *
```

in Forth. This is much as you'd think of it: take 1 and 2 and add them, noting the result (on the stack), then take 3 and 4 and subtract them, noting the result, then take the noted results and multiply them. Actually, brackets are sufficiently redundant that they are used to enclose comments: the word (gobbles up everything in the input until a) is found. As a result, you can write

```
1 2 + ( Add 1 and 2 ) 3 4 - ( Subtract 4 from 3 )
```

Try it. What happens if you leave out the final closing bracket? By the way, (is a Forth word and so must be surrounded by spaces. The closing) is just something the word (is looking for so it doesn't need a space before it, though most people put one there.

Often you want to make a copy of what's on top of the stack. The word DUP does this, replacing the stack top with two copies of what was there. So 99 DUP leaves 99 99 and 99 DUP + is one way to get 198. The word SWAP reverses the order of the top two items so 99 100 SWAP - leaves 1 rather than the -1 that 99 100 - would leave. The word OVER copies the second top item so 99 100 OVER leaves 99 100 99 . With these words you can do most of the manipulations you need. Other useful ones are

ROT — (rotate) which pulls out the third item down and puts it on top: 1 2 3 ROT leaves 2 3 1.

ROLL — is a generalized rotate, defined so that typing 3 ROLL gives the same effect as ROT. So

99 100 101 3 ROLL leaves 100 101 99 . Verify for yourself that 2 ROLL is like SWAP and 1 ROLL does nothing. What happens with 0 ROLL ?

PICK is like ROLL except that it copies the appropriate item rather than pulling it out. So 99 100 101 3 PICK leaves 99 100 101 99. What are 2 PICK and 1 PICK equivalent to? What about 0 PICK ?

Actually, DUP SWAP OVER and ROT are enough for nearly all work with the stack, and since they are efficient they tend to be favoured. If you are having to fiddle with lots of stack items you probably should be breaking your definitions up into smaller pieces: more on that later.

Some arithmetic operations are so common that it's worth taking advantage of inbuilt machine facilities to do them faster, even though the same effect can be achieved otherwise. Thus 1+ is defined to have the same effect as 1 + (i.e. it increments the stack top), and 1- 2+ 2- 2* all have the obvious effect.

Other useful words will be found in the reference lists. For example, .R prints an integer right-aligned in a field, so 101 4 .R prints 101 preceded by a blank, but without the usual following blank that the free-format output word . gives The idea is that with .R you always know how many character positions a number will occupy (as long as it isn't too big to fit in the space you asked for). SPACE prints a space and 6 SPACES prints 6 spaces. CR moves to the start of a new line while 3 CRS moves down 3 lines. 44 TAB moves the cursor or print head of the terminal to the 44th position in the line, if it can do so without backing up.

Chapter 4

Defining your own words.

Now we can really get going. As you saw earlier, programming in Forth is about defining new words. This is done mostly by the word : (colon) which takes the next word in the input and prepares a dictionary entry for it. The definition to be entered in the dictionary is specified by all following words until a semicolon is encountered. So

```
: 4* 2* 2* ;
```

defines a new word that multiplies the stack top by 4. In future, when 4* is read by the interpreter, the sequence of actions 2* 2* will be carried out. However these words won't be looked up again so the interpreter doesn't waste time or space: this is part of the secret of Forth's small size and high speed.

If you have programmed in Basic or in APL, you will need to know that when you make a definition like this, Forth does not keep the exact text you type in. When xForth or any other Forth system puts a word in its dictionary, it keeps the definition in special internal form. This means you can't edit the definition like you can in, say, Basic. If you think this is a limitation, you haven't yet learned how fast and powerful xForth is! To make definitions that you can edit - the normal way of working - you put them in disc blocks as explained in the next chapter.

Since Forth works differently from other languages, it is a good idea to learn a bit about how the interpreter and the colon operate in definitions. When : executes, it enters the word following (for example, 4* in the above example) into the dictionary and then tells the interpreter to store (compile) the actions described by words that follow instead of executing them as it reads them. The interpreter does so until it finds a semicolon, after which it reverts to its normal behaviour. Comments still work correctly. Try out

```
: Blast-off 1 2 3 4 5 . . . . . ." We have lift off."
( And notice you can have a colon definition stretching )
( over more than one line, but comments should always )
```

```
( be closed at the end of a line.)
2 CRS  47 EMIT  92 EMIT  15 CRS ;
```

You'll discover that xForth reminds you you're in the middle of a colon definition by putting a prompt &: at the start of each new line until the closing ; is typed. Incidentally, the word EMIT outputs one character whose ASCII code is on the stack. Later you'll see how to avoid having to look up the ASCII codes for characters to be emitted. Now type Blast-off to see what you just defined.

We could define a word 2DUP that would make a copy of the top two stack items.

```
: 2DUP OVER OVER ;
```

If you try this xForth will tell you a definition already exists for that word, since 2DUP is built into the system with, as it happens, an equivalent definition to the present one. This is not treated as an error: your definition is added to the dictionary and will be the one used in future definitions, since the most recent definition is always found when the interpreter looks in the dictionary. Now we could write a word that prints the sum and difference of the top two stack numbers.

```
: sum&diff   2DUP   + .   - .   ;
```

Notice that definitions are more readable if you put logically connected items close together, separated by spaces or new lines from others. In xForth, unlike Basic, neat formatting and comments do not cost you memory space, so you may as well make life easy for yourself!

As you might expect,

```
: NOOP ;
```

defines a word that has no effect - anything. Definitions can be as short as this or as long as you like, but on the whole they should be at most a few lines long. Complicated problems should be split into simpler parts to achieve this; the solution will then be much easier to write and test. Generally, you should design 'from the top down', by looking at your problem and deciding how to split it up, then deciding how to split up the separate bits themselves, and so on until you have words that can be defined in terms of words that are already in the dictionary. Then you type in and test your low-level definitions first, followed by the next level up, and so on until you're back up at the top level with a debugged and working program. This technique not only makes sure you never need to wonder whether what you want is 8 levels deep or 9

levels deep on the stack, it also makes it much easier to write correct programs in the first place and to make the changes you invariably want to make later.

Note that when reading programs you should realize that they'll have been designed in this way, so start at the end where the high level words are and work your way back to the low level ones.

The remainder of this Chapter may be skipped on first reading.

To help you when you come back 6 months later to a definition, you should add lots of comments as we noted above. A useful convention is to add, right after the word being defined, a comment that describes the action on the stack, as follows:

```
: 2DUP ( n1 n2 --- n1 n2  n1 n2 )  OVER OVER  ;
```

The information before the --- is what's on the stack before and the information after is what's there after. If you use descriptive names like addr or #bytes you'll find it much easier to keep track of what your words are supposed to be doing, and xForth's stack display will be more useful to you. Another piece of notation used in some xForth descriptions (mainly in the technical manual) is

```
: " ( +++ addr length ) ..... ;
```

where the +++ says something is read from the input stream and the other items in the comment say the word takes nothing off the stack but leaves two items, an address and a length with the length on top.

A useful trick is to start every new set of definitions with a dummy word that has a useful mnemonic name. For example, if you are writing a word processing program, you can make the first definition : Words ; and this acts as a marker in the dictionary for where you started. When you've made lots of errors and want to clean it all out and start again (or if you need the memory space for something else), type EMPTY and all your definitions will be removed. If you only want to remove some of them - say, all the definitions since Words, and including Words itself - type

```
FORGET Words
```

and everything you defined since Words will be removed.

Incidentally, to see what's in the dictionary, type VLIST (which means vocabulary-list) and you will get a long list of

words most of which you've never heard of.[2] Don't be tempted to try them out at random, since some are system words that interface to CP/M, for example, and they could have disastrous effects if misused. If you want to get deeply involved with the system programming aspect of xForth, get the xForth Technical Manual or even the Meta system. Returning to VLIST, the reason for the name of the command is that the dictionary can be split into separate vocabularies, each dealing with a specific topic. We don't say much about this in the present manual, but if you read the Appendix describing the assembler you'll learn a little about the subject since the assembler words are kept in a separate vocabulary so they don't get in the way during normal use.

If you type COLD at any time, the xForth system will reset itself to a pre-defined 'protected' state. The resetting will include removing all unprotected definitions, emptying all buffers and stacks, and resetting the filing system and the execution variables. (Some of these terms are not defined until later in the manual.) The protected state is usually the state xForth is in when you enter it from CP/M, but you can type PROTECT at any time and the present state will become the protected state and so will be restored by COLD . This is mainly of use when you have a set of working words you don't want to lose, and you're debugging some new set. You will learn later about the need for PROTECT if you alter an execution variable.

---

2. You can stop the listing temporarily by hitting control/S, and restart it by hitting any key except control/C. (To abort the listing, type control/C either while the listing is running or while it's been paused with control/S.) This general technique of pausing or breaking output applies to all words that use CR, and so to all system words that send output to the terminal.

Chapter 5

Loading and listing definitions and commands from disc.

Until now you have been typing everything at the terminal. You can also put definitions and commands onto disc where they will be kept for future use and can be altered whenever necessary. However, there's a little more to learn about Forth first.

Normally, Forth doesn't work directly in terms of disc files but uses an idea called virtual memory instead. If you imagine a huge memory space, with each place in it needing two numbers to identify it (a block number and an address within the block) then you have the idea. Of course, present microcomputer memories don't contain the many megabytes that this implies, so not all of the huge memory space can be present in the true memory at once. To get round this, xForth automatically transfers blocks (of size 1K, i.e. 1024 bytes) to and from disc as required. Even discs don't usually have enough storage, but xForth lets you assign any virtual memory block until the disc is full, then gives you an error message. You could have, for example, blocks 1 to 200 and 16321 to 16520 on a 400K disc. So when you're writing programs that manipulate files, you just pretend you have a huge memory and read and write within it nearly as easily as you do in true memory.

What has all this to do with keeping definitions on disc? Well, xForth can interpret from virtual memory instead of from the input buffer, and in practice this means reading from disc. Suppose for the moment you have somehow got a definition into virtual memory and you want to compile it. The word LOAD takes the number on the stack as the identifier for a 1K block of virtual memory, reads it into true memory if necessary, and then interprets its contents as if you'd just typed them in. For example,

```
29 LOAD
```

would load all the definitions on block 29.

When the end of the block is reached, the interpreter returns to where it left off when it obeyed LOAD. This could be

to the terminal, to another block that contained the LOAD (yes, LOADs can be nested, though it's bad style to nest them deeply) or even to a word that contained LOAD. A simple example is SYSGEN in the kernel system, which is defined as

```
: SYSGEN 1 LOAD ;
```

Now you could hunt around for an empty block (by listing everything you could think of, using, say, 54 LIST). Then you could put your definitions in it somehow and finally type 54 LOAD. In fact, this is what most Forth systems make you do. But even though the system of having numbered blocks of virtual memory is great for use within programs that manipulate files, when you want to write an application it's much nicer to have named files since these are easier to remember than a string of numbers. This is where xForth scores heavily over most other versions of Forth, since it gives you the best of both worlds. You will learn later how you can associate named files with different regions of virtual memory. For just now, let's see how easy it is to create a file and then load it.

First, type

```
SEE-FILE myfile.blk
```

The word SEE-FILE is an xForth word that takes the word following it (here, myfile.blk) and tries to convert it to a file name acceptable to the operating system. For CP/M, this would be an acceptable name if it were in upper case, so SEE-FILE treats it as if you had typed

