Table of Contents

Your new xForth system

Copy your disc before doing anything else.

The upgrade from xForth 1 to xForth 2 consists of this documentation as described in the contents table, and in particular containing a glossary update, together with a disc or discs containing the following files:

XFORTH.COM	This is the normal xForth system. Call CONFIG when you first use it.
XF808014.COM	If you have an 8080 processor or your operating system is CPM1.4 or CDOS, or you have trouble as described in "Trouble" below, use this file instead of XFORTH.COM. You will lose the ability to access user areas in CP/M and will be restricted to 256K files. Not supplied with Torch systems.
FORTH.BLK	The usual FORTH.BLK file with some updates and with blocks 30 onward removed.
SEE.BLK	The screen editor with some minor updates. TORCHSEE.BLK for Torch systems has some slightly different facilities.
SEEDATA.BLK	Sets up editor keys from a file instead of from keyboard. TSEEDATA.BLK differs slightly.
CONFIG.BLK	Called by CONFIG, minor changes from version 1. TORCONF.BLK for Torch differs slightly.
BINDINGS.BLK	Changes editor keys. Same as old version.
NEWSIEVE.BLK	Faster Eratosthenes sieve.
SPOOL.BLK	A file listing background task.
DELAY.BLK	Multitasking clock words. Needs edits to adapt to your system.
TDELAY.BLK	DELAY.BLK configured for Torch.
BUFFER.BLK	Print buffer using multitasking. Needs edits to adapt to your system.

TBUFFER.BLK	BUFFER.BLK configured for Torch.
WINDOWS.BLK	Simple multitasking windowing demonstration for Torch, needs adaptation to other systems.
TASKDEMO.BLK	Demonstration of WINDOWS.BLK, for Torch only.
NARRATE.BLK	Modified file for turtle graphics owners. See comments on changes to xForth optional packages.
SAVE.BLK	For systems with no SAVE command e.g. Turbodos, you can load this file then type SAVE-AS MYXFORTH.COM after configuration.

Other files are not supplied since they are mostly the same as in xForth 1; you can copy them to your working disc. Note that certain optional packages need slight changes: see "Changes needed to xForth optional packages".

```
              B:DIR
B:DIR?

A>B:DIR
B: XFORTH    COM : LIFE      BLK : NEW-LOOP BLK : FORTH     BLK
B: SEE       BLK : SEEDATA   BLK : CONFIG   BLK : BINDINGS BLK
B: NEWSIEVE BLK : SPOOL     BLK : TDELAY   BLK : BUFFER    BLK
B: SAVE      BLK : WINDOWS   BLK : DELAY    BLK : TBUFFER   BLK
B: TASKDEMO BLK : NARRATE   BLK : QERROR   BLK : XF808014 COM
B: COPY      BLK : TTY-RUB   BLK : FIG-ED   BLK : JACK      BLK
B: QUEENS    BLK : SIEVE     BLK : RANDOM   BLK : FRACTION  BLK
B: OLDSTUFF BLK : SEQ-IO    BLK : VMOVE    BLK : ASSEMBLE  BLK
B: DEBUG     BLK : DUMP      BLK : QUICK    BLK : VLIST     BLK
B: SEARCH    BLK : CRYPT     BLK : TO-SOLN  BLK : HILEV-TO  BLK
B: EASTER    BLK : HAMURABI  BLK : EXPONENT BLK : MODULES   BLK
B>


  DIR
B: BUILD     BLK : XFCODE80 TXT : XFHILEV  TXT : XFTASK    TXT
B: METASYS   TXT : XFUPPER  BLK : SYSEQUTS BLK : CPM-I/O   BLK
B: CPM-DISC BLK : CPM-EQUT BLK : FLAGS     BLK : XFORTH    COM
B: -READ     ME- : ASSEMBLE BLK : VMOVE    BLK : FORTH     BLK
B>
```

```
A>
A>ytype re @read.me@

                        xForth 2 source code disc.

        For your convenience, this disc contains a truncated copy of
        FORTH.BLK and a version of XFORTH.COM that is pre-configured (to
        a DEC VT-52 terminal). This means the disc contains all the
        files needed to run BUILD.BLK and build a system that is ready
        for SYSGEN.

        To build a new kernel system in the file KERNEL.COM, simply load
        the file BUILD.BLK and answer Y to the questions about parameter
        settings. (If you answer N to the question about DEBUG xForth
        will go away for some time, with only minimal comments, before
        it produces the new system.) Not all the files needed for the
        various options are supplied; in particular, you must always
        leave 68k and f83 set to FALSE. Non-Torch users must leave
        Torch set to FALSE and Torch users must set it to TRUE.

        Note that the .TXT files are loaded (slowly) by the standard
        LOAD-FILE. The sequential i/o package's 'load' word is much
        faster than LOAD-FILE for .TXT files but it insists that (:
        constructs all be on one line which would mean the files had to
        edited. You could convert the .TXT files back to .BLK files
        (adding --> where needed) if required, using the 'unascii'
        utility in the sequential-i/o package.

A>d
Extended Directory version 3.5

@READ    .ME@    2k
ASSEMBLE.BLK     9k
BUILD    .BLK    3k
CPM@DISC.BLK    10k
CPM@EQUT.BLK     1k
CPM@I@O .BLK     1k
D        .COM    3k
FLAGS    .BLK    3k
FORTH    .BLK    8k
METASYS  .TXT   10k
SYSEQUTS.BLK     1k
VMOVE    .BLK    2k
XFCODE80.TXT    24k
XFHILEV  .TXT   28k
XFORTH   .COM   20k
XFTASK   .TXT    7k
XFUPPER  .BLK   15k

Disk A:  1K blocks
Size=  185K, 17 Files, Used=  149K, Space=   36K
```

Additional notes.

SYSGEN

xForth 2 is supplied only as the file XFORTH.COM so if you want to generate a private system by modifying block 1 and calling SYSGEN, you have to remove all the words defined since the delivered block 1 was loaded. To do so, proceed as follows:

```
XPROMPT REPLACED-BY CR
XSIGNON REPLACED-BY (SIGNON)
FENCE-BELOW COLD                    EMPTY
PROTECT
SYSGEN
```

All this does is make sure the main execution variables are set to something safe then remove everything that needs to be removed. If you've re-vectored XKEY or anything else, you'll need to make sure they are safe too.

Changing KEY so it doesn't scan the keyboard.

The manual mentions (in the chapter "Trouble") a way of making xForth run under multitasking systems without competing for the keyboard. Note that this will virtually destroy xForth multi-tasking in normal compile and edit use, but may still be satisfactory for applications where you put in PAUSE wherever necessary. Here is how to do it. The solution presented here means LAST-KEY will no longer work. If you don't want to have the assembler permanently resident you can convert the following to HEX and comma it in.

```
LABEL BIOS          1 LHLD, D DAD, PCHL,

CODE (KEY)          B PUSH, 6 D LXI, BIOS CALL,
                    A L MOV, 0 H MVI, B POP, HPUSH
END-CODE

XKEY REPLACED-BY (KEY)              PROTECT .SIZE BYE

OK>SAVE ?? XFORTH.COM
```

Retain for interest.
REF
XFORTH 1

Getting started.

To run xForth you need an 8080, 8085 or Z80 microprocessor with at least 28K of memory, running the CP/M operating system. You really need at least 32K to do anything worthwhile. A VDU with cursor positioning ability is a great help but is not absolutely essential. Note that versions for 8080 and Z80 processors are different, and versions for CP/M1.4 and CP/M2.2 are different. If you have one of the so-called 'CP/M compatible' operating systems, you will probably find no problems but we don't guarantee it - there's no such thing as 100% compatibility with a system like CP/M that isn't properly defined anywhere except by its own source code.

Before you do anything else, copy the whole disc and put the original away in a safe place. Now fill out and return your licence agreement if you haven't done so already.

The disc you have copied onto will be your initial working disc. Use CP/M's program SYSGEN (or COPY with appropriate options, on some systems) to put a copy of your operating system on this disc. If you have been supplied with both 8080 and Z80 versions of xForth, you can make extra space by deleting the files you don't need (titles containing 8080 or Z80). You are ready to start.

Put the working disc in drive A and type

A>28KZ80 or A>28K8080

to load up a basic system that will run under as little as 28K CP/M (i.e. xForth takes 20K and CP/M takes 8K). The A> is typed by the CP/M system.

xForth will then sign on. Type a carriage return and you will see

```
ok
Stack empty
```

Now type

CONFIG

followed by a carriage return, and xForth will conduct a question and answer session with you to set itself up for your

terminal and your number of disc drives. You might like to know that most terminals have 80 columns and 24 rows, and do wrap long lines (i.e. lines longer than 80 characters spill over to the next line rather than having the rightmost part lost altogether).

To allow the screen editor to work, xForth needs to know how to postion the cursor. If you have any of a number of common cursor addressable terminals such as ADM3, Superbrain or Z19, the code is already written and you will be able to choose the right code from a menu. If you don't have a terminal that appears on the configuration menu or is compatible with one of the terminals appearing there, you'll have to write a little cursor handler as explained in the Appendix 'Altering your system'. However, you can use everything except the screen editor right away, so don't bother about this until you've got used to the basic system.

