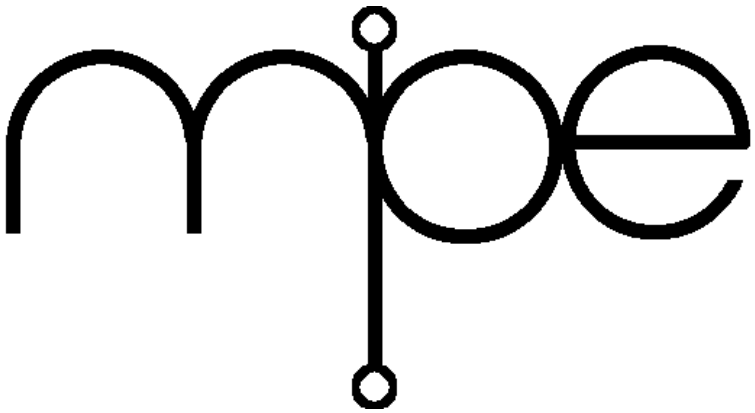
Kinetis K60 Cortex-M4 Target Code

v7.3

Microprocessor Engineering Limited



Published by Microprocessor Engineering

Kinetis K60 Cortex-M4 Target Code v7.3
User manual
Manual revision 7.3
2 April 2014

Software
Software version 7.3

For technical support
Please contact your supplier

For further information
MicroProcessor Engineering Limited
133 Hill Lane
Southampton SO15 5AF
UK

Tel: +44 (0)23 8063 1441
Fax: +44 (0)23 8033 9691
e-mail: mpe@mpeforth.com
tech-support@mpeforth.com
web: www.mpeforth.com

Table of Contents

1 Introduction

This manual documents the MPE PowerForth system for the Kinetis K60 hardware. The code was built and tested for a Freescale Tower System with the TWR-K60N512 and TWR-SER modules. The CPU board must be jumpered for the 50MHz module on the the TWR-SER board and the Ethernet PHY on the TWR-SER module set up for RMII operation.

When you program the board using the P&E or Freescale tools, you may have to power cycle the board.

1.1 Tower System set up

At MPE, the system is powered by a USB cable plugged into the primary elevator card.

The Forth console runs on RS232 (Tx, Rx only) at 115200 connected to the RS232 DB9 on the TWR-SER module. This is connected to UART3 on the K60.

If used with Powernet, an Ethernet cable is connected to the Ethernet connector on the TWR-SER board.

The jumpers on the MPE boards are set as shown below.

```
TWR-K60N512
J1    on/closed
J2    off/open
J6    Jumper on 2-3, clocked at 50MHz from TWR-SER
J8    on/closed
J9    Jumper 2-3 if coin cell connected
J10   off/open
J12   off/open
```

```
TWR-SER
J2    Jumper 3-4 (centre) for 50MHz
J5    default
J6    all open, no PHY interrupt
J10   Jumper 1-2
J11   all open, no OTG interrupt
J12   Jumper 9-10, RMII selected
J13   all open
J15   Jumper 1-2 for RS232
J16   Jumper 1-2
J17   Jumper 1-2 for RS232
J18   all open, RTS/CTS unused
J19   Jumper 1-2 for RS232
```

To put the initial Forth system into the K60, connect your usual debugger or BDM unit to the TWR-K60N512 module and Flash the board with the file *K60TWR.IMG*. MPE .IMG files are simple memory images. Power cycle the board, and the Forth should sign on at the serial port.

1.2 Terminal emulators

You can also use almost any terminal emulator, including the Windows HyperTerm.

AIDE is MPE's Integrated Development Environment (IDE) that includes a simple editor for your source code and a terminal emulator (PowerTerm) tuned for use with the PowerForth on the board. AIDE is provided with MPE development tools and is what we use with our cross compilers.

Set the terminal emulator to 115200 baud, 8 data bits, no parity, 1 or 2 stop bits.

1.3 Using the Forth

The interactive Forth on the board can use AIDE as a file server for source code. Note that code compiled on the target Forth:

- is always in RAM,
- and the on-board compiler is very simple. The code produced by the cross compiler is very much faster and shorter.