```
SEE-FILE MYFILE.BLK
```

and then proceeds to try to find a file of that name. If none exists, a new file is created. Then the screen editor is called up at the start of the first block of the file. If this is a new file, the editor will tell you that you have a new block as soon as it starts up. You can now enter your definitions (see the relevant Appendix for detailed instructions on how to use the editor) and then return to the xForth interpreter level by typing control/Z.

Now you can load your definitions by typing

```
LOAD-FILE myfile.blk
```

which makes the interpreter go to the first block of that file and start interpreting from there instead from the keyboard input buffer. (Actually, what LOAD-FILE does is to make the file 'myfile.blk' possess blocks 8001 to 8999 and then do 8001 LOAD. The Appendix on the filing system describes this in detail.)

You may have wondered what the '.blk' extension is doing on the end of these file names. The answer is that this extension tells the screen editor and the loader that this file is a picture of blocks of virtual memory and so is suitable for loading Forth programs. If .blk is missing SEE-FILE and LOAD-FILE will refuse to cooperate, to give a measure of protection against mistakes.

From now on you can put your definitions and so on into disc blocks, using the editor. This means you can always see just what you defined, and also means that when you make a mistake you don't have to retype everything. In fact, to make things easy the interpreter leaves information on the stack if it finds an error during loading. You can then type WHERE and the editor will come into action with its cursor pointing just after the offending word.

To use the screen editor on a numbered block, just type (say) 45 SEE. Blocks 1 to 1000 are set up initially to belong to a file called FORTH.BLK on the currently selected default dics drive, and the system expects to find its error messages and so on there. So if you insist, you can treat xForth like any other Forth system and ignore the filing system interface altogether. You should, however, note that blocks 1 to 48 inclusive are regarded as reserved for system use, even if your system is delivered with nothing on these blocks. In particular, block 3 is used for temporary storage in most systems.

Notes:

1. The word --> makes the interpreter go to the start of the next block, even in the middle of a colon definition. It can be used to string blocks together so you don't have to type lots of LOADs.

2. To help catch errors, a colon definition isn't allowed to cross a block boundary except when --> is used.

3. To see what's on a block without editing it, use LIST. For example, 4 LIST will show you some of the error messages, and LIST-FILE see.blk will list the screen editor. LIST-FILE will try to list any file that doesn't have a .blk extension, by treating it as a normal ASCII file and listing it with line numbers at the console. You can get a printed copy of any file (.blk or not) by typing control/P to turn on the printer, then using LIST-FILE, then typing control/P again to turn the

printer off. As with all words that produce output, LIST and LIST-FILE can be paused or aborted by typing control/S or control/C.

4. A word TRIAD is useful in keeping permanent documentation if you'd rather work in terms of numbered blocks than named files. Try typing 30 TRIAD and then try a few other block numbers until you see what's happening. If you want to fine-tune the format to suit your printer, the source for TRIAD (and LIST and INDEX) is on block 37 in most systems.

5. A useful convention is to make the first line of every block a comment line. This is particularly useful with INDEX which prints the first lines of a range of blocks: 30 41 INDEX shows the headers for the basic system file.

6. The whole block is treated as one long line so the last column of each line shown by the editor is effectively adjacent to the first column of the next line. So take care not to run a terminating ; into the : starting the next definition.

7. Use the same trick as you learned earlier: type

```
: TASK ; LOAD-FILE foo.blk
```

or the like, so you can do FORGET TASK when you're finished, or to remove junk caused by errors. Better still, put

```
: a-descriptive-name ;
```

at the beginning of your load blocks and then you needn't remember to define a dummy before loading. You can type

```
FORGET a-descriptive-name
```

to clean up.

This might be a good time to try out the example 'Jack' described in the Appendix. It uses one or two features you haven't met yet but you should still get the general idea. If, like us, you always try things out before reading the instructions, you only need to know to type

```
LOAD-FILE jack.blk
```

to get started. You might well want to look at the instructions a few seconds later, though.

Chapter 6

Control Structures.

Within colon definitions, though not outside, the interpreter allows you to use certain control structures for repetitive or conditional execution. The main ones are IF ... ENDIF and BEGIN ... UNTIL but some others are also useful and will be described.

Often one wants a certain sequence of words to be performed only if a certain condition is true, indicated by the stack top being nonzero. This is achieved by enclosing them between IF and ENDIF which may be regarded as a special pair of opening and closing brackets. Thus

```
: say-if-greater  > IF ." Greater." ENDIF ;
```

types 'Greater.' in reply to 2 1 say-if-greater but nothing in reply to 1 2 say-if-greater . If you want something else to be done if the condition is false (i.e. if there's a 0 on top of the stack when IF executes) put the do-this-if-true part between IF and ELSE and the do-this-if-false part between ELSE and ENDIF .

```
: say-whether-greater > IF  ." Greater."
                        ELSE ." Not greater."
                        ENDIF ;
```

Actually, FORTH-79 uses THEN instead of ENDIF but we feel this is confusing to people who know other languages so xForth allows you the choice of either ENDIF or THEN . This manual always uses ENDIF .

Another common problem is to execute a series of words again and again until some condition is found to be true. To do this, enclose the words between the special brackets BEGIN and UNTIL , and make sure the word before UNTIL leaves the condition test result on the stack.

```
: annoy BEGIN BELL ?TERMINAL UNTIL ;
```

will sound the bell until any key is hit, since ?TERMINAL is a word that returns true (1) if a key has been struck, and false

(0) if not.

If you want to make a test at the beginning or in the middle of a loop, use BEGIN ... WHILE ... REPEAT , in which the words between BEGIN and WHILE are executed, then WHILE tests the stack and allows execution to continue if it's true but jumps to beyond REPEAT if it's false. If the condition is true, the words between WHILE and REPEAT are executed and then execution starts again from just after BEGIN .

A counting loop is obtained by using DO ... LOOP .

```
: count-to-9 10 1 DO I . LOOP ;
```

defines a word that prints the digits 1 through 9. What happens is that at execution time, DO takes the stack top as the initial value and the number below it as one more than the final value. Then everything between DO and LOOP is executed the requisite number of times, with the counting index I being set to the initial value the first time through, to that value plus 1 the next time through and so on. Notice that writing 10 1 DO rather than 1 10 DO is nice because the finishing condition is more often passed as a parameter on the stack than the starting condition. If we define

```
: count 1+  1 DO I . LOOP ;
```

then '10 count' prints out 1 through 10.

Like all conditionals and loop words, DO ... LOOP may be nested. The limit to DO ... LOOP nesting depth is machine-dependent but is never less than five and is usually far more. The index I always refers to the innermost loop (the one you're presently in) but you can get the one from the next enclosing loop by using J . Try out

```
: pairs 3 0 DO CR
                2 0 DO J . I . 10 SPACES LOOP
             LOOP
          CR ;
```

for yourself.

The structuring words you have met so far are the most often used. There are two more constructs to describe, but you may want to skip them for now and come back later.

If a loop index is to be incremented by some different amount than 1 each time, use DO ... +LOOP . The increment is placed on the stack just before +LOOP is reached and control passes back to just after DO if the index, after incrementing, is greater than or equal to the limit, assuming the increment

is positive. So DO ... 1 +LOOP is equivalent to DO ... LOOP . Owing to an unfortunate FORTH-79 standards committee decision, +LOOP behaves oddly if the increment is negative, so we advise you to avoid negative increments - the more so because a change in the standard is threatened on this point. The action for this case is defined in the FORTH-79 'Handy Reference'.

Quite a common problem is to do a lot of successive tests for equality of the stack top against some constants. The word KEY waits for a terminal key to be struck and returns its ASCII code on the stack, so in an editor one might want something like the following.

```
: obey-command
    KEY DUP 65 = IF do-the-A-thing
                ELSE DUP 66 = IF do-the-B-thing
                             ELSE  DUP 67 = IF ...
                                            ...
                                          ENDIF
                             ENDIF
                ENDIF ;
```

This works, but is ugly and error-prone. It's better to write

```
: obey-command
    KEY  CASE  65 OF do-the-A-thing ENDOF
               66 OF do-the-B-thing ENDOF
               67 OF do-the-C-thing ENDOF
                        ... DEFAULT
do-the-default-thing ENDCASE ;
```

which is xForth's way of writing successive tests. The stack top before CASE was entered is tested against the stack top before the first OF and if they're equal, the words up to ENDOF are executed and then control passes to beyond ENDOF . If they're unequal, the next stack top is tested and so on until either a single OF ... ENDOF part has been executed or the DEFAULT part has been reached. If the DEFAULT part is reached, it always executes, with the original test number (left by KEY here) still available on the stack.

Any number of words, including none, can appear between OF and ENDOF . Also, any number of words may appear between CASE or ENDOF and the next OF , as long as precisely one number is left on the stack for comparison with what was originally there before CASE was entered. If an OF part is obeyed, the original number (i.e. the result of KEY here) is gone, since the OF part knows what it must have been. If the DEFAULT part is obeyed, however, the number is still there in case you have some other action to perform. If there is no default part, the effect is as if you had written DEFAULT DROP.

Chapter 7

Constants and variables.

It is good practice not to have too many magic numbers in a program. For instance, the editor needs to use the number 63 quite a lot because the columns are numbered 0 through 63, but it would be a bad idea to have lots of 63's around since that would mean lots of changes if the number of columns were changed. Instead, at an early stage a constant is defined:

```
63 CONSTANT MAX-COL
```

which puts the word MAX-COL in the dictionary and makes its execution code put 63 on the stack. Note that CONSTANT is a defining word like : and constants go in the dictionary just like anything else, are forgotten by FORGET just like anything else, and have the same rules for naming as anything else - up to 31 characters excluding blank and null. Now whenever MAX-COL appears inside or outside a colon definition it will have the same effect as if 63 had appeared there. Of course, we could have produced the original 63 indirectly, since CONSTANT is only interested on what's on the stack, not how it got there:

```
B/BUF 16 / 1- CONSTANT MAX-COL
```

would have done the job since B/BUF is a predefined constant that returns the number of bytes in a buffer, viz 1024.

Although the stack makes it less necessary in Forth than in most languages to use lots of temporary storage locations, there are times when the only sensible way to go is to define a variable. This is done by the word VARIABLE which defines a new dictionary entry and gives it execution code that merely returns on the stack the address of a storage location. So if we've typed

```
VARIABLE x
```

then whenever x executes, a memory address appears on top of the stack. We're usually interested in seeing what's stored at that address or in putting something there. (The contents will be rubbish when the variable is first defined because FORTH-79

doesn't initialize variables.)

To put something in it, we use ! (pronounced 'store') which stores the number at the second top of the stack in the address at the top. So 2 x ! makes x point to a location containing the value 2. Make sure you get the order right, since x 2 ! is a perfectly good instruction that will put something in the address 2 and will probably cause grief later. The optional debug package protects you against these errors and others at the expense of execution speed: we recommend you use it until you're used to xForth, and even then use it except for production runs.

To see what's in a variable, use @ (pronounced 'fetch') which de-references the address on the stack, i.e. it replaces the address by whatever is stored at that address. So x @ now leaves 2 on the stack.

Some common operations on variables have special words. If x has been defined as above, 3 x +! adds 3 to the contents of x and leaves nothing behind on the stack. Adding and subtracting 1 are so often needed that 1+! and 1-! are supplied so you can type x 1+! to increment the value of x and x 1-! to decrement it.

Arrays work as follows. The word []VARIABLE (pronounced 'row variable') defines a vector such that 10 []VARIABLE y makes y refer to a vector of 11 elements, numbered 0 through 10. At execution time, 3 y returns the address of the element with index 3. The notation []VARIABLE helps you to remember where to write the index. Similarly 3 4 [,]VARIABLE z defines a matrix with 4 rows numbered 0 through 3, and 5 columns numbered 0 through 4. The defining word [,]VARIABLE is pronounced 'matrix variable'. At execution time, 1 2 z returns the address of the element with row index 1 and column index 2. Now you can type things like