After the question and answer session, xForth will adjust itself to be able to use as much memory as possible in your system. Notice that after adapting itself to your system, xForth tells you how many CP/M pages it occupies. Type BYE and then, when you are back in CP/M, type

```
A>SAVE 65 XFORTH.COM
```

where 65 is replaced by however many pages xForth just told you. From now on, you merely need to type

```
A>XFORTH
```

to get going, unless you change your CP/M system size, in which case you should repeat the above steps.

Chapter 1

Updates in xForth 2

Don't be put off by the length of this document; much of it is a glossary containing a detailed description. To get going you probably only need to know about the following:

- Multitasking;
- New file system facilities (mainly user numbers and DIR);
- New character input/output facilities (mainly enhanced input/output streams);
- Miscellaneous new features;
- Minor changes needed to assembler and floating point pack due to change to direct threaded code, and to debug and turtle graphics packs due to changes to i/o system. Also, the decompiler no longer works.

The following is a summary of the most important features. The detailed glossary should be consulted in cases of doubt, and for precise definitions.

1.1 Multitasking

This is described in detail in the next chapter. Note here that the multitasking demonstration files may need to be set up for your system before you try to load them.

1.2 New file system facilities

The filing system now knows about CP/M2 CP/M3 and CPN user areas. If a user number is given in square brackets at the end of a file name that user number will be used for the file,

e.g.

```
LOAD-FILE random.txt[0]
```

This is particularly useful for users with high capacity discs. If no user number is given the currently set operating system default user number is assumed. The new words setuser and getuser have been introduced to allow control over this.

DIR now reads an ambiguous file specification with optional user number from the input stream, and lists on the currently selected output streams all files in the stated user area that match the specification. Note that no stack argument is used as in xForth 1. If the file specification is empty it is assumed to be *.* so that typing DIR and pressing return has the same effect as it has in the operating system. Some examples of calls are:

```
DIR
DIR *.BLK
DIR f?g*.x[5]
DIR c:[2]
DIR b:
DIR [15]
DIR a:b?d.*[11]
```

The word $DIR takes a string argument and operates on it in the same way.

LOAD-FILE now allows ordinary text files (prepared with any standard editor) to be read as if they were being typed from the keyboard. This is done by selecting input stream 2 which is connected to the file in question. Loads may be nested and files of type .BLK, which are treated as usual, may load and be loaded by text files. The sequential i/o package provides utilities to convert block files to and from ASCII files.

Since the word WHERE is no longer so useful with text files, a variable `echo´ is provided. If it is set to TRUE, as in `echo on´, each line of the file being loaded is reflected to all currently selected outputs before being interpreted. Note that the usual rules about (and { having to have matching) and } on the same line in console input are relaxed for text files, but that defining words such as : and VARIABLE must be followed on the same line by the word they are defining.

1.3 New input/output facilities.

The input-output facilities are much shinier than before. They are now vectored in such a way that there may be 4 distinct input streams and 4 distinct output streams, and any task may select any of them.

The method is to use bits set in the user variable OUTPUTS (which all tasks have a private copy of) to select output streams, and bits set in the analogous user variable INPUTS to select input streams. Conventionally, the lowest order bits are the normal output and input so that setting INPUTS and OUTPUTS both to 1 gives normal operation. The next higher order bit in OUTPUTS corresponds to the printer and is toggled when control/P is typed. Thus to select the printer alone from within a program, set OUTPUTS to 2; to select both the printer and the vdu set it to 3, since EMIT sends its character to every stream that has its bit set. (Note, however, that KEY only asks for a character from the input stream with the lowest order bit set.)

The next bit, corresponding to setting OUTPUTS or INPUTS to 4, is reserved for file stream i/o though only a (slowish) version for input is provided in the standard system. It is used by LOAD-FILE to load ASCII files containing xForth programs.

The next bits, corresponding to setting OUTPUTS or INPUTS to 8, are free for your own use.

The actual streams are defined by the execution vectors XEMIT and XKEY which have room for 4 codes each instead of the 1 each in xForth 1. On delivery, the codes for XEMIT are EMITT, EMITP, DROP and DROP. On delivery, the codes for XKEY are (KEY), EOF, getc and EOF where getc reads from a file. The intention is that you should always use EMIT in your programs, altering OUTPUTS and INPUTS to get the desired effect, and leaving use of EMITT and EMITP and so on to system words.

Finally, ^EMIT now interprets ^M as a call to CR, interprets ^L as a call to PAGE, ignores ^J, and expands tabs to a multiple of the constant tabsize which is normally 8 but may be changed on configuration. (5 is a good value for programming.)

1.4 Miscellaneous changes.

xForth 2 is somewhat faster than xForth 1 since it uses direct threaded code instead of indirect threaded code. This change was made because it paves the way for a code optimizer in future. The overall improvement in speed is about 15%: test it on SIEVE and on QUEENS. By the way, your disc includes a program NEWSIEVE which was written to do the Byte SIEVE benchmark properly in FORTH and so give a fairer comparison between Forth and other languages. It takes about 43 seconds for 10 iterations in xForth 2 on a 4MHz Z80 machine.

Note that a few changes are needed as a result of the move to direct threaded code. They are described in the next section.

xForth 2 now sizes memory automatically so there is no need for the 28KZ80 followed by CONFIG followed by SYSADAPT operation formerly needed. Just type XFORTH. You still need to run CONFIG to set up for your terminal.

xForth 2 is about 2600 bytes larger than xForth 1.21, though because of the way free space is now measured (see DICTLIM) the amount shown on signon may not appear to agree with this.

Three new words have been added: `on´ `off´ and `0!´. They set a variable whose address is on the stack to TRUE, FALSE and 0. For example, you can say `echo on´ and `echo off´.

A number of other new words have been added, some of them from Forth 83. One example is >BODY which converts a code field address to a parameter field address, i.e. it is the opposite of CFA. Make sure that anywhere you were using 2+ to do this job you now use >BODY since the size of the code field is no longer 2 and is not guaranteed to have any fixed size.

Two other new words from Forth 83 are SPAN and #TIB which hold the number of characters read by the last call to EXPECT, and the number of characters in the text input buffer.

One new word, not from Forth 83, is +LIST which lists from a block number until the end of file is encountered.

A number of words have been removed: an example is (LOOP) which is merely the code compiled by LOOP. It is hoped that those removed will not be missed: actually, we were glad to

see the back of them. They are summarized at the beginning of the glossary update.

There are minor changes to the screen editor: for example, you get a chance to change your mind if you hit the abandon key. The word `count´ has been shifted into the (EDITOR) vocabulary.

USER is slightly changed: it no longer uses a stack argument to tell it where to put its variable relative to the bottom of the USER variable area. Instead it keeps track of how many user variables there are in the constant #UVARS.

1.5 Changes needed to xForth optional packages

The following changes need to be made to packages you have already purchased from us. They are mainly concerned with production of CODE sections and result from the change to direct threaded code which has changed the header structure of xForth words. If you have written any CODE words yourself they will only need to be recompiled unless you have made use of special knowledge of the structure of word headers.

There are also a couple of changes to the DEBUG pack which are concerned with deletion of a definition and with the new input output facilities.

If you have bought packages with or since the upgrade to version 2 the changes will have been made for you.

1.5.1 ASSEMBLE.BLK

In line 10 of block 3 of the assembler, the definition of CODE should now be

```
: CODE  ?EXEC   CREATE SMUDGE   -3 ALLOT
        [COMPILE] ASSEMBLER  !CSP ;      IMMEDIATE
```

i.e. instead of `HERE DELTA - -2 ALLOT ,´ we simply have `-3 ALLOT´.

1.5.2 FP.BLK

In line 11 of the first block of the floating point pack, we now need

```
CREATE  fppack  HERE 4 +  -3 ALLOT  HEX 0C3 C, ,
DECIMAL
```

which puts a jump to HERE+4 in to the code field instead of merely comma-ing in the address of HERE+4.

In line 1 of block 2 of the floating point pack, replace `FIND fppack @´ with `FIND fppack 1+ @´ to compensate for the above change.

1.5.3 TRIGS.BLK

Owners of the turtle graphics package should remove the code definition for `U2/´ in the file TRIGS.BLK and replace all occurrences of it with `2/´.

1.5.4 DEBUG.BLK

The debug pack needs to have a couple of changes. Since the lethal word LIT is no longer present, line 12 of the first block of the debug pack, which redefines it to be safer, should be deleted. Also, because of the new input/output facilities the definition of `wait´ on block 4 of debug.blk doesn't work. Delete the definition of (last-key) and replace the definition of `wait´ with

```
: wait  KEY   DUP INTRPT-KEY @ =
        IF  XINTRPT @ EXECUTE  ENDIF ;
```

You can also check for hitting control/C if you like. The normal interrupt facilities during output are still there; all that this affects is the operation when pausing is turned on.

1.5.5 NARRATE.BLK from Turtle graphics pack.

A tricksy Forth definition was supplied with the turtle graphics demo, to let selected parts of Forth blocks be echoed while loading. Because of the changes to input/output facilities, a slightly different set of dirty tricks is needed. Replace the file NARRATE.BLK from your original turtle demo with the new one on the upgrade disc. Ignore the revolting style and layout of this file!

Chapter 2

Multi-tasking in xForth.

In computer jargon, "multitasking" refers to a situation where a computer appears to be doing several different and possibly unrelated things ("tasks" or "processes") at the same time. Most computers, and nearly all microcomputers, do not really do several things at once, but chop up all of their tasks into bits and do a bit of one, then a bit of the next, and so on. This requires great care if the tasks want to communicate with one another successfully.

xForth multitasking facilities are more powerful than those we know of in any other Forth. For most purposes they are easy to use; the instructions that follow describe them in general terms, and the example files and the update glossary give more details.