The real virtue of the on-board compiler is for configuration and testing purposes. The on-board interpreter is also heavily used as a scripting engine by the web server. We also use AIDE to set the clock. In the PowerTerm window, press Ctrl-D. This sends Forth words that set the clock to your current time and date.

If you have the PowerNet demo file *K60twrNet.img*, put it into the Flash. If you are using AIDE with the file server enabled, you can do this by typing

```
REFLASH
```

and selecting *K60twrNet.img* when the file selecter dialog opens. If you are not using AIDE, you will have to start the XModem 128/checksum download manually. If this fails, you will have to go back to your BDM/JTAG tools. After the board signs on and an Ethernet cable is connected, type:

```
PowerNet
```

Point a browser at the IP address, and you will see a simple web page served.

If you have a Telnet client, connect it to port 5023 on the board. This will log in to another Forth console.

1.4 About the manual

This manual is derived directly from the Forth source code used to generate the on-chip Forth. The full source code is supplied with the MPE VFX Forth ARM/Cortex Cross Compiler. Consequently the documentation includes some words that do not have target entries in the on-chip Forth.

Some words and code routines are marked in the documentation as `INTERNAL`. These are factors used by other words and do not have dictionary entries in the standalone Forth. They are only

accessible to users of the VFX Forth ARM Cross Compiler. This also applies to definitions of the form:

```
n EQU <name>
PROC <name>
L: <name>
```

1.5 Technical support

Technical support is available from your supplier in first instance, or from MicroProcessor Engineering.

```
tel: +44 (0)23 8063 1441
fax: +44 (0)23 8033 9691
net: mpe@mpeforth.com
     tech-support@mpeforth.com
web: www.mpeforth.com

From North America, our telephone and fax numbers are:
  011 44 23 8063 1441
  011 44 23 8033 9691
  901-313-4312 (rings in UK)
```

1.6 Licensing

If you want to distribute applications built with the code you have two options.

- Purchase an MPE VFX Forth ARM/Cortex Cross Compiler. This includes the full source code for ARM and Cortex CPUs and unlimited application distribution rights, provided that you do not ship an open Forth interpreter. If you need to ship an open Forth interpreter for engineering and maintenance access, you must get permission from MPE in writing (a fax will do), and this is normally provided free of charge.
- Purchase an OEM PowerForth license, either on a quantity basis or an unlimited basis. This license includes rights to redistribute the software manuals in PDF and HTML form.

2 Cortex start up for Kinetis K60

The file *Cortex/Hardware/Kinetis/startK60twr.fth* contains start up code for Kinetis K60 devices. Start up code for other Cortex-M3/M4 implementations, e.g. STM32, can be found in parallel directories. The absolute minimum code to start a Cortex-M4 is in *Cortex/StartCortex.fth*.

Note that this start up code is specific to the K60N512 Tower Module. Use this file as the basis of your own code after copying and renaming the file.

```
l: ExcVecs      \ -- addr ; start of vector table
```
The exception vector table is `/ExcVecs` bytes long. The equate is defined in the control file. Note that this table table must be correctly aligned as described in the ARMv7-M Architecture Reference Manual. If placed at 0 or the start of Flash, you'll usually be fine.

```
: SetExcVec     \ addr exc# --
```
Set the given exception vector number to the given address. Note that for vectors other than 0, the Thumb bit is forced to 1.

```
L: CLD1         \ holds xt of main word
```
Holds the xt of the main action after initial booting. For a standalone Forth, this is usually the xt of `COLD`.

```
0 equ sp-guard  \ -- +n
```
The number of cells reserved as guard space on the Forth data stack. Usually set to two in the control file. The value allows the data stack to underflow by the given amount before other data is corrupted.

```
init-s0 tos-cached? sp-guard + cells - equ real-init-s0  \ -- addr
```
The data stack pointer value set at start up.

```
  equ vCLKDIV1  \ -- x
```
Initial value for clock divider registers.

```
IsLeaf : (setClocks)    \ --
```
Enable the clocks, set the bus dividers, and enable the PLLs as required. This routine is run from RAM. See errata sheet *KINETIS_0M33Z.pdf*, e2448.

```
uhere equ rSetClocks    \ -- addr
```
The start of the RAM buffer used for the clock set up code. By default, the buffer space is reserved. If you know that `setClocks` will only be run once at start up, you may be able to get away with not reserving the space!

```
: setClocks     \ --
```
Initialise the clocks and peripheral dividers. All GPIO port clocks are enabled.

```
: setNVIC       \ --
```
Initialise the NVIC. By default it points into Flash.

```
: disDog        \ -- ; no watchdog
```
Disable the watchdog at start up.

```
: StartCortex   \ -- ; never exits
```
Set up the Forth registers and start Forth. Other primary hardware initialisation is also performed here.