```
6 y @  x !     1 2 z  1+!     0  7 y  !
```

and so on.

You can try the first 3 examples described in the Appendix now. Example 4 is also worth looking at, but uses ideas you haven't met yet. The final paragraphs of this section are a little more advanced and may be skipped the first time through.

Some variables already defined within the system are OUT BASE PRINTER-ON? WARNING >LINE and XOFF-CHAR. There are also many so-called execution variables. The rest of this section explains how these work.

The variable OUT holds the present position of the cursor or print head, and is adjusted by all the output operations. So you might define

```
80 CONSTANT cols
: ENSURE-LINE ( n --- ) ( Check room for up to n chars )
    OUT @ + cols > IF CR ENDIF ;
```

(This word already exists in the dictionary and does exactly this job.)

The variable >LINE is incremented by CR just as OUT is incremented by EMIT; it is not used by xForth but you may find it useful for keeping track of output.

The variable XOFF-CHAR normally contains the code for control/S, and is read by ?PAUSE. If XOFF-CHAR is altered to contain -1, the word ?PAUSE does nothing at all. Otherwise ?PAUSE looks for a key to be typed; if none is typed, it returns to the word it was called from while if a key has been struck, it checks whether it was control/C - in which case it aborts by calling 6 ERROR - or the contents of XOFF-CHAR, in which case it waits until another key is struck before returning to the calling word. (If control/C is struck, however, it aborts.) Of course, this is what CR uses to check whether to pause, but you can put ?PAUSE in any word you like. If you don't want any pausing or breaking, set XOFF-CHAR to -1. If you want breaking but not pausing, set it to -2 or some other value that isn't an ASCII code.

The variable BASE holds the current base for input and output of numbers, so 2 BASE ! is equivalent to BINARY . Since BASE @ . will always print 10 (why?) the word .BASE is supplied to show the present base in decimal. Try out the effect of

```
HEX      .BASE  BASE ?
BINARY   .BASE  BASE ?
DECIMAL  .BASE  BASE ?
```

(The word ? is exactly equivalent to @ . i.e. it prints the contents of a variable.)

The variable PRINTER-ON? is 0 when output is going to the terminal only and 1 when it is being reflected to the printer, so you can switch reflection on and off from inside a colon definition.

The variable WARNING is normally set to 1. If you set it to 0, error messages will come out as numbers instead of as text. Message 0 means a word can't be found in the dictionary and others are relative to the start of block 4. The main reason

for this facility is to allow you to use a non standard disc without getting nonsensical error messages. If WARNING is set to -1 error messages will be read from disc, but you will no longer get the "isn't unique" message that you normally get when you redefine a word.

One special use of variables in xForth is for execution addresses. Certain variables such as XPAGE and XCURSOR contain addresses of definitions that are executed by other words. This makes it easier to alter installation-dependent features such as cursor addressing. The Appendix 'Altering your system' tells you how to do the latter; let's look here at XPAGE. Whenever a word calls on PAGE to be executed (for instance, TRIAD does so) the definition stored in XPAGE is actually used. On delivery this is set just to be CR . Suppose your terminal needs control/L to clear the screen (or your printer needs it to do a paper throw). Define

```
: (page)  CTRL L  EMIT ;
```

(note that CTRL L gives the ASCII code for control/L - we'll learn more about this later) and then put this definition into XPAGE by typing

```
XPAGE REPLACED-BY (page)
```

The word REPLACED-BY takes the execution definition of the word following it and assigns it to whatever address was on the stack, which is just what we want. Whenever you make a change like this, type PROTECT because FORGET (page) would leave XPAGE pointing into limbo.

Chapter 8

Text strings and characters.

Strings of text are handled as single entities almost like numbers. You can declare

```
16 STRING Customer's-name
```

which makes the word after STRING into a string variable with room for up to 16 characters (or in general, for up to the number of characters given by the stack top). Then

```
" Smith" Customer's-name $!
```

will assign the text 'Smith' to the variable. Note that $! should only be used to assign to string variables. It trims the string at the right if necessary to fit in the maximum length you have asked for.

Now when you type Customer's-name two items will be left on the stack: the address of the first character and, on the top, the present length of the text stored. This means that, for example, TYPE will type it out. If you type

```
4 STRING abbrev Customer's-name abbrev $! abbrev TYPE
```

you'll get 'Smit' typed at the terminal.

Strings are joined (concatenated) using $+ as follows:

```
Customer's-name " -Jones" $+ TYPE
```

types 'Smith-Jones'.

Note that string literals, defined by the word " which reads text up to the next " , leave the same stack information as string variables so they can be used in exactly the same way except that they can't be assigned to. The word " works inside colon definitions just as ." and (do. When you use " from the terminal but outside a colon definition, you should realize that " merely returns the address of the relevant text in the input buffer, so you must not expect the string literal to still be there when you've typed another line. Thus you can

type

```
" abcd"  " efgh"  $+  TYPE
```

and get abcdefgh but if you type

```
" abcd"
" efgh" $+ TYPE
```

you'll not get what you probably intended. This never causes any problems in practice because the second example is unrealistic, but it is as well to be aware of what " is actually doing.

Strings are compared using $= which returns true if they are identical, and $< which returns true if the first is alphabetically prior to the second, using the entire ASCII character set. So

```
Customer's-name " Adamson"  $<
```

leaves a false flag (0) on the stack.

xForth also provides facilities for dealing with single characters. It doesn't seem worth saving single bytes by having special variables for characters, but you do need single byte store and fetch. These are C! ('c-store') which stores the low-order byte of the second-top stack item in the address on the stack top, and C@ ('c-fetch') which replaces the address on the stack with the byte contents, setting the high-order 8 bits of the stack top to 0.

A common requirement is to get the ASCII code of some character for comparison with some input command. This is done with ASCII which takes the first character of the next word and leaves its ASCII code on the stack. It works in or out of colon definitions. Similarly, CTRL leaves the low-order 4 bits of the next word's first character, so

```
CTRL C      and      CTRL c
```

both leave 3.

If you want to read a string in during a program, the easiest way is to use EXPECT$ which takes an address and a maximum length and returns a string literal. So

```
PAD 40 EXPECT$
```

would allow you to type in up to 40 characters terminated by a retrun, with all the usual facilities like rubout, printer toggle, break with control/C and so on. The string would be

stored at PAD which is the address of a scratchpad area of store that can hold at least 80 characters. Then the address of PAD and the actual length of the string would be left on the stack so your program could do

```
Customer's-name  $!
```

to store the information away.

The words #->$ and $-># are used for internal number formatting. The first converts a double precision number to a string, so

```
S->D #->$ TYPE
```

is equivalent to 0 .R because S->D converts a single precision number to a double precision number. The second word $-># tries to convert a string to a double number. If the string contains anything other than digits and possibly a leading minus sign, the number found so far is left and a FALSE flag is left on top of it. Otherwise the number is left with a TRUE flag. So you can do things like

```
: get-number
        BEGIN   PAD 10 EXPECT$  $->#  NOT WHILE
                ." ???" CR
        REPEAT
        DROP ; ( DROP converts double to single )
```

Chapter 9

Virtual memory.

One of the most powerful features of all proper Forth systems is the virtual memory system we learned about in Chapter 5, which gives you the effect of a much larger memory by swapping 1K blocks to and from disc. These blocks (sometimes called screens) are handled automatically by commands such as LOAD and SEE but you can use them yourself directly via the words BLOCK and UPDATE. For example, 14 BLOCK returns the address of a buffer containing the 14th 1K block, which will be read if need be from disc. So 14 BLOCK C@ will give the contents of the first byte of the block, while

```
14 BLOCK B/BUF 0 DO DUP I + @ foo 2 +LOOP DROP
```

inside a colon definition passes the word 'foo' the 512 sixteen-bit integers contained in block 14, assuming foo leaves nothing on the stack.

Blocks are read in and out of buffers automatically, so the address given is only valid until the next call of BLOCK (or LIST or any other word that uses BLOCK). If a buffer is needed for another block, its previous contents are written back to disc if they have been updated: you mark the contents as updated by calling UPDATE (which doesn't touch the stack) while the buffer address is still valid.

To associate a named file with a virtual memory segment, you have to create a file structure and then assign it to that segment, like this:

```
FILE name.ext        5 name.ext fassign
```

will create a file structure called name.ext by xForth and (initially, at least) referring to a CP/M file called name.ext on the current default disc drive. Then the file will be associated with the 5th virtual memory segment, namely blocks 5001 to 5999. So now, 5001 BLOCK C@ will give you the first character in the file name.ext. If the file is newly created, that character will be a CP/M end of file marker (control/Z) followed by 127 zero bytes. When you are finished with the file it's a good idea to tidy up by typing

5 frelease

which removes the association you set up earlier.

There are 8 segments available for normal use, numbered 0 to 7. For more details see the Appendix on the filing system.

Notes:

1. SAVE-BUFFERS ensures all updated buffers are written to disc. Use it before doing anything risky. Note that BYE calls SAVE-BUFFERS automatically so get in the habit of always logging out with BYE rather than just switching off.

2. EMPTY-BUFFERS marks all buffers as empty, so undoing the effect of any updates.

3. See also the operating system interface description in Chapter 10 and the Appendix describing the filing system.

021

Chapter 10

Interfacing with the operating system.

This section describes how xForth for CP/M2.2x systems interacts with CP/M, how you can use CP/M's facilities, and how to access 8080/Z80 input-output ports.

Virtual memory blocks are 1K segments of random access files. The file for blocks 1 to 999 is set up on loading to be FORTH.BLK on the current CP/M default drive and if you like you can work with this all the time, as long as you remember to leave blocks 1 to 48 for the system's use. Within xForth this file is called SYSFILE. Note that error messages are always read from blocks 4 to 7 so you should not change these blocks. Also, it is a good idea to make sure you have a backup copy of the original FORTH.BLK file in case you clobber it, since it contains the filing system and the structuring and input/output words, loaded by SYSGEN.

To copy blocks either within a file or between two files, just use COPY for a single block or COPIES for several, giving the appropriate block numbers, as in 20 1020 12 COPIES. This is explained in Appendix A.3.

For more information on the filing system, read the relevant Appendix. On the whole, it's best to use lots of short files rather than one enormous one, since LOAD-FILE can happily be called from within another file being loaded so you can have one master file that loads many others. It may be useful to you to know that CP/M's PIP doesn't copy properly in the case where you've used, say, blocks 1 to 54 and 82 to 99 of a file: the second part will not normally be copied. This is a well-known PIP bug and is nothing to do with any limitation of xForth. So use xForth's COPIES command instead.

You can access any of CP/M's system functions using the word CPM-CALL which takes the stack second top as the parameter to be left in the DE register and the top item as the number of the function. So 2 14 CPM-CALL DROP will select drive B as the default drive. Another example is 0 13 CPM-CALL DROP which resets the disc system and allows you to change discs. If you do this, type SAVE-BUFFERS first to make sure everything is safely on disc. The contents of the HL register on return from

CP/M are left on top of the stack, since they often have necessary information. This is why we needed DROP above. Unfortunately, imitations of CP/M such as CDOS don't leave this information compatibly with CP/M2.2, so an additional word CPM-CALLb is provided to leave the contents of the accumulator on return from the CP/M call.

To allow direct use of i/o ports from xForth, the words P! and P@ are supplied. They work very like normal store and fetch operations:

127 15 P! and 14 P@

respectively send 127 to port 15 and read port 14, leaving the result on the stack.

Chapter 11

Other topics.

This section contains miscellaneous information. It covers <CMOVE> FILL R> >R R@ and it refers to double number formatting and mentions some advanced topics, namely recursion and the words IMMEDIATE COMPILE [ ] [COMPILE] CREATE DOES> Most of what is described here is not covered in detail.

To move blocks of memory around, use <CMOVE> which is a 'smart' word in that it handles overlapping blocks correctly. It is pronounced 'bidirectional c-move'. The arguments are

from to #bytes <CMOVE>

where 'from' is the address of the first byte of the block to be moved, 'to' is the address of the first byte of the destination, and '#bytes' is the number of bytes, treated as an unsigned integer in the range 0 through 65535. FORTH-79 defines a word CMOVE that only works safely for non-overlapping blocks and that treats the number as a signed integer. We recommend you use <CMOVE> instead.

To fill a block of memory with a single byte, use FILL for which the arguments are addr #bytes byte FILL.

The words >R R> and R@ deal with the return stack, which is what xForth uses to keep track of where it is in the program. It can be used for temporary storage (within colon definitions only) by executing >R ('to-r') which transfers the normal (parameter) stack top to the return stack, and R> ('from-r') which brings it back again. These must be used with great care, and must always be balanced correctly within any level of structure of a word. Moreover, they shouldn't be used within a DO ... LOOP construction since the return stack is used for index manipulation. You can read whatever you left on the return stack without deleting it from there using R@ ('r-fetch').

It is possible to handle double precision numbers; definitions are given to add and subtract them and to do the basic stack operations. A versatile set of formatting words is available for double number output. All of these facilities

are standard Forth facilities - see any book on Forth.

Look at the '8 queens' program described in the examples Appendix as a non trivial program with some information on the topic of recursion. In general, recursion in xForth can be done in several ways but the easiest is via the word MYSELF which makes a word call itself. Another and more versatile way is to use execution variables as in