An obvious use for multitasking is to let slow peripherals like printers be driven by special background tasks instead of holding up normal use of xForth. This can be done either by providing a print buffer or by making sure that any tasks which talk to the printer are in the background. Examples of both are supplied: for the second case, a word PRINT-FILE is defined which passes a file over to a background task for printing and does not need a large print buffer.

Some of the other uses for multitasking range from simply showing a clock or calendar in a fixed part of the vdu screen, through displaying the current values of memory locations as a check during debugging, to timed control and data collection tasks. A sample windowing program is provided so you can see different tasks working at once. The only limitation is your imagination!

2.1 Simple multitasking.

A "task" in xForth is a special sort of definition that looks like a colon definition except that it starts with `TASK:´ instead with `:´. When first defined it does nothing,

but once it is started, it takes over control of the computer and retains control until it runs to completion, or it temporarily hands over control to another task, or it gets blocked until some event happens. What this means is that tasks in xForth cooperate with one another, by not deliberately hanging on to the computer for long periods of time.

In the first case, where the task runs to completion, control then passes to the next task in a queue which is managed in such a way as to give all tasks a fair chance. The same happens in the second case, except that the task is reinserted at the tail of the queue. When its turn comes it will continue where it left off. The third case, waiting for an event, occurs when the task asks to be held up until some particular time, or until it receives a signal from another task saying, perhaps, that there is some data waiting to be processed.

Here is an example of a task that sounds an alarm if the value of a variable gets outside a certain range. This sort of thing is useful for debugging, as you can run the watchdog task then start up whatever you are debugging. This will work as long as the word being debugged gives up control from time to time. As was mentioned above, this is only one of many uses for multitasking.

```
VARIABLE var                -- Assumed to be used by some other task.
0 CONSTANT minvar           9998 CONSTANT maxvar

TASK: checkvar  BEGIN   var @  minvar maxvar in-range?
                WHILE   PAUSE
                REPEAT
                BELL  2 CRS  ." var = " var ?  2 CRS ;
```

As you can see, this is just like a colon definition except that it starts with `TASK:´ instead with `:´. Nothing happens when you first define it but once you start it, with

checkvar START

it loops around, checking the value of `var´ on each pass. If the value is ok, checkvar calls the special word `PAUSE´ which says "pass control to the next task that's waiting to go, but put me back in the queue so I get a chance to run again". When it next runs it will restart where it left off, with the stacks and so on all intact, so it will meet REPEAT which sends it back to BEGIN and another time around. If the value is bad, it sounds the alarm and then stops, because you presumably want a chance to put things right. In this case the tasks stopped because it reached the end of its definition but it could also have called QUIT from within the definition.

Now try typing in the above example and starting the task, and then define

```
: changevar   BEGIN  10001 RANDOM var !  PAUSE  ?TERMINAL UNTIL ;
```

and run `changevar´. You can stop it by hitting any key, but if you don't then eventually it will set `var´ to 9999 or 10000 and `checkvar´ will complain. The use of PAUSE in this example makes sure the user task (the one you're talking to normally) gives other tasks their chance to run. It would also have done so without special action on your part if it had called KEY which has a call to PAUSE in its definition. In fact, a possible definition of the keyboard reading part of KEY is

```
: (KEY)  BEGIN ?TERMINAL NOT WHILE  PAUSE  REPEAT  LAST-KEY ;
```

Notice how the so-called `busy waiting´ loop

```
: (KEY)  BEGIN ?TERMINAL UNTIL  LAST-KEY ;
```

has been changed to give other tasks a chance if the present task is waiting for input.

If that were all you had available to control when a task runs, you could still do quite a lot. Many large data monitoring and control applications have been written in this way, including, I understand, American Airlines' baggage handler, speed and depth monitoring on Missippi tugboat trains, and many others. However, if two tasks have to cooperate they must be written very carefully. The xForth task handler allows advanced control via delays, semaphores, counting semaphores, demons and monitors. Before studying them we have to think a little about the structure of a task.

2.2 User variables and private stacks.

A task needs to be able to work largely independently of other tasks. To do so it needs its own stack and return stack (although so-called multitasking systems exist which try to do without this). It also needs some private variables so that it can, for example, change OUTPUTS to send its output somewhere without messing up other tasks.

Such variables are traditionally called USER variables in Forth. They are provided in xForth for each task, with a complete set being available for normal tasks and a limited set for small "background" tasks. When you type the name of a task it leaves the base of its user variable area on the stack

and this is used by other words to manipulate the task.

When you define a task xForth sets aside an area of memory for user variables, with the stacks and, for normal tasks, a terminal input buffer and a small private dictionary area. This is initialized when a task starts up and is accessed only by the task itself. Tasks communicate via ordinary VARIABLEs or via semaphores as we will see later.

2.3 Delays

Delays are easiest to explain. Suppose you want a task that just does one thing at a fixed future time: maybe it opens a control valve, puts an appointment reminder on your vdu, or something quite different that you can no doubt think of yourself. You need a clock. If your computer has a built-in clock you will find out later how to tell xForth about it, but for now just load the file DELAY.BLK which simulates a clock by having a little task that just increments a variable every time it has its chance to run. (Torch users can load the file TDELAY.BLK which uses the BBC micro's clock correctly.) Now type

```
5 seconds DELAYFOR
```

and xForth will go dead for a while (which is unlikely to be very near 5 seconds unless you have a Torch or your computer is very similar to ours). What has happened is that the user task - the one that talks to you and listens to your answer - was put on a delay list with a tag saying when it was to be restarted, and a clock monitor checked the list whenever its turn came to run; when the pseudo-timer said the time was 5 seconds later than it was when you first typed DELAYFOR, the user task was allowed to continue.

The reason for having a special clock monitor is that there may be many delayed tasks, but they do not all need to keep coming back to check the clock since the special clock monitor task can do so more efficiently. Indeed, it may be possible on some systems to make the clock monitor interrupt-driven, so it doesn't consume any time at all unless a task is due to be run.

A more realistic use of DELAYFOR (and its friend DELAYUNTIL, which waits until the clock shows a certain time) is in a task such as

```
TASK: readdata
        19 hours DELAYUNTIL
```

```
initialize
BEGIN   meter1 read  meter2 read
        5 minutes  20 seconds  D+  DELAYFOR
AGAIN ;
```

The words `hours', `minutes' and `seconds' all take a single precision integer from the stack and scale it to a double precision number which is the correct number of clock ticks for your system. To combine them we use D+ as in the last example. Both DELAYUNTIL and DELAYFOR expect a double precision number of clock ticks on the stack, and you can generate it any way you like, not just with `seconds' and so on.

2.4 Semaphores, or how tasks communicate.

Tasks communicate to some extent via public (non-USER) variables, but there are certain types of communication that are so common and so important that it is a good idea to provide special facilities. These facilities are concerned with event control.

A good way to think about the way the delay words work is to say that words that call them are giving up control of the computer until a certain event occurs: in this case, a certain time shows on the clock. Other sorts of events can be handled too, via "semaphores" which are indicator lights that are red when an event hasn't happened or a facility is in use by someone else, and green when the event has happened or the facility is available for use.

2.4.1 Two state semaphores, and demons.

For example, suppose several tasks are sending output to the printer. We don't want their output to get tangled up so we make them obey the semaphores like traffic lights or railway signals. They use the word WAIT to look at the signal and hold off if required. We define a semaphore:

```
SEMAPHORE printer
```

which is an object with a variable and a queue. If the variable is nonzero, (typically 1) the light is green and we can go on:

```
printer WAIT
```

will decrement the variable and do nothing else. But assuming

the variable was 1, if some other task calls `printer WAIT' then WAIT discovers the signal is red and suspends execution of the task, adding it to the tail of the queue managed by WAIT.

When our first task is finished with the printer, it tells the world by saying

```
printer AVAILABLE
```

which removes the first task from the queue (leaving the light at red to hold up the next one) and makes it ready to continue execution where it left off. If there were no tasks waiting, AVAILABLE would have changed the light to green. In either case, AVAILABLE doesn't pass on control: if you want this to happen, call PAUSE afterwards.

By cunning use of semaphores you can have demons: programs which lurk unseen until some event occurs, when they jump onto the stage and cast their spells. The clock monitor and the print buffer manager in the example files are both implemented as demons: they don't appear unless there's work for them to do. This is much better than constantly cycling round, calling PAUSE when there's nothing interesting to do.

2.4.2 Multi-state semaphores.

Sometimes it is useful to have values other than red and green for the variable, giving so-called counting semaphores. SEMAPHORE and WAIT work as happily with counting semaphores but AVAILABLE has to be replaced by SIGNAL. For most uses, AVAILABLE is the one to use.

2.4.3 Buffers.

One of the supplied facilities using semaphores is a set of buffer management routines. To define a buffer of size 100 you say

```
100 BUFFER: foo
```

and this defines a data object with lots of associated semaphores, counters etc. Buffers are first-in, first-out devices, unlike stack which are last-in, first out devices. Buffers may only be used via their monitor words `deposit' and `fetch', which work in the obvious way:

```
foo fetch EMIT
```

will fetch and display a character from buffer foo if there is one available, while if the buffer is empty the task will be

put on a queue managed by a semaphore in the buffer. (As you may have guessed, fetch does a call to WAIT.) Similarly,

```
ASCII Q  foo deposit
```

will put a Q in the buffer foo if there is room, and otherwise wait until the buffer is signalled as being nonfull.