3 Cortex code definitions

The file *Cortex/CodeCortex.fth* contains primitives for the standalone Forth kernel built for Cortex-M3/M4 CPUs. See *Cortex/CodeM0M1.fth* for Cortex-M0/M1 CPUs.

3.1 Notes

Some words and code routines are marked in the documentation as `INTERNAL`. These are factors used by other words and do not have dictionary entries in the standalone Forth. They are only accessible to users of the VFX Forth ARM Cross Compiler. This also applies to definitions of the form:

```
n EQU <name>

PROC <name>

L: <name>
```

3.2 Register usage

For Cortex-M3/M4 the following register usage is the default:

```
r15         pc      program counter
r14         link    link register; bit0=1=Thumb, usually set
r13         rsp     return stack pointer
r12         psp     data stack pointer
r11         up      user area pointer
r10         --
r9          lp      locals pointer
r8          --
r7          tos     cached top of stack
r0-r6       scratch
```

The VFX optimiser uses R0 and R1 for internal operations - you can make no assumptions about their contents on entry to a word. `CODE` definitions must use R7 as TOS with NOS pointed to by R12 as a full descending stack in ARM terminology. R0..R8 are free for use by `CODE` definitions and need not be preserved or restored. You should assume that any register can be affected by other words.

For Cortex-M0/M1 code generation, the parameter stack pointer is moved from R12 to R6.

```
r15         pc      program counter
r14         link    link register
r13         rsp     return stack pointer
r12         --
r11         up      user area pointer
r10         --      RFU
r9          lp      locals pointer
r8          --      RFU
r7          tos     cached top of stack
r6          psp     data stack pointer
r0-r5       scratch
```

3.3 Configuration

These equates are set false (zero) if they have not already been defined.

`false equu DSQRT?  \ -- flag`
Set this non-zero to compile `DSQRT`.

`false equ FastCmove?  \ -- flag`
Set this flag true to use a fast but vast (~1kb) version of `CMOVE`. If your application uses either `CMOVE` or `MOVE` in time-critical code, the fast version offers an overall speed up of about four times. The code for the fast but vast version was written by Rowley Associates, whose permission to adapt and publish the code with the MPE cross compiler is much appreciated.

3.4 Logical and relational operators

`: AND  \ x1 x2 -- x3`
Perform a logical AND between the top two stack items and retain the result in top of stack.

`: OR  \ x1 x2 -- x3`
Perform a logical OR between the top two stack items and retain the result in top of stack.

`: XOR  \ x1 x2 -- x3`
Perform a logical XOR between the top two stack items and retain the result in top of stack.

`: INVERT  \ x -- x'`
Perform a bitwise inversion.

`: 0=  \ x -- flag`
Compare the top stack item with 0 and return TRUE if equals.

`: 0<>  \ x -- flag`
Compare the top stack item with 0 and return TRUE if not-equal.

`: 0<  \ x -- flag`
Return TRUE if the top of stack is less-than-zero.

`: 0>  \ x -- flag`
Return TRUE if the top of stack is greater-than-zero.

`: =  \ x1 x2 -- flag`
Return TRUE if the two topmost stack items are equal.