```
VARIABLE  forward-ref
: this  forward-ref @  EXECUTE ;
: that  this ;
forward-ref  REPLACED-BY that
```

which makes 'this' call 'that' and makes 'that' call 'this' - to no avail here since we've just set up an infinite loop which will finally terminate when the return stack overflows.

You may have wondered how the words (" ." ASCII and CTRL (not to mention all the structuring words) manage to do their tricks when the interpreter is supposed to be compiling rather than executing everything it sees. The answer is that it is possible to mark a word for execution even when everything else is being compiled: if the word IMMEDIATE is executed, it marks the most recent dictionary entry so that the interpreter will know it is to be executed even if found inside a colon definition. Once we start getting involved with this, we have to get into the advanced topics of words like COMPILE and [COMPILE] and then into [ ] CREATE DOES> which would take too much explaining for this introductory manual. If you get to the stage of needing to find out about them, it's time to join FIG and get some books on the subject! The xForth Technical Manual has information on these topics, but it assumes you've absorbed everything in this manual first.

Appendix A

The screen editor.

xForth's editor is a screen editor that is specially suited to the block-based virtual memory disc system needed for the FORTH-79 standard. It is compatible with nearly all cursor-addressable VDU's, since it can scroll both horizontally and vertically if your VDU screen is too narrow or too short; in addition, since the source code is supplied it can be adapted for any special requirements you have.

Merely changing the key bindings (i.e. saying which special key is for what action) doesn't require you to touch the source code. There is a special program to do that interactively, described later. As delivered, the command keys are set up as shown in the table later in this Appendix: most are simple mnemonics of the form control/S for search, but there are too many actions to fit happily into the set of control characters, so some are of the form ESC[1] followed by a character. This fits in well with the special function keys of many terminals, which send ESC followed by a character. By following the table you will be able to use the editor until you have time to change the command keys to suit yourself.

The cursor addressing is set up during normal xForth configuration; the configuration program knows about the more common VDUs such as those that use DEC VT52 compatible cursor addressing. Unusual cursor sequences require you to do some minor programming tasks, as explained in the Appendix 'Altering your system'.

---

1. ESC is the code control/[sent by the key marked ESC or ESCAPE or ALTMODE on most terminals.

A.1 Starting up the editor.

The usual way to enter the editor is to type, say,

```
SEE-FILE my-defs.blk
```

to invoke the editor starting at the first block of file MY-DEFS.BLK. As usual, Forth79 compatibility is maintained in that you can also type

```
12 SEE
```

to edit block 12 (and blocks before and after this, if you like). If you are loading a prepared set of definitions from disc, either with LOAD or with LOAD-FILE, and an error message is given, you can type WHERE to go straight to where the error occurred: the block being read when the error was detected will be shown with the cursor positioned immediately after the offending word.

A.2 Using the editor.

The editor is simple yet versatile. A cursor is moved around the screen by control keys. Whatever you type appears at the cursor position, either overwriting the present contents or displacing the present contents to the right, depending on the mode which you can change with another control key. You can delete a single character, causing the text to close up to fill the gap, you can delete a whole line, causing the lines below to move up, or you can delete it but leave a blank line instead of moving up the lines below. You can search for a string, with or without replacing it by another one. You can copy a line to another place, move to the block after or the block before the present one, or exit from the editor with or without writing the updated block to disc.

Assuming you've set up the screen width and depth correctly during configuration, the screen will scroll sideways and/or up and down whenever necessary to keep the cursor in the visible part.

The following description gives names to each of the editor actions. The keys corresponding to the names are shown in the

table, e.g. **fwd** (meaning forward to the next block) is control/F, obtained by holding down the control key and typing F.

When you enter the editor by typing something like 12 SEE the block appears on the screen with the cursor in the top left (home) position. To return the cursor there at any time, use **home**. To move the cursor right one place, use **right**; to move it left use **left**; to move it up use **up**; to move it down use **down**; and to move it right in multiples of 8 columns use **tab**. You will find your terminal's repeat key (or automatic repeat) useful for moving the cursor rapidly.[2] The **newline** key functions as you'd expect, moving the cursor to the start of the next line so you can type in text normally.

While typing in text, you will find the **rubout** and **cancel** keys useful: the first deletes the character immediately to the left of the cursor position, closing up any text to the right and putting a blank in the last column, and the second replaces the whole line with blanks. Like all keys, these may also be used during correction of previously prepared text.

Initially the editor is in overwrite mode, as shown in the top right of the screen. This means anything you type replaces what was there before. Issuing the **mode** command changes you to insert mode, in which whatever was at the cursor position moves to the right as you type in new text. Anything that disappears off the right hand edge is lost. Issuing **mode** again returns you to overwrite mode.

To delete the character at (rather than before) the cursor, use **del-ch**. The line closes up and a blank appears at the end. To insert a line before the cursor line, use **open**. The lines below scroll down, leaving a blank line to work with. Whatever was in the last line of the block is lost. To delete the whole cursor line use **del-line**. The lines below scroll up, and blanks appear in the last line. The line you have just deleted is saved (in the xForth PAD) and can be recalled by **from-pad**, which takes whatever was last saved and copies it to the cursor line, destroying the line's previous contents. You will see a copy of what's in the PAD at the foot of the screen. To save a copy of a line without deleting it, use **to-pad**. Saved lines survive outside the editor for a short time, so you can save a

---

2. However, take care with terminals having cursor movement keys that send multiple characters, as sometimes escape sequences can be missed if the repeat key is used, so that spurious characters appear. The best thing to do if you get this problem is to use the **prev-cmd** key to do the repeating, making sure you've bound it to a simple control character!

line, exit, type say 99 SEE, and then copy the line into block 99. (Note the commands **fwd**, **back**, **start** and **end** described later: they may save you from having to exit.)

On rare occasions you may want to write control characters in the text. This is done with the **quote** key, which puts any character following it into the cursor position (either inserting or overwriting) regardless of what that character may be. So to insert an ESC you hit the **quote** key and then the ESC key. If your VDU has a reverse video or dim mode or the like, you can alter the word *EMIT in the editor to display a character in this mode, and then control characters (and characters with the high bit set) will be shown like this. An ASCII delete (127) will be shown as a reverse video or dim question mark.

To search for a string, use **search**. This is essentially self-explanatory, since it asks for the string and you terminate it with the return key as you'd expect. However, here are a few notes:

- When you are asked for the search string, note that it is entered with the normal xForth interpreter's editing conventions for character deletion and so on, rather than with whatever keys you have set up for SEE.

- The search starts with the character just after the cursor, so if you search for xyz and the cursor is on an x, that x won't play any part in the search. This is done to simplify repeated searches as described below.

- If you want to go on with the search, you can use **prev-cmd** to continue without being asked for the search string again. The search continues until a CP/M end of file is encountered, so the search facility is most suitable for use with SEE-FILE rather than with SEE.

- If you get unexpected results you might like to remember that in Forth79 blocks, the end of one line is adjacent to the start of the next!

You can also do global replacements: use **replace** instead of **search**. You will be asked for a new search string and then for the string to replace it. The replacement string you give will be trimmed, or padded on the right with spaces, to make it the same length as the search string. Again, **prev-cmd** will continue the operation, using the same strings as before.

If you want to replace a block's contents completely, use **clear** to fill it with blanks. Since this is a potentially disastrous command, you will be asked for confirmation. Note that a newly created block will automatically be filled with

blanks, and there will be a message at the top of the screen indicating that this is a new block.

To move on to the next block, use **fwd**. The present block may or may not be written to disc if it was changed: this depends on whether the virtual memory system needs the memory space at present. Similarly, to move back a block use **back**. To move to the end of the file use **end** and to move to the start use **start**. The latter is especially useful with searching and replacement since when a search fails you are left with the block containing the last match, or the block you started from if there was no match. You can use **start** to go back to the beginning and try a different string.

To finish editing and ensure that all changed blocks are written to disc, use **finish**. To abandon editing and scrap any work you've done on the present block, use **abandon**. In this case, the block you're on is discarded but all other blocks are written back to disc to preserve the integrity of the filing system. (Note that this is different from the effect of ABANDON in the old xForth editor.)

Table of editor keys as set up on delivery.

Action	Keys
left	^H or backspace
right	^L
up	^K
down	^J or linefeed
home	ESC H
tab	TAB or ^I
newline	RETURN or ^M
rubout	Key marked RUBOUT or DELETE
cancel	^X
mode	ESC I
del-ch	^D
del-line	ESC X
open	^O
from-pad	ESC <
to-pad	ESC >
quote	^Q
search	^S
replace	^R
prev-cmd	^P
clear	^C
fwd	^F
back	^B
start	ESC S
end	ESC E
finish	^Z
abandon	^A

Notes: ^H means control/H etc. A sequence like ESC A means an escape character (control/[) is sent and then an A is sent. Most terminals' special function keys work in this way, so it should be easy to set up the editor for your terminal. Any control keys or escape sequences not shown are rejected: the terminal beeps.

A.3 Changing the key bindings.

To change the relationship between keys and actions, type

LOAD-FILE BINDINGS.BLK

and wait while the program loads. You will be asked if you want to delete all the old bindings. This will set all control keys just to beep - handy if you want to remove an old set of key bindings before you start. If you have just generated a new system from the kernel, the keys will be set up like this and you'll have to set the bindings to make the editor usable. Otherwise, there's no need to re-initialise everything since you will be told what action corresponds to any key before you change it.

Then you will be asked to hit a control key (or the ESC key followed by a key) and state what action from the table is to be bound to that key. Several keys can have the same action. To make a key have no action, give its action as BELL. To finish, type return when you're asked for an action. (The original action for the present key will remain.) Note that after ESC, any of the keystrokes **A** **a** or ^A will have the same effect - you don't need to worry about what shift you're in. As a result, when you type things like ESC > you may see something else reflected, but you can still type the mnemonic that suits you best: for example, the > in ESC > is intended to be an arrowhead.

A.4 Editor-related functions.

To copy blocks use COPY and COPIES. The first behaves in the obvious way: 15 7061 COPY would copy block 15 to block 7061. Multiple copying is done as in 15 7061 2 COPIES which would copy block 15 to 7061 and 16 to 7062. Overlapping sequences are handled correctly, so 1 2 30 COPIES and 2 1 30 COPIES both do what you'd hope.

By the way, PIP (at least in the versions of CP/M we've used) doesn't work properly if there are holes in the file. That is, if you've used a file and created, say, block 2 but not block 1, then PIP will fail to copy it. This is a bug in PIP, not in xForth. Use COPIES to copy individual blocks in

'funny' files like this. Any file created by SEE-FILE will be perfectly all right to copy with PIP - it's only if you've been doing your own direct virtual memory access that there may be problems.

You may want to read the Appendix on the filing system to find how to handle copies between different files.

Appendix B

Using the FIG Editor.

Note:

Appendix B is part of the Forth Interest Group Installation Manual and describes the FIG editor, supplied with xForth as file FIG-ED.BLK. The description was written by Bill Stoddart. It and the editor itself have been updated by A.I.M. Research to meet the FORTH-79 Standard. Like all publications of FIG, this Appendix (but not any other part of the xForth manual) may be freely copied provided the following notice is included:

> This publication has been made available through the courtesy of the Forth Interest Group, PO Box 1105, San Carlos, CA 94070, USA; and of A.I.M. Research, 20 Montague Road, Cambridge, England.

B.1 Loading the editor

The FIG context editor is loaded by typing LOAD-FILE FIG-ED.BLK from xForth. It can be installed permanently in your system in place of the screen editor: see the Appendix 'Altering your system'.

B.2 Selecting a block and input of text

To start an editing session, the user loads the editor if necessary and then types EDITOR to invoke the appropriate vocabulary. To end it later, it is important to type SAVE-BUFFERS to ensure that changes are written to the disc, and then to type FORTH (or DEBUG if the debug vocabulary is being used) to reset the vocabulary to normal. Note that if the vocabulary is not reset, very strange things can happen since, for example, the word 'I' has a different meaning in the EDITOR vocabulary from its meaning in the FORTH vocabulary.

The block (or 'screen') to be edited is then selected, using either:

```
n LIST          ( List block n and select it for editing ) OR

n CLEAR         ( Clear block n and select it for editing )
```

To input new text to block n after LIST or CLEAR the P (put) command is used, as in:

```
0 P This is how
1 P to input text
2 P to lines 0, 1 and 2 of the selected block.
```

B.3 Line editing

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

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

Line editor commands

n H	Hold line n at PAD. Used by system more often than by user.
n D	Delete line n, but hold it in PAD. Line 15 becomes blank as lines n+1 to 15 move up one line.
n T	Type line n and save it in PAD.
n R	Replace line n with the text in PAD.
n I	Insert the text from PAD at line n, moving down the old line n and following lines. Line 15 is lost.
n E	Erase line n with blanks.
n S	Spread at line n. Line n and subsequent lines move down one line. Line n becomes blank. Line 15 is lost.

B.4 Cursor control and string editing

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

Commands to position the cursor

TOP	Position the cursor at the start of the block.
n M	Move the cursor by a signed amount n and print the cursor line. The position of the cursor on its line is shown by a _ (underline).

String editing commands

F text	Search forward from the current cursor position until the string 'text' is found. Leave the cursor at the end of the text string, and print the cursor line. If the string is not found, give an error and position the cursor at the top of the block.
B	Used after F to back up the cursor by the length of the most recent text.
N	Find the next occurrence of the string found by an F command.
X text	Find and delete the string 'next'.
C text	Copy in text to the cursor line from the cursor till the end of the string 'text'. Note: Typing C with no text will copy a null into the text at the cursor position. This will abruptly stop later compiling, since a null marks the

end of the input stream. To correct this error, type TOP X 'return'.

B.5 Block editing commands

n LIST	List block n and select it for editing.
n CLEAR	Clear block n and select it for editing.
n1 n2 COPY	Copy block n1 to block n2. Note that this and the more general word COPIES are contained in the file COPY.BLK and are described in the screen editor ('SEE') Appendix.
L	List the current block. The cursor line is re-listed after the block listing, to show the cursor position.
SAVE-BUFFERS	Used at the end of an editing session to ensure all entries and updates of text have been transferred to disc.

Appendix C

The demonstration package and the basic examples.

This Appendix describes the examples supplied with the basic system as well as the demonstration package. The first five examples are the basic ones and the rest are from the demo package, though the 8 queens example from the basic package is a lot harder to understand than most of the others. If you have the option, look at later ones before this.

The demonstration package consists of a set of applications which are either useful or enjoyable in themselves, or which show how to do particular things. For example, you will see how to modify the xForth language, adding new data types and operators and even new structure words. The best way to learn Forth is to use it and to try to understand Forth code, and this package is intended as much as a teaching aid as anything else.

The following notes are a brief guide to the main features of the programs. In general, you type LOAD-FILE name.blk to load the package called NAME and you type SEE-FILE name.blk to edit it.

C.1 Jack

For a simple set of definitions that's just for fun, type LOAD-FILE JACK.BLK and then type a return every time you've finished reading what's on the screen. List the file to see how the effect is achieved. The original was by Frederick Winsor, and the Forth version by Bill Ragsdale.

C.2 Fractions

The file FRACTION.BLK defines arithmetic operations on fractions. Type LOAD-FILE FRACTION.BLK and then

```
1 2 1 4 fr+ fr.
```

and get 3/4 as output. List the file. The first word gcd was contributed to 'Forth Dimensions' by R. L. Smith. If you don't know Euclid's algorithm for finding the greatest common divisor of two numbers, you may find it mysterious, but it's certainly concise!

The other words should be clear. As an exercise, try changing fr. to give 2/1 instead of 2 in response to 78 39 fr. Ask yourself if the word simplify needs to appear as often as it does.

C.3 Random numbers.

Another numerical example is in the file RANDOM.BLK. It produces pseudo-random numbers, useful in games and in simulation programs. This is based on a 'Forth Dimensions' article by J. E. Rickenbacker. To use it type (say)

```
25 RANDOM
```

to get a random number between 0 and 24. The workings should be apparent, though unless you know about random number generators you probably won't see why some of the arithmetic is as it is.

C.4 The Sieve of Eratosthenes.

The file SIEVE.BLK contains a program that calculates all the primes less than 16384 in about 7.8 seconds. This was used as a benchmark test in Byte and has become very popular among software sellers as 'proof' that their system is the best. The xForth program runs at about 1/5 of the speed of an optimized compiled program (in, say, Whitesmith's C) and up to 1000 times

as fast as some Basic versions. It is slightly faster than FIG Forth according to the figures in the Byte article.

C.5 The eight queens problem.

On the file QUEENS.BLK there's a program developed from a Forth Dimensions submission by Jerry Levan. It solves the eight queens problem in chess: find all the ways you can put 8 queens on a chessboard such that none of them threatens any other. In this version you can actually have a chessboard that's any size from 1 by 1 to 12 by 12, with the requisite number of queens. You type 8 queens or 3 queens etc. The program is quite hard to follow, but the main reason for its inclusion is that it shows you how to do recursion, i.e. how to let a word call on itself.

In xFORTH, a word can call on itself in several ways, the easiest of which is to use the special word MYSELF as follows

```
: a-word  some-words
     IF MYSELF
     ELSE 1
     ENDIF
     some-more-words ;
```

The effect is as if MYSELF had been replaced by a-word. The reason it's done like this is that until the ; has been reached, a new definition is made invisible to help in error checking. This has the useful side effect that you can do things like

```
: LIST  PAGE  LIST   BEGIN ?TERMINAL UNTIL  ;
```

to define a new word called LIST in terms of the old one.

The idea behind the main word 'try' of the queens program is that once a queen has been placed, you can delete its row and column from the chessboard and then solve the problem for the new, smaller board with one queen fewer. A complication is that the smaller board has to have some squares painted out because they're threatened by the queen's diagonal moves. This is dealt with by the words 'same' 'mark' and 'unmark'.

C.6 Quicksort.

The file QUICK.BLK contains an implementation of the quicksort algorithm. (See Knuth, 'Searching and Sorting'.) One of the nicest features of Forth is that you can sort anything you like - numbers, words, database records - just by changing the word "<" to do the required job. There is a suggestion for strings in the text of the file; to sort, say, integers in descending order you only need change the definition of "<" in the first block to

```
: "<"  > ;
```

See the last block of the file for a simple example.

Quicksort is another example of a recursive process. Contrary to what some people say, recursion is not necessarily inefficient and in xForth, this particular program runs fast.

The work is all done in the word 'partition' which splits an array into two pieces, with all the elements in the left piece less than or equal to a given element and all those on the right greater than or equal to that element. The special element is taken to be the initial last element of the array, but other choices are possible. There are several tweaks one can make for efficiency - at present, there may be as many as around 1.5nlogn comparisons[3] for n elements instead of the nlogn or so that can be achieved, because the word partition avoids special cases like the plague, and so sometimes takes longer than it might. For all normal use, the effort of improving the performance is almost certainly not worthwhile.

C.7 SEARCH

This is a simple utility for searching through some blocks for a piece of text. It uses the search facility in the FIG editor, so load FIG-ED before you load SEARCH. You use it like this:

---

3. logs are to base 2.