The files DELAY.BLK, BUFFER.BLK, SPOOL.BLK and WINDOWS.BLK have examples of the use of fetch and deposit and of WAIT and AVAILABLE.

2.5 The example files.

2.5.1 DELAY.BLK

Look first at the file DELAY.BLK. This has a pseudo-clock made from a background task that just increments a double precision variable every time it runs. The function `time´ returns the value of that variable. If your computer has a clock or timer that you can access, redefine `time´ to return the number of clock ticks as a 32 bit count. For example, TDELAY.BLK does this by calls to OSWORD which is an operating system call.

Now redefine `seconds´, `minutes´ and `hours´ to convert a stack quantity to the requisite number of clock ticks. For the BBC micro, the ticks are every centisecond so the definitions are

```
: seconds  100 U* ;
```

and so on.

This file also has a clock monitor which is a demon that stays out of things when no tasks are waiting but stays active all the time, checking the clock, when any are waiting. This means only one task is checking the clock, which is better than everybody having to keep looking at it. The monitor uses some knowledge of the internals of semaphores to compensate for the fact that the delay queue isn't an ordinary first-in first-out queue. We suggest you look at the other files before trying to work out what it does; and we don't recommend you copy this procedure yourself since other sorts of queues are best managed in the standard ways. The clock monitor, like the print demon defined later, is a background task defined via BTASK: since it needs few (in this case, no) input/output facilities. Background tasks use less memory than foreground tasks.

2.5.2 BUFFER.BLK

This file implements a print buffer. You need to do a little installation as explained at the end of this section.

The buffer works by setting aside an area (1000 bytes as delivered) for character storage when the printer isn't ready. There is a small task whose only job is to fetch from the buffer and output to the printer using EMITP. It is a background task since it doesn't use the dictionary or PAD areas at all. It writes to the printer using EMITP and no other task should use EMITP. Once it has been defined, we can re-vector the printer output in XEMIT+2 to deposit a character in the buffer and let the buffer task handle the actual output. Now whenever OUTPUTS has bit 1 set (e.g. `2 OUTPUTS !´) output will go via the buffer. This will be true for all tasks that use EMIT or any standard xForth output, which always uses EMIT.

This file also defines the semaphore `printer´ which - as long as everyone trying to print uses it - keeps output from different tasks separate. Tasks wanting to use the printer should do

```
printer WAIT
```

before trying to output and

```
printer AVAILABLE
```

when done. The control/P toggle for the slaving the printer to the main task output does not consult this semaphore: don't use it when the print monitor is loaded. The reason it doesn't work is that we thought it would be too risky to have your system hanging because you accidentally hit control/P and another task had failed to reset the semaphore.

Before loading BUFFER.BLK you need to edit it to tell xForth how to read your printer status. This is often, but not always, possible by a BIOS call, and you may want to implement this yourself. Another common way is by a direct port read; an example is shown in the file for our North Star Horizon. The idea is to return a TRUE flag if the printer is busy. (If you are completely stuck you can leave the definition as it is in the file, returning FALSE always so the buffer printer always tries to output and so will often hang uselessly.) The Torch has a special OSBYTE call that is implemented in TBUFFER.BLK.

Depending on what you manage to do about the printer busy flag, on your printer speed, and on other characteristics of

your system, you may find it helpful to tune the printer buffer performance by trying different combinations of the presence or absence of PAUSE in the words (EMITT) and (EMITP) defined in the file. Don't be afraid to experiment.

It is possible, but less often useful, to define keyboard and VDU buffers. Samples are given in the file; you need to provide a status word (analogous to ?PRBUSY) for your vdu, and insert a `-->´ continuation word at the end of the last print buffer definition block so that the other buffers get loaded too.

2.5.3 SPOOL.BLK

This file implements a background file listing spooler. A word PRINT-FILE reads a file name and assigns it to a temporary file in order to check it exists. If all is well it puts the name in a buffer which is monitored by a demon whose job is to wait for file names then print them out. This can work whether or not you have installed a print buffer. The point is that when you say `PRINT-FILE abc.def´ you get control immediately and you can send another file to the printer in the same way, and so on until the file name buffer fills up. The supplied version allows at least 5 files at a time; more if they have short names.

2.5.4 WINDOWS.BLK and TASKDEMO.BLK

Full windowing requires a memory mapped display. Some VDUs such as the BBC micro allow hardware defined windows for both text and graphics, and the file WINDOWS.BLK shows how to take advantage of these. If you have more powerful facilities, you should find it easy to modify the code.[1] If all you have is a standard VDU you can still manage if you write software to take care that all output stays within its window.

The method is to define a new set of USER variables that specify text and graphics window edges, and graphics cursor position. (The text cursor position is kept in >LINE and OUT which is consistent with normal xForth usage). Then we define words `make´, `open´ and `close´ which create such windows and

---

1. For example, if you can read the contents of a window you can handle overlapping windows by arranging that when a window is opened you store the existing contents somewhere, and restore them when the window is closed. Each task will need its own temporary storage area, big enough to hold a complete copy of the window. This can even be done with the BBC micro and Torch.

control access to the screen. For example

```
" Headlines" twindow make
```

makes a text window with a label "Headlines" at the top left. And

```
" Brownian motion" gwindow make
```

makes a graphics window with a suitable label. In each case the window edges are given by the values of the variables at the time make is called: for example, the left edge of a task's graphics window is given by xl.

To use a window, you have to say

```
twindow open
```

before sending output, which checks a semaphore to see if the screen is available, then sets up the windows. After writing to it for a while, say

```
twindow close
```

to restore the default text area and flag the screen as available to some other task. It would be reasonable to do this after every line or two of text. With graphics, how often you close the window to give others a chance depends on the application.

Torch users can load the file TASKDDEMO.BLK to get a running demonstration with 2 graphics windows, 2 text windows and a reserved window at the bottom of the screen where normal editing, compilation and so on can take place. Try typing DIR or VLIST and see what happens. When you pause the output using control/S the other tasks will go on while the display is help up. As well as the tasks owning the windows, there are several other tasks like the clock monitor running, and you should still find it feasible to load the print buffer and print spooler and use them. To get a four colour display we use mode 1 which makes editing a bit painful because the screen width is too small, but if you use mode 0 instead you will find you can edit quite happily while the demonstration is going on. It may help if you modify the `log´ task to be less greedy in its use of the window. At present it may print out several lines before giving anyone else a chance to run.

2.6 Summary: task types and states.

2.6.1 Task types

A task may be one of two types: a normal task or a background task. A normal task can call all xForth words, though unless it is the normal user task it usually only has a tiny private dictionary so it should not attempt to compile anything. A background task takes up less room than a normal task but has no terminal input buffer, has smaller stacks, and has no PAD and only a few user variables. Nevertheless, as long as it does no file name operations, or calls to WORD or number formatting words or most string words, it can do a great deal. For information about how to change stack sizes etc of tasks, see `taskframe´ and `btaskframe´ in the glossary.

2.6.2 Task states

A task may be in any of 3 states: stopped, active and waiting.

Tasks are stopped when first created or when they have attempted to run after STOP has been applied to them (see later). In particular, a task goes into the stopped state if it calls QUIT or if it runs to completion.

Active tasks may be running or ready to run. The running task is the one that has control at any particular moment and the address of its user variable area is returned by the word UP@. Tasks that are ready to run get their chance when PAUSE is called, either explicitly or implicitly as in KEY. They also get their chance to run if a running task gets moved to a queue by WAIT or by one of the DELAY words.

Waiting tasks may be waiting in a semaphore queue, where they were put by WAIT and whence they may be removed by SIGNAL or by AVAILABLE, or in the clock queue where they were put by a DELAY word and whence they may be removed by the clockmonitor.

2.7 General task management

To start a task that is in the stopped state, use START. To stop one that is active or waiting use STOP. Thus

```
task1 START      task2 STOP
```

can be called from the main user task. A STOP does not take effect until the next time a task tries to run; it is equivalent to inserting QUIT in place of the instruction the task is next due to execute. As we saw before, a task will always STOP itself if it calls QUIT or if it reaches the end of its definition.

To output the name of a task, use .TASK as in

```
task1 .TASK
```

which would merely output `task1´. If you have loaded the file DELAY.BLK you can output the names of all active tasks by using .TASKS which takes no arguments, and you can output the names of all tasks that are waiting for the clock by using .DELAYED which also takes no arguments.

When tasks have been defined, you have to be rather careful about uses of FORGET or EMPTY. You must make sure all tasks that will be removed from the dictionary are stopped, by calling STOP for them then signalling any semaphores they are waiting on. (For example, it is ok to type `delayed AVAI¥AB¥E´ as often as required to flush out the delayed queue.) Alternatively you can reinitialize all semaphores by repeating

```
0 0 sema 2!
```

for every semaphore, replacing `sema´ by the appropriate name. In that case any tasks that were waiting on them are lost unless restarted with START.

If you call COLD, there is an immediate call to EMPTY and then all tasks are killed on the spot. The main user task will restart after the code in XSIGNON has been executed. If you want to make xForth come up running several tasks either on initial startup or after COLD is typed, you can vector XSIGNON to a word that starts all necessary tasks. Don't forget to run PROTECT before you save the modified system.