`: <>  \ x1 x2 -- flag`
Return TRUE if the two topmost stack items are different.

`: <  \ n1 n2 -- flag`
Return TRUE if n1 is less than n2.

`: >  \ n1 n2 -- flag`
Return TRUE if n1 is greater than n2.

`: <=  \ n1 n2 -- flag`
Return TRUE if n1 is less than or equal to n2.

`: >=  \ x1 x2 -- flag`
Return TRUE if n1 is greater than or equal to n2.

`: U>  \ u2 u2 -- flag`
An UNSIGNED version of `>`.

`: U<  \ u1 u2 -- flag`
An UNSIGNED version of `<`.

`CODE DU<  \ ud1 ud2 -- flag`
Returns true if ud1 (unsigned double) is less than ud2.

`: D0<  \ d -- flag`
Returns true if signed double d is less than zero.

`: D0=  \ xd -- flag`
Returns true if xd is 0.

`CODE D=  \ xd1 xd2 -- flag`
Return TRUE if the two double numbers are equal.

`CODE D<  \ d1 d2 -- flag`
Return TRUE if the double number d1 is (signed) less than the double number d2.

`CODE DMAX  \ d1 d2 -- d3 ; d3=max of d1/d2`
Return the maximum double number from the two supplied.

`CODE DMIN  \ d1 d2 -- d3 ; d3=min of d1/d2`
Return the minimum double number from the two supplied.

`CODE MIN  \ n1 n2 -- n1|n2`
Given two data stack items preserve only the smaller.

`CODE MAX  \ n1 n2 -- n1|n2`
Given two data stack items preserve only the larger.

`CODE WITHIN?  \ n1 n2 n3 -- flag`
Return TRUE if N1 is within the range N2..N3. This word uses signed arithmetic.

`CODE WITHIN  \ n1|u1 n2|u2 n3|u3 -- flag`
The ANS version of WITHIN?. This word uses unsigned arithmetic, so that signed compares are treated as existing on a number circle.

`CODE LSHIFT  \ x1 u -- x2`
Logically shift X1 by U bits left.

`CODE RSHIFT  \ x1 u -- x2`
Logically shift X1 by U bits right.

3.5 Control flow

`CODE EXECUTE  \ xt --`
Execute the code described by the XT. This is a Forth equivalent to an assembler JSR/CALL instruction.

`CODE BRANCH  \ --`
The run time action of unconditional branches compiled on the target. INTERNAL.

`CODE ?BRANCH  \ n --`
The run time action of conditional branches compiled on the target. INTERNAL.

`CODE (OF)  \ n1 n2 -- n1|--`
The run time action of OF compiled on the target. INTERNAL.

`CODE (LOOP)  \ --`
The run time action of `LOOP` compiled on the target. INTERNAL.

`CODE (+LOOP)  \ n --`
The run time action of `+LOOP` compiled on the target. INTERNAL.

`CODE (DO)  \ limit index --`
The run time action of `DO` compiled on the target. INTERNAL.

`CODE (?DO)  \ limit index --`
The run time action of `?DO` compiled on the target. INTERNAL.

`CODE LEAVE  \ --`
Remove the current `DO..LOOP` parameters and jump to the end of the `DO..LOOP` structure.

`CODE ?LEAVE  \ flag --`
If flag is non-zero, remove the current `DO..LOOP` parameters and jump to the end of the `DO..LOOP` structure.

`CODE I  \ -- n`
Return the current index of the inner-most `DO..LOOP`.

`CODE J  \ -- n`
Return the current index of the second `DO..LOOP`.

`CODE UNLOOP  \ -- ; R: loop-sys --`
Remove the `DO..LOOP` control parameters from the return stack.

3.6 Basic arithmetic

`CODE S>D  \ n -- d`
Convert a single number to a double one.

`: D>S  \ d -- n`
Convert a double number to a single.

`: NOOP ;  \ --`
A NOOP, null instruction.

`CODE M+  \ d1|ud1 n -- d2|ud2`
Add double d1 to sign extended single n to form double d2.