```
EDITOR  10 16 SEARCH Some text to look for.
```

will look through blocks 10 to 16 and type out every occurrence of the text from SEARCH to the end of the line. The blank immediately after SEARCH is ignored, being eaten up by the interpreter, but all other blanks are significant. SEARCH has been put in the EDITOR vocabulary, but there is no reason why you shouldn't leave it in FORTH if you prefer. If you do keep it in EDITOR, don't forget to type FORTH (or DEBUG) when you are finished searching, especially as the FIG EDITOR redefines 'I' and so you will get some very mysterious goings-on in your loops if you remain in the EDITOR vocabulary.

C.8 crypt

This is an enciphering and deciphering word patterned after one in Software Tools. The source code gives full instructions on how to use it. The idea of choosing two passwords of different lengths is that this gives the same effect as a single long password and even if you were to encode a long string of nulls, it would be hard to find the key for decoding.

C.9 The 'to solution'.

If you have the assembler, load it and then load TO-SOLN.BLK. Otherwise load HILEV-TO.BLK which is slower, but still adequate for most purposes. The idea is to define a new data type (called INT here, for integer) which acts like a constant most of the time but can still be assigned to without fuss. This means the words ! and @ needn't be used except for writing to buffers and the like. As a result, one of the biggest sources of error has been removed and the code is easier to read. The method of assigning to an INT is to precede it with the word to as in

```
INT x  INT y   100 to x   x 3 + 6 MOD  to y
```

which sets x to 100 and y to 1. The magic is done by the words x and y themselves which look at a flag to decide whether to write their values on the stack or store the value from the stack. The flag is called %to and all that to does is set %to.

The assembler version is just as fast as @ and ! would be with standard FORTH-79 variables.

Examples of how to use 'to' are given in the next two programs.

C.10 Easter

Load the file EASTER.BLK. This calculates dates of Easter using Clavius's algorithm, as explained by Knuth in volume 1 of 'The Art of Computer Programming'. (Buy it! It'll teach you more about computing than a hundred other books.)

As an example, 1970 1990 Easters will print a table of dates of Easter for the years 1970, 1971, ... 1990. The code shows how to use 'to' and also shows how to make one defining word make lots of dictionary entries.

Notice how this program automatically loads the 'to' words if they aren't already there. It uses FIND to see if the needed words are in the dictionary and tests whether the result is zero using the { ! } construction described in the assembler Appendix. If the result of the FIND is zero (so the word isn't there) it loads the needed file, again checking whether the assembler is available to decide which file to load.

C.11 Hamurabi

If you load HAMURABI.BLK, the RANDOM and Easter words will be loaded automatically if required, using the trick described under the Easter heading. Hamurabi is a game from the People's Computer Company's famous book 'What to do after you hit RETURN'. It has spawned many imitators, and is a good example of a simulation-type game. Load it and then type

```
Hamurabi
```

and try to work out what's going on. Look at the source code later to find out how to read keyboard input from within a word, giving the user the usual facilities to delete characters, switch the printer on and off, and break out using control/C.

C.12 Exponent

This is a definition of a word ** that does integer exponentiation. It isn't as trivial as it sounds - it takes less than 20 arithmetic operations to raise something to the power 1000, for instance (though 16 bit integers can only hold 1 to the power 1000 anyway). This method is used in the floating point package. See if you can understand it.

C.13 Life

The file LIFE.BLK contains is a fast version of John Conway's game 'Life'. Most computing magazines have had articles on it. This implementation could be made even faster by improving the word +neighbours. The word Life itself expects a virtual memory block number to be on the stack. You can use the screen editor to set up an initial pattern: anything non-blank is considered to be alive. Note that you must have the cursor positioning commands and the word SEE working correctly before you can use this. Also, there is no check that your screen is big enough - it assumes you can use SEE without the automatic horizontal and vertical scrolling coming into action.

A sample pattern to start with is tacked on the end of the file LIFE.BLK and can be read by typing

```
INSTALL-$$$ life.blk     8005 Life
```

C.14 DisForth

This is a solidly useful program, yet is surprisingly short. Load DISFORTH.BLK and type

```
DECOMPILE  SEE
```

to reconstitute the original form of SEE. It can't cope with everything, but it's easy to add to, as all the special cases are dealt with in one place. If it runs off the end of a word

(e.g. SYSADAPT) hit control/C to stop it.

C.15 New-loop

Here is how to change the syntax of the language! The file NEW-LOOP.BLK contains definitions that overcome what we think are weaknesses in the FORTH-79 loop words. The new words are fewer, simpler, and more logically coherent than the existing ones. Be warned, though, that this is only a demonstration and there is no proper error checking. If enough people write to A.I.M. Research and tell us this is what they want, we'll produce fast and safe versions.

If you load it, your definitions can contain

```
begin  some-stuff  repeat
```

where 'some-stuff' can contain any number (including zero) of occurrences of the word while giving a direct generalization of the BEGIN ... WHILE ... REPEAT and BEGIN ... UNTIL and BEGIN ... AGAIN structures. There is no speed penalty at run time, but beware of typing WHILE when you really mean 'while' and so on.

If 'begin' is replaced by 'cycle' you get a properly designed counting loop with index 'i' though the given implementation is slow because it's all at high level. Thus

```
5 1  cycle  i .  repeat
```

types 1 2 3 4 5 (not 1 2 3 4 as you'd get with DO ... LOOP). You can also use 'while' for early exits:

```
B/BUF 1  cycle  word1  word2  ?TERMINAL NOT while
                word3  while
          repeat
```

If the final value is less than the initial value, 'cycle' does the right thing by not doing any iterations at all:

```
-1 0  cycle  ." Error"  repeat
```

prints nothing.

You can use +cycle for a step other than +1, as in

```
0 10  -1 +cycle  i . repeat
```

which prints 10 9 8 7 6 5 4 3 2 1 0.

Nested loops work correctly and the words 'j' and 'k' return the values of the loop counters one and two levels out. The return stack isn't used so if you like you can even read loop indices from within other words, though this is bad style.

C.16 Modules

Modular programming is a particular sort of structured programming. The idea is to write small modules (say, one block each) which have their own private words as well as public ones. You write applications that take the form

```
START-MODULE
  some-definitions
EXTERNAL
  more-definitions
END-MODULE
```

and then the definitions between START-MODULE and EXTERNAL will be private, known only to those between EXTERNAL and END-MODULE, which themselves are public, appearing in the dictionary normally.

Note that the module words use the stack so you must take care to leave it intact while you are making your definitions.

Appendix D

The Filing System.

The xForth filing system gives you a versatile but straightforward means of using virtual memory and disc files. It maintains compatibility with Forth blocks so that applications written for less advanced systems still work. For example, you can make

```
LOAD-FILE c:accounts.blk
```

do the same job as 134 LOAD might do on an ordinary system. You can have many random access files open at once, all looking like segments of virtual memory, and you can also access files sequentially, taking advantage of features, like pipes, which are not usually found on microcomputers. The two kinds of file access are described separately below, and if you are not using the sequential access facilities you need not load them, so they cost you nothing in terms of wasted memory. Before going into detail, let us look briefly at the simplest and commonest kinds of use. You can then try out the system without worrying too much about the details, and come back later on to understand more advanced use.

Suppose you have the demonstration package and you want to load the decompiler. You need only type

```
LOAD-FILE disforth.blk
```

(where the file name may be in either lower case or upper case) and the decompiler will be loaded and be ready for use. The word LOAD-FILE reads the word following it and tries to interpret it as a file name as defined by your operating system. For CP/M systems, the file extension .blk indicates that the file is a virtual memory image and can be loaded as above. A file being loaded may itself contain LOAD-FILE instructions, of course, so this is a very useful feature for organizing your programming.

If you want to read or edit the decompiler you can type

```
SEE-FILE disforth.blk
```

and the screen editor will come into action with the cursor at the beginning of the file. The block numbers will be shown as 8001 onwards for reasons explained later. Modified virtual memory blocks will always be written back to disc if they are changed, even if you exit using ABANDON. This is because the system keeps its house in order at all times to allow things like nested loading as described above. If you like to jump in and out of the editor without constantly retyping file names, there is a way to do so using INSTALL-$$$ as described below.

To list a file with line numbers, use LIST-FILE name.ext which handles both .blk and other files, by assuming that any extension other than .blk indicates a text file that is to be listed in an obvious way. You can pause or break the listing by hitting control/S or control/C as usual. LIST-FILE is particularly useful in conjunction with the printer, which is switched on and off using control/P or by setting the variable PRINTER-ON? to TRUE or FALSE.

D.1 Random access files.

Files are xForth data objects like strings, integers and so on. They are declared like this:

```
FILE data
```

which creates an xForth word called 'data' which contains all the information the system needs for file manipulation. The word can be anything you like, since it is not necessary for the xForth name to be the same as the CP/M name, but when you declare a file it is initially given the same CP/M name as its xForth name. (If this is an illegal CP/M name, even after conversion to upper case, you will get an error message but the xForth word will still be there ready for you to give a valid CP/M name to.) Since assignment of CP/M names to files need not be done until you are running a program, you can write applications in terms of files and then decide on the CP/M names of your files at execution time. To give a file a new CP/M name, use the word fname! as follows:

```
" b:results9.dat" data fname!
```

The word fname! (pronounced 'f-name-store') takes a string and a file argument and sets the file name to the string if the string is a legal name.

If you forget what a file's CP/M name is, you can find it by using 's-name like this:

```
data 's-name TYPE
```

will type out B:RESULTS.DAT. The word 's-name takes a file argument from the stack and replaces it by a string argument which is the CP/M name of the file.

To use a file you must allocate it a segment of virtual memory. This is done by the word fassign as follows:

```
2 data fassign
```

will assign virtual memory segment 2, i.e. blocks 2001 through 2999, to the file you have just declared. (If any other file previously owned segment 2, it will be closed and detached from the segment before data is assigned.) There are 8 segments, numbered 0 to 7, and on initial startup or after COLD has been typed, segment 0 belongs to FORTH.BLK on the default CP/M drive. (The xForth name of this file is SYSFILE.) You are advised not to change the allocation of segment 0 because error messages are read from there, and some internal buffer operations use this file. Blocks 1 to 48 of segment 0 are reserved for system use, even in systems which apparently do not use all of these blocks.

Virtual memory segments are all the same size - contained in the constant seg-size - and are normally 1000 blocks long, though if you type, say,

```
SAVE-BUFFERS 1024 ' seg-size ! COLD
```

the segments will become 1 Megabyte in (virtual) size. The value 1000 was chosen because 100 is sometimes a bit small and anything else is a lot less convenient for human beings to use.[4] Now you can type, say,

```
2071 1010 COPY
```

to copy block 71 of the CP/M file B:RESULTS.DAT to block 10 of the file B:FORTH.BLK.

Initially, blocks 2 to 7 are not allocated to any file. You can allocate them as explained above and then treat them just like pieces of virtual memory: if block 2 were assigned to the file B:RESULTS.DAT you could read or write that file

---

4. If you have a CP/M1.4 system, files can only be 256K in size so block numbers within a segment are taken modulo 256. Thus blocks 1 to 256 behave normally but 257 is the same as 1, 258 is the same as 2 and so on. Similarly 2260 is the same as 2004 etc.

starting at block 2001 or anywhere else. This makes file access hardly more complicated than writing to a memory location. See chapter 9 of the manual for details. Files are created if necessary when written to, and will be opened and closed automatically. If you try to read from a part that has not been written you will get a block whose first character is the CP/M endfile marker control/Z, followed by 127 nulls. This condition is recognised by the screen editor, and causes it to fill the block with blanks and announce it's a new block. (The block is not marked as updated until you actually do some editing on it, however.)

To find what file, if any, is allocated to a segment, use the word 'th-FILE as in 4 'th-FILE which will return 0 if there is no corresponding file and the address of the file otherwise. So

```
2 'th-FILE 's-name TYPE
```

gives B:RESULTS.DAT.

You can de-allocate a file using 'frelease' like this:

```
2 frelease
```

which is a good idea when you're finished with it as your valuable data will then be protected from mistakes you might make later.

For simple use, there is a quick and dirty way of handling files. The system maintains a temporary file called $$$ which always owns the segment just beyond the last user-accessible one, i.e. blocks 8001 to 8999 unless seg-size has been changed. You can just type

```
INSTALL-$$$ FILE.BLK
```

to make these blocks correspond to the file X:FILE.BLK where X: is the present default CP/M drive. So for example, if you had a file you wanted to do some editing on and also to do some copying to, you could install it as above and type, say,