Chapter 3

Trouble

Obviously we want to know about any bugs you discover, but our experience, believe it or not, is that most problems are caused by the fact that xForth reaches the parts of your operating system that most programs don't.

If you have trouble getting xForth to run at all, there is possibly a bug in the BIOS of your CP/M. We have been amazed at how many well-known and respected systems have bugs in their versions of CP/M. The commonest are as follows:

- Failure to produce output on vdu, or crash after disc access e.g. Zorba.

  * The most likely problem is that the BIOS uses some Z80 additional registers and fails to restore them. Use the 8080/CPM1.4 version of xForth.

- VDU output and terminal input OK but can't handle disc files, e.g. listing a file you know to exist gives long strings of ^@ (i.e. nulls), e.g. early ALTOS.

  * The random access disc routines are not properly implemented. Use the 8080/CPM1.4 version of xForth.

- Cursor fails to appear, but input and output are ok except that you don't know where you are on screen, e.g. Sharp with Xtal CPM, early Gemini.

  * The cursor is flashed in software and is not interrupt driven. The only effective solution is to insist that your supplier give you a bug-free version of CPM. In the meantime, put up with the lack of cursor or else write your own word to read keys (using CPM-CALL for function 6, for example) and put it in XKEY. You will lose most multitasking abilities. It might be worth trying to build a flashing cursor with the xForth delay words: some character appears for so many clock ticks, then is removed, then is replaced and so on.

- xForth greatly slows down a system running MP/M or other multitasking DOS.

    * What can you expect when two multitasking systems compete for resources! You can re-vector KEY as above, and give up most multitasking, or you can persuade other users to use xForth instead!

XFORTH 2 UPGRADE!

Appendix A

Alphabetic list of words in xForth

A.1 Words deleted.

.COVOC .CUVOC ;S (DO) (LOOP) (+LOOP) (OF) BLISTS FILE-INIT INDEX LISTS LIT PRINTER-ON? SEC/BLK TRIAD XCANCEL

Also, 0BRANCH has been renamed ?BRANCH and CPM-CALL and CPM-CALLb have been renamed DOS-CALL and DOS-CALLb

A.2 Words unchanged.

!	!CSP	"	#
#->$	#>	#BUFF	#files
#S	$!	$+	$->$
$<	$=	$FIND	'
's-FCB	's-STATUS-BYTE	'th-FILE	(
(ASCII)	(EDITOR)	(file-voc)	(FIND)
(FLUSH)	(ID)	(LINE)	(skip-until)
(skip-while)	([,]VARIABLE)	([]VARIABLE)	*
*/	*/MOD	+	+!
+-	+LOOP	,	-
-->	-1	-TRAILING	.
."	.CPU	.LINE	.R
.SIZE	.STACK	.VERSION	/
/MOD	0	0<	0=
0>	1	1+	1+!
1-	1-!	2	2!
2*	2+	2-	2@
2CONSTANT	2DROP	2DUP	2OVER
2ROT	2SWAP	2VARIABLE	3
79-STANDARD	:	;	;CODE
<	<#	<=	<>
<CMOVE>	=	>	>=
>IN	>LINE	>R	?
?COMP	?CSP	?DEPTH	?DUP

?ERROR	?EXEC	?LOADING	?PAIRS
?PAUSE	?TERMINAL	@	ABS
AGAIN	ALLOT	AND	ASCII
B/BUF	BASE	BEGIN	BELL
BINARY	BL	BLANKS	BLK
BLOCK	BRANCH	BUFFER	BYE
C!	C#	C,	C/L
C@	CAN-KEY	CASE	CLOSE
close-files	CMOVE	COLD	COMPILE
CONFIG	CONSTANT	CONTEXT	CONVERT
COPIES	COPY	COUNT	CR
CREATE	CRS	CSP	CTRL
CURRENT	D+	D+-	D-
D.	D.R	D0<	D0=
D<	D=	DABS	DECIMAL
DEFAULT	DEFINITIONS	DEL-KEY	DEPTH
DLITERAL	DMAX	DMIN	DNEGATE
DO	DOES>	DOS-CALL	DOS-CALLb
DP	DPL	DROP	DU<
DUP	ELSE	EMITP	EMITT
EMPTY	EMPTY-BUFFERS	ENDCASE	ENDIF
ENDOF	ENSURE-LINE	EOF	ERASE
ERROR	EXECUTE	EXIT	FALSE
fassign	FCREATE	FENCE	FILE
FILL	FIND	FIRST	fname!
FORGET	FORTH	HERE	HEX
HLD	HOLD	I	ID.
IF	IMMEDIATE	in-range?	INSTALL-$$$
INTRPT-CHAR	J	L/S	LATEST
LEAVE	LFA	LIST	LITERAL
LOAD	LOOP	M*	M/
M/MOD	MAX	MESSAGE	MIN
MOD	MOVE	MYSELF	NEGATE
NEXT	NFA	NOOP	NOT
NUMBER	OPEN	OR	OUT
OVER	P!	P@	PAD
PAGE	PFA	PICK	PREV
QUERY	R#	R/W	R0
R>	R@	REPEAT	REPLACED-BY
RESTORE-$$$	REVERSE	ROLL	ROT
RP!	RP@	S->D	S0
SAVE-BUFFERS	SCR	SEE	SEE-FILE
seg-size	SIGN	SMUDGE	SP!
SP@	SPACE	SPACES	STATE
STRING	SWAP	SYSADAPT	SYSFILE
SYSGEN	THEN	TIB	TOGGLE
TRUE	TYPE	U$	U*
U.	U.R	U/MOD	U<
UCHAR	UNTIL	UPDATE	USE
VARIABLE	VLIST	VOC-LINK	VOCABULARY
WARM	WARNING	WHILE	WIDTH
WORD	wrap	XCURSOR	XNUMBER
XOFF-CHAR	XOK	XOR	XPAGE

XPROMPT	XRUBOUT	XSIGNON	[
[,]VARIABLE	[COMPILE]	[]VARIABLE	]
{	|	}	

A.3 Vocabulary listing with definitions of new words.

These are all the words in the vocabulary FORTH in xForth 2. Where they are changed from xForth 1 or new to xForth 2, they are defined here. The usual notation of the xForth technical manual is used: stack pictures are of the form

(stack before --- stack after)

unless text is read from the input in which case they take the form

(stack before +++ stack after).

Stack items are

$	string (address and count),
c	character or byte,
flag	logical,
n	16 bit integer,
u	16 bit unsigned integer,
d	32 bit signed integer,
du	32 bit unsigned integer, and
addr	16 bit address.

!	Unchanged
!CSP	Unchanged
"	Unchanged
#	Unchanged
#->$	Unchanged
#>	Unchanged
#BUFF	Unchanged
#files	Unchanged
#S	Unchanged

#TIB (--- addr) U

A USER variable containing the length of text in the terminal input buffer when it is being used for input.

#UVARS (--- n)

A CONSTANT returning the number of USER variables so far defined.

```
$!              Unchanged
$+              Unchanged
à¾"£            Unchanged

$->U#           ( $ --- du flag )
```

Convert string $ to an unsigned double number and leave the result under a flag which is TRUE if and only if the string consisted entirely of digits in the present base.

```
$<              Unchanged
$=              Unchanged

$break          ( $ c --- $1 $2 flag )          "String break"
```

If character c is contained in string $, return $1 and $2 such that $ is $2c$1 and leave a true flag. If c is not contained in $ then $2 is empty, $1 is identical with $, and the flag is false.

```
$DIR            ( $ --- )
```

Send a directory listing to the current output device corresponding to the ambiguous file and user specification contained in string $. See the entry for DIR.

```
$FIND           Unchanged
'               Unchanged
´s-FCB          Unchanged

´s-name         ( file --- $ )
```

Return the name of the operating system file currently assigned to `file´, in the form A:NAME.EXT[u] where u is a number in the range 0 to 15.

```
´s-STATUS-BYTE  Unchanged
´th-FILE        Unchanged
(               Unchanged
(ASCII)         Unchanged
```

(EDITOR) Unchanged
(file-voc) Unchanged
(FIND) Unchanged
(FLUSH) Unchanged

(get) Internal use; do not use.

(ID) Unchanged

(KEY) (--- c)

The default code for input stream 0. (See INPUTS.) Returns a character typed at the console, calling PAUSE as often as necessary if a character is not ready.

(LINE) Unchanged

(prompt) (---)

The default prompt message: a stack display.

(SIGNON) (---)

The default signon command which displays the text printed after cold start. If the contents of XSIGNON are changed from (SIGNON) to a user defined word then xForth will execute that word on startup.

(skip-until) Unchanged
(skip-while) Unchanged
([,]VARIABLE) Unchanged
([]VARIABLE) Unchanged
* Unchanged
*/ Unchanged
*/MOD Unchanged
+ Unchanged
+! Unchanged
+- Unchanged

+LIST (u ---)

Call LIST for all blocks from block u until a block starting with EOF is read.

+LOOP Unchanged

,	Unchanged
-	Unchanged

-- (+++)

Ignore all text until the end of the current input line being interpreted. Lines are terminated with carriage return (control/M) during input from all streams except when BLK is nonzero, when they are all 64 characters long.