```
15 8001 10 COPIES    8007 SEE
```

The assignment lasts until next time you use INSTALL-$$$ and in fact is restored whenever LOAD-FILE, LIST-FILE and SEE-FILE make temporary use of $$$.

D.2 Sequential input/output.

The commonest form of file access is to read or write one character at a time in order, starting from the beginning. Programs that only read one file and write another in this way are often called filters - they perform one operation on the file such as changing tabs to multiple spaces. It is often better to write a lot of simple filters and then string them together when required, as this allows you to put together new operations in terms of simple existing ones. For a full discussion of this and many other points of good programming technique, read the book 'Software Tools' by Kernighan and Plauger (Addison-Wesley, 1976.)

To read a sequential file, use the word getc which gets a character from the presently selected sequential input file. If the end of the file has been reached or the file does not exist, getc returns control/Z every time it is called. To write a character to the presently selected sequential output file, use putc as in

```
ASCII A putc
```

which writes the letter A. To write a string, use put$ as in

```
" This will go to the output file."  put$
```

If you load the sequential i/o operations by LOAD-FILE seq-io.blk (as you will have to do to use getc, putc and put$) the default output file is the console, except that output is spooled: that is, it is all collected together in a temporary file before typing out at the console. So if you had an xForth word poem that produced random gibberish, you could type

```
poem
```

and the whole output would be written to virtual memory and then typed back at you. The default input file is another spool file that is initially empty. Suppose you have written words encode and decode that use getc and putc to convert text between encrypted and plain forms. You could type

```
<< secure.dat  decode
```

to decode and then type the data from the file SECURE.DAT. The word << takes a file name following it and makes it the sequential input file. In similar vein, the word >> selects

the file name following it for output, as in

```
poem >> archive.txt
```

You can also redirect to the pseudo-file LST: which is the CP/M list device. Of course, you can combine these:

```
<< raw.dat  process  >> LST:
```

will let the word process take its input from the file raw.dat and send its output to the list device.

The simplest such filter is one that just does enough getc's and putc's to transfer its input to its output, and this is supplied for you. It is called, logically enough, copy, and one way to type out a text file without line numbers is

```
<< letter.txt  copy
```

If you redirect the output to LST: the file will be sent to the printer.

The words >> and << are called redirection words, since they redirect input and output from their defaults. This idea seems to have been most extensively used in the Unix operating system, where the equivalent of our xForth words would be different processes in the machine, all running at once. Another idea used in Unix is the pipeline. The output from one word can be piped (think of the output as a stream in the literal sense!) into another. In xForth the pipeline word is == which is supposed to look like a picture of a pipe. You might do something like

```
<< secure.dat  decode == process == encode  >> secure.res
```

to decode some data, process it, and encode the results.

The pipline word can be very useful since it lets you write small words that do simple jobs on files, and then put them together as you need them. An additional example is

```
<< unsorted.dat  sort ==  10 discard-all-but-last
```

which might type out the last 10 items in a list. This may seem inefficient, but the point is that if you aren't going to want to do this very often, why bother writing a special word to do it?

D.3 Some other useful words.

To examine your disc directory, use 1 DIR or 2 DIR which gives a listing very like the CP/M directory command. If you have some other number of disc drives than 2, you should change the drive number checking part of DIR (in block 16 in most systems). The reason for trapping illegal drive numbers at this stage is that CP/M gives a hard error if an illegal drive is called for, rather than returning an error code to the calling routine.

To delete a file, use fdelete with an xForth file identifier. If data is still as it was above,

```
data fdelete
```

will delete the file B:RESULTS.DAT and return a TRUE flag to let you know it managed. If fdelete fails you get a FALSE flag back. If you intend to do a lot of deleting, you can define a new word DELETE-FILE like this.

```
: DELETE-FILE  (file-voc)  INSTALL-$$$  $$$ fdelete
               NOT IF  CR ." Can't delete." CR ENDIF
               RESTORE-$$$ ;
```

The first word (file-voc) tells xForth that you are going to be using some definitions from the vocabulary called (file-voc). The second word INSTALL-$$$ reads the file name from the input stream at the time DELETE-FILE is executed, and makes that the name of $$$. Then $$$ is deleted and if the deletion fails (because the file doesn't exist, or is write protected) a message is given. Finally RESTORE-$$$ resets $$$ to whatever it was before INSTALL-$$$ was called.

You can now try to define a few file handling words of your own. A good one to start with is PRINT-FILE which prints a file without line numbers. Either use the method suggested in (b) or use the fact that LIST-FILE actually won't put in line numbers if the flag #s? in (file-voc) is set to FALSE. If you do it this way, don't forget to reset #s? before exit from PRINT-FILE. Note that #s? is a constant, so you have to use ' to get at its address so it can be altered. This is the same as in the case of seg-size above.

Appendix E

The debug package.

The debug package provides two things: extra protection against common errors, and a tracing facility that allows execution of a word or words to be followed either interactively or otherwise. A bonus is that execution can be paused or interrupted even if tracing is turned off. Since the source code is supplied, it is easy to add other facilities. An example might be a profiler, which records how often certain words are executed as an aid to locating 'inner loops' when speed is important.

The package is installed by loading the file DEBUG.BLK. The system is then left in an altered state. A minor point you might notice is that XOFF-CHAR is set to -1, so that CR doesn't intercept key presses. This is to allow the debug words easy access to the keyboard. Although scrolling control for listings etc is switched off, you can now pause and interrupt execution altogether for words that are in the debug vocabulary.

Other changes are more significant. Instead of new words being added to the FORTH vocabulary, and existing definitions being looked up there, words are added to a new vocabulary called DEBUG. (A vocabulary is merely a conceptually distinct part of the dictionary, rather like a separate chapter.) Words are also looked up in DEBUG, but DEBUG is linked to FORTH in such a way that if a word isn't found in DEBUG, the FORTH vocabulary will be searched for it. This means that, for example, the new definition of ! in DEBUG will be used but since @ does not appear in DEBUG, the original FORTH definition will be used for it.

The idea is to load DEBUG before testing any new definitions, then test your new ideas with the help of its facilities. If the DEBUG vocabulary has already been loaded, you merely type DEBUG DEFINITIONS before defining untested words and FORTH DEFINITIONS before defining tested ones. (Actually, you could use the DEBUG vocabulary at all times, but there is an execution time penalty which may be significant in some cases.)

Once your new definitions are working properly, you can FORGET them, then reload them into the FORTH vocabulary. By the way, FORGET does FORTH DEFINITIONS so if you want to use the DEBUG vocabulary after a FORGET, don't forget to type DEBUG DEFINITIONS. Words defined after FORTH DEFINITIONS will be added to and will use definitions from the original FORTH vocabulary; they will not be traceable, for example.

E.1 Protection.

The protection facilities divide into two groups: compile time protection and execution time protection. We take the latter first.

On the last block of the file a word ?! is defined to be executed before any of the words ! C! 2! or TOGGLE alters the contents of a memory location. It checks that the location is safe to write to and issues an error message if not. The definition of 'safe' is rather difficult, and the choice made here is a trade-off between speed loss and reasonable protection: it is possible to write to a few dangerous locations or to be prevented from writing to some safe ones. The commonest beginners' errors are trapped, however, such as X 1 ! instead of 1 X !

If you are trying to understand the action of ?!, note the use of [...] LITERAL to calculate an expression at compile time and store its value as a literal, to be put on the stack at execution time. Here the use is to find the code executed by something that is known to be a variable at compile time, to determine whether a variable is being assigned to at execution time. (A constant is also allowed, being checked for in the same way. Otherwise, any location in the disc buffer area and any location not protected against FORGET is permitted.)

More execution time checks are provided on an earlier block, where arrays are redefined to check they have legitimate index values. (If your original xForth system already did this, you will now have been supplied with code that doesn't, and so runs a little faster. As always, you should only use the unsafe code if speed is important and your definitions are debugged.)

On the very first block of the file the words >R R> LIT and LEAVE are defined to do compile-time checks. (The same is done for : later on.) This incurs no penalty at execution time. To do this they have to be made IMMEDIATE so that even when most words are being compiled, the redefined words execute and check

that they like their environment, then themselves compile the necessary execution-time actions. Except for >R and R>, the new definitions merely check that compilation really is taking place, since (for example) LEAVE alters the return stack with fatal effects if typed from a terminal. The new >R and R> do an additional job of making sure they are nested and balanced correctly. If they are not, an error message is given. If an R> appears without a preceding >R at the same nesting level, the error message says the return stack isn't balanced. If a >R isn't closed by an R> the error message says conditionals aren't matched if you are just leaving a conditional or looping structure, and it says the definition isn't finished if you've reached a semicolon.

E.2 Tracing.

The tracing facility works by redefining : to compile a call to the tracer as the first instruction in any word. To allow selective tracing, you can turn this off by typing make-untraceable and on again by typing make-traceable .

A word with tracing ability compiled will behave apparently normally unless you type a special key while it is executing, as described below, or you have previously typed trace-on or pause-on . (The opposites of these are, of course, trace-off and pause-off .) If trace-on has been typed, the name of every traceable word is typed when it is executed, together with a picture of the stack. If pause-on has been typed, the action is the same except that 'ok?' is then typed and the system pauses until you hit a key. The special keys below have their special actions and all others cause normal execution to continue.

Traceable words call ?TERMINAL as well as checking the state of the tracing and pause switches. Since ?TERMINAL has been redefined in the debug package, certain keys are treated specially. (Of course, any of your definitions that use ?TERMINAL will also have access to this facility, regardless of whether they are traceable.) The keys are as follows; all others are ignored.

Control/C causes immediate return of control to the terminal.

Control/P toggles the printer on and off, just as in normal input.

Control/S — temporarily stops everything until any key (except control/C, which behaves as above) is typed. This is useful for controlled scrolling of output, among other things.

ESC — acts to suspend execution: it returns control to the terminal, but saves everything including the previous terminal input buffer contents. If you type

```
resume
```

then execution will continue where it left off, but in the meantime you may have turned pausing or tracing on or off, altered the stack, or even edited blocks of virtual memory. Note that the interpreter now types 'ok+' to let you know you are not at the usual level of interpretation.

Appendix F

The assembler.

Most programs spend nearly all their time executing a very small part, known as the 'inner loop'. For example, Forth systems on microcomputers - including xForth - spend most of their time in the 'inner interpreter' which arranges for instructions to be threaded together in the correct order. If speed is important, inner loops can be coded in assembler while the rest of the program is written normally. This can result in a program that runs nearly as fast as a pure assembler program, while retaining xForth's advantages of compactness and ease of writing and maintaining.

This Appendix assumes you already know 8080 assembly language. It tells you how to define words which execute just like other xForth words but which operate at the machine code level, so bypassing the inner interpreter. You should never start by writing xForth words in assembler: write and test them at high level first, then if absolutely necessary, re-code the few words that are executed most often. (If you are not convinced, re-code a randomly chosen word from one of your programs. If it isn't an inner loop word, you will probably be disappointed at the small speed increase).

The file ASSEMBLE.BLK contains the source of the xForth assembler which is itself written in high level xForth. The file DUMP.BLK contains a dump utility producing output similar to that of DDT'S dump; use it like this: 20000 100 DUMP which displays the contents of 112 bytes starting at 20000. (The reason 112 bytes are displayed rather than 100 is that DUMP alway rounds up to a multiple of 16.)

To define a word that will execute directly, enclose it between CODE and END-CODE instead of between : and ; as you usually do. Then you type in assembly language instructions using Intel 8080 mnemonics except that reverse polish (i.e. stack-oriented) notation is used. For example,

```
A C MOV,
```

stores code to move the contents of the A register to the C register. The order of all the instructions is

source destination instruction

We use, say, 10 D LXI, where conventional assemblers use LXI D, 10. The comma at the end of MOV, or LXI, reminds you it's an instruction rather than a parameter, and so will actually store some code.

All the standard 8080 instructions are available (see the source code) together with some Z80 extensions like EXX, which switches back and forth between the two registers sets, EXA, which does the same for the accumulator/flag sets, PCIX, which jumps to the address in the IX register, and LDI, LDD, LDIR, LDDR, which perform Z80 semi-automatic block moves. Instructions like

0 IX LXI,

work correctly but not all Z80 instructions are supplied. With the samples given you can see how to make your own extensions.

While in the assembler you can still use xForth's structuring facilities. For example,

0= IF, 1 OUT, ENDIF,

will write the accumulator to port 1 if the zero flag is set. The code produced is exactly the same as a conventional assembler would produce given

JNZ L1 ! OUT 1 ! L1 ...

Similarly BEGIN, LDI, PE NOT UNTIL, performs a Z80 semi-automatic block move in which the LDI instruction has to be repeated until the parity flag is unset.

The available constructions are IF,... ELSE,...ENDIF, and BEGIN,...UNTIL, and BEGIN,... WHILE,...REPEAT, The tests performed by IF, UNTIL, and WHILE, have to be stated explicitly: PE for parity flag set, 0= for zero flag set, 0< for negative flag set and CS for carry flag set; these can all be negated by following them directly by NOT

For the rare occasions where explicit labels are needed, you can set them by saying, for instance,

LABEL L1 H D MOV,

and then any of the usual jumps like L1 JNZ, will work. Of course, labels are just xForth words so things like John's-label are legitimate. You must, however, take care: labelling a subroutine and then calling it is fine, but putting a label in the middle of some code (whatever for?) will cause

disaster unless you jump round it, since a label is a dictionary entry and will be entered along with the machine code being assembled.

To insert a character string in line, use " as in

```
LABEL string1 " This is string one "
```

which puts a one byte character count at the point labelled by string1 and puts the text immediately following.

Conditional assembly is often useful. For example, the source of xForth has a flag Z80 that is true when Z80 code is to be compiled, and false otherwise. The flag is tested by { which acts like IF except it takes effect at assembly time. The analogues of ELSE and ENDIF are ¦ and }. For example,

```
Z80 { A XRA, SBX, ¦ SSUB CALL, }
```

either inserts in-line Z80 code for 16 bit subtraction, or calls a subroutine. As it happens, {...¦...} can also be used in other places outside a colon definition so you can type things like

```
2 RANDOM      { : colour ."Black" ;
              ¦ : colour ."White" ;
              }
```

The last thing you must do in an assembler word is to thread your definition back into xForth. To do so, restore the original BC register contents (which point to the next xForth instruction to be obeyed) and jump to NEXT which is a label in the inner interpreter.

Example 1

```
CODE 2* H POP, H DAD, H PUSH,
       NEXT JMP, END-CODE
```

(Note that the 8080 stack pointer really does point to the xForth stack)

Example 2

```
Z80 ( CODE 2SWAP EXX, H POP,  D POP,
           EXX, H POP,  D POP,
           EXX, D PUSH, H PUSH,
           EXX, D PUSH, H PUSH,
           NEXT JMP,    END-CODE

    ! :    2SWAP ROT >R ROT R> ;
    )
```

(Note that (! and) must all be in the same xForth block)

Example 3

```
CODE BYE 0 JMP, END-CODE
```

Example 4

```
CODE  2/MOD  ( unsigned --- remainder result )
        H POP,  A XRA,          ( Clear carry )
        H A MOV, RAR, A H MOV, ( Left half   )
        L A MOV, RAR, A L MOV, ( Right half  )
        CS IF, 1 D LXI, ELSE, 0 D LXI, ENDIF,
        NEXT 2-  JMP,     ( Push D and then H )
END-CODE
```

A note on vocabularies

To avoid confusion between the many assembler mnemonics and other xForth words, the mnemonics are kept in a separate part of the dictionary that is only looked at if CODE or LABEL has been typed,and until END-CODE is typed. The separate part is

called a vocabulary and advanced programs like metaForth manipulate several different vocabularies. The interpreter searches the assembler vocabulary when looking for words, and if it fails to find a word it then searches the Forth vocabulary. This means you can use all of xForth's normal facilities for arithmetic and so on, while in the middle of a CODE definition. Note that CODE definition time is not considered to be compile time, so all words between CODE and END-Code execute at once, which is always what's required. All you really need to know here is that an error during a CODE definition leaves you in the assembler vocabulary. Type FORTH (or DEBUG if you have the debug package) to return to normal.

Appendix G

Adapting your system

G.1 General

You can alter or add to your system and then save the new version so it can be run directly from CP/M. For example, you might want to add the assembler permanently: in that case, just load it and then read on.

(a) The usual way is to make whatever changes you want, type n SYSADAPT (where n is however many virtual memory buffers you want - at least 2) and then save the system as described in the Preface 'Getting Started'. That is, you use the CP/M SAVE command to save a file XFORTH.COM of whatever size SYSADAPT tells you to use. (You can use CONFIG instead of SYSADAPT; this is useful if you intend to change assignments to things like the delete key.) If you are doing a lot of work with files, it is worth having more buffers: SYSADAPT won't let you allocate more buffers than there's space for, but of course if you're about to load a huge application then it's prudent to leave enough room for it. With the sequential i/o file package, it's sometimes useful to have as many as 16 buffers, since this reduces disc activity for the spool files.

(b) If you want to alter the editor or other basic xForth words, you can change the relevant blocks then exit to CP/M and type

A>KERNZ80 or A>KERN8080

to load a kernel system. Then type SYSGEN and the new system will be built. It can be saved as in (a) above. Block 1 handles the system building so by changing that you can custom-build a system to suit yourself.

G.2 Basic terminal handling.

The following information is not normally needed, but is included so you can understand the action of CONFIG. Terminals that have several possible screen formats can be dealt with by defining a Forth word that does whatever is needed to change the mode, then puts the correct values in things like C/L and L/S.

xForth needs to know how wide your screen is so it can wrap long lines and so the editor can decide whether and how to scroll. If you have a VDU that wraps lines but has the common bug that if you write in the last column it takes a new line without waiting to see if you were going to send a printable character next (e.g. SuperBrain and TRS80 Model II), you should tell xForth you have one column less than you really do have: 79 instead of 80, in most cases. This is not needed if you use CONFIG, since it subtracts 1 always if you say your terminal wraps long lines.

The constant C/L contains what xForth thinks is the number of characters per line for your terminal. It is set to 80 on delivery unless we have told you otherwise. To change it to, say, 79, type

```
79 ' C/L !
```

which uses the word ' to get at the address where the constant value is stored. Similarly, you can change L/S to the number of lines in your VDU screen.

Two variables you can alter are DEL-KEY and CAN-KEY which are respectively the character used to rub out the last character typed and the character used to remove the whole line. For example,

```
127 DEL-KEY !
```

will make the standard 'DELETE' key rub out characters and backspaces will then be reflected as ^H. Note that these keys only refer to standard input as done by EXPECT or QUERY, which are the words used by the interpreter. If you call KEY you get exactly what was typed, with no system intervention.

Another variable which has already been mentioned is XOFF-CHAR, which contains the code used in controlled scrolling of the screen. You could change it to space by using

BL XOFF-CHAR ! and then just hit the space bar to arrest output, or you could set it to some impossible value like -2 to switch off scrolling control. If you set it to -1 there is a special effect - both scrolling control and control/C breaking are switched off.

G.3 Cursor addressing

If you have a cursor-addressable terminal, the first thing is to tell xForth how to position the cursor. If your terminal wasn't mentioned in the configuration menu, you will have to write a special word. This is quite simple. For example, here is how we wrote a word to cope with the popular DEC VT52 method: this allows us to go to row r, column c relative to r=0, c=0 as top left, by sending ESC Y 32+r 32+c where ESC is the escape code control/[. This is done as follows:

```
: (cursor) CTRL [ EMIT
           ASCII Y EMIT
           SWAP 32 + EMIT
           32 + EMIT ;
```

Look at the manual for your terminal to find how to position the cursor. If, say, it uses control/P 32+r 32+c then type in

```
: (cursor) CTRL P EMIT
           SWAP 32 + EMIT
           32 + EMIT ;
```

Now test your work: type 0 0 (cursor) and make sure ok appears in the top left corner. Take particular care that it appears in the top left and not 1 character away in either the horizontal or the vertical direction. You will know you've added the wrong offset (32 above) if this happens or if the cursor is out of step with where it should be when you use the editor later. Now type 10 0 (cursor) to get it about half way down the left column and 0 40 (cursor) to get it about half way across the top row. This makes sure you have row and column the right way round.

Once all is well, type

```
XCURSOR REPLACED-BY (cursor)
```

and go to 1(a) above.

G.4 The screen editor.

The screen editor is now configured almost entirely by using the file BINDINGS.BLK which lets you choose the keys to have whatever functions you want.

If your terminal can display in reverse video or dim (we don't really recommend flashing) you can redefine *EMIT in the editor source to output a character in this form, which will show up control characters better.

If you make any changes, type LOAD-FILE SEE.BLK and test the modified editor very thoroughly. When you are sure all is well, go to 1(b) above.

G.5 Removing the screen editor

If you don't have a cursor-addressable terminal, you may want to remove the screen editor to save space, replacing it with the FIG editor supplied on the file FIG-ED.BLK and described in the relevant Appendix.

To do the replacement, edit block 1 to load FIG-ED.BLK instead of SEE.BLK, and alter occurrences of (EDITOR) in COPY.BLK to EDITOR. Then go to (b) above.

G.6 Prompts and showing the stack

To remove the stack prompt altogether, type

```
XPROMPT REPLACED-BY CR
```

To change it, say, so that the stack picture appears on your terminal's status line, define and test 3 words

```
save-cursor
to-status-line
restore-cursor
```

and then edit block 9 so that .STACK becomes

```
: .STACK save cursor
      to-status-line
      ...
      restore-cursor
      CR ;
```

Check your new definition and then go to 1(b)

Similarly you can change the "ok" message, for example,

XOK REPLACED-BY NOOP

will remove it altogether.

Appendix H

Bugs.

We believe our software is reliable and well-designed, but of course we welcome information that will help us to remove errors or make improvements. If you think there is a fault in the system we supplied, or even a bad design feature that is inherent in xForth rather than in the FORTH-79 standard, please let us know.

To help us help you, please make absolutely sure the fault isn't in your program and make sure you have done your best to isolate where the fault lies. Try to remove anything not relevant to the problem, and send us the shortest program you can together with output that displays the fault. You must show every step from loading the kernel system, typing SYSGEN (with blocks 1 through 41 exactly as in the delivered system) to the point where the error occurred. We are sorry that we cannot undertake to deal with errors in systems that have been altered or patched in any way, or in systems that use imitations of CP/M instead of CP/M itself. None of this is intended to intimidate you - it's merely to give us some chance of helping you!

Note that because of the great freedom xForth gives you, it is possible to crash the system by overwriting xForth, CP/M or the buffers, stacks or user variables. This is why we recommend the debug package and vocabulary for all normal use: it protects you against nearly all common errors. Even without the debug option, you are far better protected in xForth than in most other Forth systems, and if you take care with the words ! C! 2! R> and >R you should have no trouble.