-->	Unchanged
-1	Unchanged

-TEXT (addr1 n1 addr2 --- n2)

Compare two strings over the length n1 beginning at addr1 and addr2. Return zero if the strings are equal. If unequal, return n2, the difference between the last two characters compared.

-TRAILING	Unchanged
.	Unchanged
."	Unchanged

.BASE (---)

Type the current base, in decimal, on the currently selected output stream(s), followed by a space.

.CPU	Unchanged
.LINE	Unchanged
.R	Unchanged
.SIZE	Unchanged
.STACK	Unchanged

.STORE (---)

Type on the standard output an unsigned number which is the number of bytes between the the current dictionary top (HERE) and the value stored in DICTLIM.

.TASK (addr ---)

For the task whose user variable base is at addr, print its dictionary name on the currently selected output stream(s).

.VERSION	Unchanged
/	Unchanged
/MOD	Unchanged
0	Unchanged

0! (addr ---)

Store 0 in the two bytes starting at addr.

0<	Unchanged
0=	Unchanged
0>	Unchanged
1	Unchanged
1+	Unchanged
1+!	Unchanged
1-	Unchanged
1-!	Unchanged

lsttime (--- ud)

Leave on the stack a double unsigned number which is the time the soonest delayed task is due to be restarted. Undefined if there are no delayed tasks.

2	Unchanged
2!	Unchanged
2*	Unchanged
2+	Unchanged
2-	Unchanged

2/ (n1 --- n2)

Replace n1 with its arithmetic right shift, i.e. n1 is shifted right but the highest bit remains the same. Equivalent to floored division by 2, i.e. 3 2/ is 1 but -3 2/ is -2.

2@	Unchanged
2CONSTANT	Unchanged
2DROP	Unchanged
2DUP	Unchanged
2OVER	Unchanged
2ROT	Unchanged
2SWAP	Unchanged

2VARIABLE Unchanged
3 Unchanged

4 Leave 4 on the stack.

79-STANDARD Unchanged
: Unchanged
; Unchanged
;CODE Unchanged (but see comments on direct threaded code)
< Unchanged
<# Unchanged
<= Unchanged
<> Unchanged
<CMOVE> Unchanged

<MARK (--- addr) "backward mark"

A system word extension. Used at the destination of a backward branch. addr is typically only used by <RESOLVE to compile a branch address.

<RESOLVE (addr ---) "backward resolve"

A system word extension. Used at the source of a backward branch after either BRANCH or ?BRANCH. Compiles a branch address using addr as the destination address.

= Unchanged
> Unchanged
>= Unchanged

>BODY (cfa --- pfa)

Replace the execution address (code field address) on the stack with the corresponding body address (parameter field address).

>IN Unchanged
>LINE Unchanged

>MARK (--- addr) "forward mark"

A system word extension. Used at the source of a forward branch. Typically used after either BRANCH or ?BRANCH. Compiles space in the dictionary for a branch address which will later be resolved by >RESOLVE.

>R	Unchanged

>RESOLVE (addr ---) "forward resolve"

Used at the destination of a forward branch. Calculates the branch address (to the current location in the dictionary) using addr and places this branch address in the space left by >MARK.

?	Unchanged
?BRANCH	Same as 0BRANCH in xForth 1.
?COMP	Unchanged
?CSP	Unchanged
?DEPTH	Unchanged

?DICT (---)

Check that the PAD (which lies above HERE) is at least 96 bytes below the value stored in DICTLIM and call ERROR if not.

?DUP	Unchanged
?ERROR	Unchanged
?EXEC	Unchanged
?LOADING	Unchanged
?PAIRS	Unchanged

?PAUSE Unchanged.

But note that if output is suspended using XOFF-CHAR other tasks may still execute.

?STACK (---)

Check that the stack pointer lies between S1 @ and S0 @, and that the return stack pointer lies between R1 @ and R0 @, and call ERROR if not.

?TERMINAL	Unchanged
@	Unchanged

ABORT (---)

Clear the stack and call QUIT. No message is printed.

ABS Unchanged

active (--- addr)

A variable which points to the tail of the circular list of currently active tasks.

advance (addr ---)

Adjust the circular list tail pointer stored in addr so that the current head becomes the tail and the successor of the current head becomes the new head.

AGAIN Unchanged
ALLOT Unchanged
AND Unchanged

append (addr1 addr2 ---)

Add the item at addr1 to the circular linked list whose tail pointer is addr2. Used typically to append the task which has user area base at addr1 to the tail of the circular list whose tail pointer is at addr2.

ASCII Unchanged

AVAILABLE (addr ---)

In conjunction with WAIT, manage a two-state semaphore. That is, assuming addr is the address of a semaphore, check whether there are any tasks queued by the semaphore. If so, remove the first in the queue and add it to the tail of the active list, but do not call PAUSE. If there are no tasks queued, set the semaphore's counter to 1. See WAIT and SEMAPHORE and SIGNAL.

B/BUF Unchanged
BASE Unchanged
BEGIN Unchanged
BELL Unchanged
BINARY Unchanged

BL Unchanged
BLANKS Unchanged
BLK Unchanged

blkdestroy (u ---)

If virtual memory block u is in a buffer, mark the buffer as empty without writing its contents to disc.

BLOCK Unchanged
BRANCH Unchanged

BTASK: (+++)

Define a background task; that is, allocate space for 12 USER variables (the only public ones being S0 S1 R0 R1 XERROR INPUTS OUTPUTS restart-time) and initialize them according to the pattern in btaskframe, but with the values offset correctly. Then perform various pre-initialization tasks, and finally switch to compilation mode. If `;´ is executed successfully the result is to define a background task that is initially in the stopped state. A background task has only the user variables indicated and in particular has no dictionary or PAD so it may not call WORD . .R #->$ $+ LOAD-FILE DIR or any other number formatting word or string handling word (or any other word) that uses the PAD.

btaskframe (--- addr)

The address of a 4 byte region which is used to determine the size of the stacks in a background task. The values are the size of the stack and the size of the return stack in that order. On delivery the stacks are both 40 (decimal) bytes long.

BUFFER Unchanged

BUFFER: (n +++)

Define a buffer of size n, where n lies between 1 and 10,000. This buffer is only to be accessd via the monitors `deposit´ and `fetch´. Example:

1000 BUFFER: printbuffer

When `printbuffer´ is executed it leaves its address on the stack. See also `buffinit´ which is called on buffer creation

but may be called, with care, at other times.

buffinit (addr ---)

Initialize the buffer at `addr´ so that it is empty, has no waiting tasks, and is ready for use.

BYE Unchanged
C! Unchanged
C£ Unchanged
C, Unchanged
C/L Unchanged
C@ Unchanged
CAN-KEY Unchanged
CASE Unchanged

CFA (pfa --- cfa)

Return the execution address (code field address) corresponding to the body address (parameter field address) on the stack. Not necessarily equivalent to 2- as it was in xForth 1.

CFA->NFA (cfa --- nfa)

Return the code field address corresponding to the name field address.

ch-in-str? (c $ --- n)

If character c is to be found in string $, return its position in the string (with the first character treated as 1). If not, return 0.

CLOSE Unchanged
close-files Unchanged
CMOVE Unchanged
COLD Unchanged in meaning; does additional jobs.
COMPILE Unchanged
CONFIG Unchanged
CONSTANT Unchanged
CONTEXT Unchanged
CONVERT Unchanged

COPIES Unchanged
COPY Unchanged
COUNT Unchanged

CR Unchanged

But note that it calls ?PAUSE as it has done since xForth 1.21

CREATE Unchanged
CRS Unchanged

CS-SIZE (--- n)

Return the number of bytes in the cold start table which is used to initialize the USER variables from S0 up.

CS-TABLE (--- addr)

Return the address of the cold start table. See CS-SIZE.

CSP Unchanged
CTRL Unchanged
CURRENT Unchanged

CURSOR (row col ---)

Move the cursor to the given row and column, and set OUT to the value of col and >LINE to the value of row.

D+ Unchanged
D+- Unchanged
D- Unchanged
D. Unchanged
D.R Unchanged
D0< Unchanged
D0= Unchanged
D< Unchanged
D= Unchanged
DABS Unchanged
DECIMAL Unchanged
DEFAULT Unchanged

defdrv (--- n)

Return the presently selected drive known to CP/M as the default drive.

DEFINITIONS Unchanged

defuser (--- n)

Return the CP/M user number that was in force the last time COLD was called. This will be used by file operations (including DIR) if no user number is specified. Typically 0.

DEL-KEY Unchanged

delayed (--- addr)

A semaphore used by the clock monitor to hold a queue of tasks waiting to be restarted when the value returned by `time´ becomes greater than or equal to their restart-times. Not for use except by the clock monitor.

DELAYUNTIL (ud ---)

Set the USER variable `restart-time´ to ud. Then remove the present task from the active list and insert it into the list managed by the semaphore `delayed´, such that the list is in increasing order of restart times.

deposit (c addr ---)

A monitor for buffers. If the buffer at addr is not full, deposit byte c in the buffer, and signal that the buffer is not empty. If the buffer is full, remove the present task from the active list and add it to the queue managed by the buffer and its monitors.

DEPTH Unchanged

DICTLIM (--- addr) U

A USER variable containing the address used by ?DICT to determine whether the dictionary is within bounds. Set on delivery to be 2 less than the contents of S1.

DIR (+++)

Read the next WORD, delimited by blanks, from the currently selected input stream. Convert all letters to upper case and then attempt to interpret it as an ambiguous file and user specification (afus) as follows:

afus: drive:name.ext[u]

where `drive:´ is any drive specification legal for the system, `name´ is up to 8 characters legal for CP/M file names, together with ? and * with their usual meanings; `ext´ 3 such characters; and `u´ is a user number in the range 0 to 15. All parts are optional and have defaults defdrv for the drive; defuser for the user number; and empty for the name and extension except that if both are empty they are treated as `*.*´.

DLITERAL	Unchanged
DMAX	Unchanged
DMIN	Unchanged
DNEGATE	Unchanged
DO	Unchanged
DOES>	Unchanged
DOS-CALL	Unchanged (but was called CPM-CALL in some versions).
DOS-CALLb	Unchanged (but was called CPM-CALLb in some versions).
DP	Unchanged
DPL	Unchanged
DROP	Unchanged
DU<	Unchanged
DUP	Unchanged

echo (--- addr)

A variable which is used by LOAD-FILE to determine whether each line of input from streams other than the terminal or .BLK files is to be copied to the currently selected outputs before it is interpreted.

ELSE Unchanged

EMIT (c ---)

Send the character c to each of the currently selected output streams. See OUTPUTS and XEMIT.

EMITP	Unchanged
EMITT	Unchanged
EMPTY	Unchanged
EMPTY-BUFFERS	Unchanged
ENDCASE	Unchanged
ENDIF	Unchanged
ENDOF	Unchanged
ENSURE-LINE	Unchanged
EOF	Unchanged

eof? (--- flag)

A constant used by EXPECT to signal to LOAD-FILE that the end of file has been reached on an input stream. See EXPECT and XEOF.

eol? (--- flag)

Return TRUE if a return ^M or an end of file ^Z has just been read, or if at the end of a block when BLK is nonzero.

ERASE Unchanged

ERR>IN (--- addr)

A variable which is set by STD-ERROR to the value of >IN when the error occurred. Used by WHERE.

ERRBLK (--- addr)

A variable which is set by STD-ERROR to the value of BLK when the error occurred. Used by WHERE.

ERROR	Unchanged
EXECUTE	Unchanged

exist-delayed (--- addr)

A semaphore used to put the clock monitor into a waiting state if there are no delayed tasks to execute. Not to be used except by the clock monitor.

EXIT Unchanged. (The code compiled by `;´)

EXPECT (addr n ---)

Reads up to n characters from the currently selected input stream and stores them in a string beginning at addr. Allow editing by character and line deletion when the characters in DEL-KEY and CAN-KEY are received. The string is terminated when n characters have been read, or if return ^M or end of file ^Z is read. A space is output if return is read. The USER variable SPAN is set to the number of characters actually put in the buffer. Tabs are expanded to multiples of `tabsize´, line-feeds are ignored, and ^Z, as well as being treated as a carriage return, causes the code in XEOF to be executed after the line has been interpreted. These changes are to allow input from text files. See LOAD-FILE.

EXPECT$ (addr n --- addr n')

As EXPECT but return the string read.

fallocate (file --- n)

If the file is already allocated to a virtual memory segment, return the number of the segment (0 to #FILES-1). If the file is not allocated, allocate it an unused segment if there is one and return its number. Otherwise return 0. (Note that SYSFILE is permanently allocated to segment 0.)

FALSE Unchanged
fassign Unchanged (but it is better to use fallocate).
FCREATE Unchanged
FENCE Unchanged

FENCE-BELOW (+++)

Read the next word from the input stream, delimited by blanks, and call ERROR if it is not in the dictionary. Example:

FENCE-BELOW word

Then set the value of FENCE to what HERE was before `word´ was created.

fetch (addr --- c)

If the buffer at addr is nonempty, remove a character c from it and signal that it is nonfull. If it is empty, remove the current task from the active list and put it on a queue managed by the buffer and its monitors deposit and fetch.

FILE	Unchanged
FILL	Unchanged
FIND	Unchanged
FIRST	Unchanged
fname!	Unchanged
FORGET	Unchanged
FORTH	Unchanged

frelease (n ---)

Close the file that is allocated to virtual memory segment n, and mark the segment as unused.

GET-FILE (u --- ul addr)

Given a virtual memory block number u, return the address of the corresponding file and the operating system sector ul of the start of that block. If the block corresponds to no file, call ERROR.

getuser (--- n)

Return the presently selected CP/M user number.

HERE	Unchanged
HEX	Unchanged
HLD	Unchanged
HOLD	Unchanged
I	Unchanged
ID.	Unchanged
IF	Unchanged
IMMEDIATE	Unchanged

in-addr (--- addr)

Return the address of the next character to be read when interpreting.

in-range? Unchanged

INPUTS (--- addr) U

A USER variable whose least significant 4 bits determine what input stream is selected when BLK is zero. KEY will take input from the stream with the lowest bit set; conventionally, setting the lowest bit by `1 INPUTS !´ will cause input to be taken from the CP/M terminal device. If INPUTS is 0 then EOF is returned by KEY. See KEY and XKEY.

INSTALL-$$$ Unchanged
INTERPRET Unchanged (but vectored through XINTERPRET).
INTRPT-CHAR Unchanged
J Unchanged

KEY (--- c)

If the 4 low order bits of INPUTS are set to 0, return EOF. Otherwise, execute the code pointed to by the element of XKEY corresponding to the lowest order set bit of INPUTS, e.g. if KEY is set to 8=2^3 then execute the code at XKEY+2*3=XKEY+6. See INPUTS and XKEY.

L/S Unchanged

LAST-KEY (--- c)

Return the last byte input from the CP/M terminal device.

LATEST Unchanged
LEAVE Unchanged
LFA Unchanged

LIMIT (--- addr)

Execute the code pointed to by XLIMIT. Used by COLD to determine how much space is available for xForth; `addr´ should be 1 more than the last address permitted.

LIST Unchanged

LIST-FILE (+++)

Read a word from the input stream delimited by blanks, and attempt to interpret it as an unambiguous file and user specification. If unsuccessful, call ERROR. Otherwise, if the file does not exist, call ERROR. Otherwise, if the file is a .BLK file then list it in the format used by LIST. Otherwise, list it as an ASCII file, expanding tabs to multiples of `tabsize´ and calling PAGE whenever control/L is read. Output goes to all currently selected output streams; line feeds ^J are ignored and carriage returns ^M cause CR to be called.

LITERAL Unchanged
LOAD Unchanged

LOAD-FILE (+++)

Read a word from the input stream delimited by blanks, and attempt to interpret it as an unambiguous file and user specification. If unsuccessful, call ERROR. Otherwise, if the file does not exist, call ERROR. Otherwise, if the file is a .BLK file then load its first block using LOAD. Otherwise, select input stream 2 (i.e. set INPUTS to 4) and interpret one file of input; if the variable `echo´ is set to TRUE then reflect each line of the file to all currently selected outputs before attempting to interpret the line. Note that even although BLK is set to 0, the normal rules about terminating comments and execution conditionals (delimited by parens () and braces {}) are relaxed. The input file is terminated by EOF i.e. ^Z. Files for loading may be nested and .BLK and others may be mixed arbitrarily.

LOCAL (addr1 addr2 --- addr3)

Given the USER variable address addr1 for the current task and the user variable base addr2 (typically belonging to another task) return the corresponding USER variable address for that base. Example:

S0 depthtask LOCAL

LOOP Unchanged
M* Unchanged
M/ Unchanged
M/MOD Unchanged
MAX Unchanged

maxdrv (--- n)

Return the number of drives xForth thinks the system has.

MESSAGE	Unchanged
MIN	Unchanged
MOD	Unchanged
MOVE	Unchanged
MYSELF	Unchanged

n-TAB (u ---)

Used for zoned printing. Output one or more spaces to make OUT a multiple of u, if this is possible without OUT exceeding C/L, and provided there would still be at least u positions left on the line. Otherwise execute CR.

NEGATE	Unchanged
NEXT	Unchanged
NFA	Unchanged
NOOP	Unchanged
NOT	Unchanged

not-in-memory? Internal use. Do not use.

NUMBER	Unchanged
OF	Unchanged

off (addr ---)

Set the two bytes at addr to FALSE.

on (addr ---)

Set the two bytes at addr to TRUE.

OPEN	Unchanged
OR	Unchanged
OUT	Unchanged

OUTPUTS (--- addr) U

A USER variable determining which output streams are selected. EMIT sends its byte to each stream whose bit is set; there are 4 streams, corresponding to the 4 low order bits of OUTPUTS. Conventionally, output stream 0 is the VDU (i.e. the CP/M standard output device). OUTPUTS is set to 1

on cold start and after errors, so that output goes to the VDU in such cases.

OVER Unchanged
P! Unchanged
P@ Unchanged
PAD Unchanged
PAGE Unchanged

PAUSE (---)

Move the present task to the tail of the active list and transfer control to the task which is now at the head of the active list. Should always be used in "busy waits" (loops which do nothing while waiting for some event to happen such as a key press) though semaphores are the preferred and usually more efficient way to handle cases such as this. Is implicitly called by KEY if there is no character waiting to be read, and by ?PAUSE if the output pause character in XOFF-CHAR has been pressed.

PFA Unchanged

physical-eof? (--- flag)

TRUE if the last disc input operation attempted to read an unallocated operating system sector.

PICK Unchanged

PREV Unchanged

PROTECT

Unchanged except that it calls EMPTY after setting other variables. This ensures the vocabulary links are correct.

QUERY Unchanged

QUIT (---)

If the current task is the main user task, clear the return stack, set interpret mode and select console input. Otherwise, put the task in the stopped state.

R# Unchanged
R/W Unchanged
R0 Unchanged

R1 (--- addr) U

A USER variable containing the return stack lower bound; used by ?STACK in checking that the stack is within bounds.

R> Unchanged
R@ Unchanged

remove (addr1 --- addr2)

Assuming addr1 is the address of a variable which is a circular list tail pointer, remove the head of the list and leave its address addr2. Typically used to remove a task from a queue.

REPEAT Unchanged
REPLACED-BY Unchanged

restart-time (--- addr) U

A USER 2VARIABLE containing the time a task will be restarted by the clock monitor. Only valid when the task is on the list managed by the semaphore `delayed´.

RESTORE-$$$ Unchanged
REVERSE Unchanged
ROLL Unchanged
ROT Unchanged
RP! Unchanged
RP@ Unchanged
S->D Unchanged
S0 Unchanged

S1 (--- addr) U

A USER variable containing the stack lower limit. Used by ?STACK to check that the stack is within bounds.

SAVE-BUFFERS Unchanged

SCR Unchanged
SEE Unchanged
SEE-FILE Unchanged
seg-size Unchanged

SEMAPHORE (+++)

A defining word used to define semaphores as in

```
SEMAPHORE printer
```

which defines a semaphore called `printer'. When `printer' executes it leaves its address on the stack. Typical usage is

```
printer WAIT
....                    -- Some operations
using the printer
printer AVAILABLE
```

See AVAILABLE and SIGNAL and WAIT.

setuser (n ---)

Set the current operating system user number to n.

SIGN Unchanged

SIGNAL (addr ---)

In conjunction with WAIT, manage a counting semaphore. That is, assuming addr is the address of a semaphore, check whether there are any tasks waiting in the semaphore's queue and if so, transfer the first one to the tail of the active list but do not call PAUSE. Otherwise, increment the semaphore's count by one. SIGNAL is mainly useful in buffer management but is included since counting semaphores have other uses. For normal use, AVAILABLE is more appropriate.

skip-until (c ---)

Advance the input pointer >IN until a character equal to c has been reached or the end of file is reached. In this context, a file is any of

1. A line of input when BLK is zero and the input stream is not 2;

2. A complete ASCII file terminated by EOF when BLK is zero and the input stream is 2;

3. A 1K block of virtual memory when BLK is nonzero.

In case 2 a new line is read into TIB and >IN is set to 0 if the end of a line is reached.

skip-while (c ---)

Advance the output pointer until it points to a character other than c or until the end of file has been reached. A file is defined in the same way as for `skip-until´.

SMUDGE Unchanged

SOFTWRAP (--- flag)

A constant that is TRUE except in Torch/BBC systems; it causes ^EMIT to call CR if there is insufficient space on the line for the character it is outputting, as measured by the contents of OUT compared with the value of C/L.

SP! Unchanged
SP@ Unchanged
SPACE Unchanged
SPACES Unchanged

SPAN (--- addr) U

A USER variable set by EXPECT to the number of characters it actually read.

START (addr ---)

Assuming addr is the address of the start of the user variable area of a task and assuming that task is in the stopped state, initialize it to start executing at the beginning of its code and append it to the tail of the active list. Then call PAUSE. If the task is not in a stopped state the effect is undefined (and often disastrous).

STATE Unchanged

STD-ERROR (n ---)

The standard xForth error routine which outputs, on stream 0 and any other streams currently selected, an error message which has been read from disc if the low order bit of WARNING is set and a numeric error message otherwise. Then it sets OUTPUTS to 1 (i.e. selects output stream 0) and calls QUIT.

STOP (addr ---)

Assuming addr is the address of the beginning of the user variable area of a task, and assuming the task is on the active list but is not executing, set its next instruction so that it will be removed from the active list and put in the stopped state as soon as it attempts to execute.

STRING Unchanged
SWAP Unchanged

SYSADAPT Unchanged

SYSFILE Unchanged
SYSGEN Unchanged

TAB (n ---)

If the contents of OUT are greater than or equal to n, do nothing. Otherwise, call SPACE often enough to make the value of OUT equal to n.

tabsize (--- n)

A constant used (by the screen editor and by EXPECT) to determine the width of a tab stop. For example, EXPECT will convert a tab character one or more spaces so as to make the number of characters input equal to a multiple of tabsize.

TASK: (+++)

Define a foreground task; that is, allocate space for #UVARS USER variables and initialize them according to the pattern in taskframe, but with the values offset correctly. Then perform various pre-initialization tasks, and finally switch to compilation mode. If `;´ is executed successfully

the result is to define a task that is initially in the stopped state. A foreground task may perform any functions but typically only has a small dictionary area used mainly for the sake of its PAD and to allow WORD to be called. Unlike a background task, it has a terminal input buffer and a PAD; it may use all file handling, string handling and number formatting words.

taskframe (--- addr)

The address of an 8 byte region which is used to determine the sizes of the stacks, of the terminal input buffer, and of the private dictionary in a background task. The values are the size of the stack, the size of the return stack, the size of the terminal input buffer and the size of the dictionary area in that order. On delivery the sizes of both stacks are 128 bytes, that of the TIB is 86 bytes and that of the dictionary is 192 bytes. Note that the PAD is 68 bytes above the dictionary pointer and is used widely, so do not make the dictionary area too small. On the other hand, it is often acceptable to make the size of the TIB zero since this is not normally used except during compilation.

THEN	Unchanged
TIB	Unchanged
TOGGLE	Unchanged
TRUE	Unchanged
TYPE	Unchanged
U$	Unchanged
U*	Unchanged
U.	Unchanged
U.R	Unchanged
U/MOD	Unchanged
U<	Unchanged
UCHAR	Unchanged
UNTIL	Unchanged

UP! (u ---)

Set the user variable pointer to u. VERY DANGEROUS. Used by the `windows´ demonstration to fool a word into saving or restoring the main user task's variables instead of those of the calling task.

UP@ (--- addr)

Leave on the stack the address of the beginning of the user variable area for the present task. For system use, but may be useful in allowing a word to determine what task is calling it.

UPDATE	Unchanged
USE	Unchanged

USER (+++)

Define a new USER variable, incrementing #UVARS so that the next defined USER variable will have a unique location. Note that COLD leaves space for 16 new user variables to be defined.

VARIABLE	Unchanged
VLIST	Unchanged
VOC-LINK	Unchanged
VOCABULARY	Unchanged

WAIT (addr ---)

Assuming addr is the address of a semaphore, check whether its count is nonzero. If this is the case, decrement the count and continue execution. Otherwise, remove the present task from the active list and transfer it to the tail of the semaphore's queue and transfer control to the new head of the active list.

WARM	Unchanged
WARNING	Unchanged

WHERE (---)

Invoke the screen editor with the cursor at the block and position within the block given by the contents of ERRBLK and ERR>IN.

WHILE	Unchanged
WIDTH	Unchanged
WORD	Unchanged
wrap	Unchanged
XCURSOR	Unchanged

XEMIT (--- addr)

Leave on the stack the start of a region containing 4 execution addresses which are used by EMIT when the corresponding bits of OUTPUTS are set. Thus when the low order bit is set, the action is XEMIT @ EXECUTE, when the next bit is set it is XEMIT 2+ @ EXECUTE and so on.

XEOF (--- addr)

An execution variable containing the code to be called when EXPECT reads the end of a file when INPUTS is set to 4 (i.e. input stream 2 is selected).

XERROR (--- addr)

An execution variable containing the code to be called by ERROR. Set to STD-ERROR on delivery.

XINTERPRET (--- addr)

An execution variable containing the code called when INTERPRET executes. Used by the metacompiler; lethal; avoid.

XINTRPT (--- addr)

An execution variable containing the code called when the user interrupt key contained in INTRPT-KEY is read by ?PAUSE.

XKEY (--- addr)

Leave on the stack the start of a region containing 4 execution addresses which are used by KEY. The code corresponding to the lowest order set bit of INPUTS is called. Conventionally, the low order bit corresponds to the operating system standard input.

XLIMIT (--- addr)

An execution variable containing the code executed by LIMIT. Set to `6 @´ on delivery.

XNUMBER	Unchanged
XOFF-CHAR	Unchanged
XOK	Unchanged
XOR	Unchanged
XPAGE	Unchanged
XPROMPT	Unchanged
XRUBOUT	Unchanged
XSIGNON	Unchanged
[	Unchanged
[,]VARIABLE	Unchanged
[COMPILE]	Unchanged
[]VARIABLE	Unchanged
]	Unchanged

^EMIT (c ---)

Strip the high order bit of c so it lies in the range 0 to 127. If it is 127, discard it and call ." ^?" while if it is 32 to 126 inclusive, call EMIT for the value on the stack. If it is 13 (^M) call CR; if it is 12 (^L) call PAGE; if it is 10 (^J) ignore it; if it is 9 (^I) tab to the next multiple of tabsize. Otherwise output a caret ^ followed by the stack value plus 64, so that 3 is output as ^C and so on.

^TYPE ($ ---)

If $ has length zero, do nothing. Otherwise, call ^EMIT for each of the characters in the string in turn.

{	Unchanged
|	Unchanged
}	Unchanged