`: 1+  \ n1|u1 -- n2|u2`
Add one to top-of stack.

`: 2+  \ n1|u1 -- n2|u2`
Add two to top-of stack.

`: 4+  \ n1|u1 -- n2|u2`
Add four to top-of stack.

`: 1-  \ n1|u1 -- n2|u2`

Subtract one from top-of stack.

```
: 2-            \ n1|u1 -- n2|u2
```
Subtract two from top-of stack.

```
: 4-            \ n1|u1 -- n2|u2
```
Subtract four from top-of stack.

```
: 2*            \ x1 -- x2
```
Multiply top of stack by 2.

```
: 4*            \ x1 -- x2
```
Multiply top of stack by 4.

```
: 2/            \ x1 -- x2
```
Signed divide top of stack by 2.

```
: U2/           \ x1 -- x2
```
Unsigned divide top of stack by 2.

```
: 4/            \ x1 -- x2
```
Signed divide top of stack by 4.

```
: U4/           \ x1 -- x2
```
Unsigned divide top of stack by 4.

```
CODE +          \ n1|u1 n2|u2 -- n3|u3
```
Add two single precision integer numbers.

```
CODE -          \ n1|u1 n2|u2 -- n3|u3
```
Subtract two integers. N3|u3=n1|u1-n2|u2.

```
CODE NEGATE     \ n1 -- n2
```
Negate an integer.

```
CODE D+         \ d1 d2 -- d3
```
Add two double precision integers.

```
CODE D-         \ d1 d2 -- d3
```
Subtract two double precision integers. D3=D1-D2.

```
CODE DNEGATE    \ d1 -- -d1
```
Negate a double number.

```
CODE ?NEGATE    \ n1 flag -- n1|n2
```
If flag is negative, then negate n1.

```
CODE ?DNEGATE   \ d1 flag -- d1|d2
```
If flag is negative, then negate d1.

```
CODE ABS        \ n -- u
```
If n is negative, return its positive equivalent (absolute value).

```
CODE DABS       \ d -- ud
```
If d is negative, return its positive equivalent (absolute value).

```
CODE D2*        \ xd1 -- xd2
```
Multiply the given double number by two.

```
CODE D2/        \ xd1 -- xd2
```
Divide the given double number by two.

3.7 Multiplication

`: UM*  \ u1 u2 -- ud`
Perform unsigned-multiply between two numbers and return double result.

`: *  \ n1 n2 -- n3`
Standard signed multiply. N3 = n1 * n2.

`: m*  \ n1 n2 -- d`
Signed multiply yielding double result.

3.8 Division

ARM Cortex provides 32/32 division instructions, but no 64/32 ones. Avoid 64/32 division routines if you can where performance matters.

`macro: udiv64_step  \ --`
Cross compiler macro to perform one step of the unsigned 64 bit by 32 bit division

`code um/mod  \ ud1 u2 -- urem uquot`
Slow and short - Full 64 by 32 unsigned division subroutine. This routine uses a loop for code size. This version is commented out by default.

`code um/mod  \ ud1 u2 -- urem uquot`
Fast and big - Full 64 by 32 unsigned division subroutine. Unrolled for speed. This routine uses 660 bytes of code space using the Thumb-2 instruction set, whereas the ARM32 version uses 920 bytes.

`macro: udiv63_step  \ --`
Cross compiler macro to perform one step of the unsigned 63 bit by 31 bit division

`proc Udiv63/31  \ r0:r1/tos ; 63/31 unsigned divide -> tos=quot, r0=rem`
Unsigned division primitive - unrolled for speed. Note that this routine does not handle the top bit of the divisor and dividend correctly. `Udiv63/31` is used for signed divide operations for which the top bits are always zero.

`CODE FM/MOD  \  d1 n2 -- rem quot ; floored division`
Perform a signed division of double number D1 by single number N2 and return remainder and quotient using floored division. See the ANS Forth specification for more details of floored division.

`CODE SM/REM  \ d1 n2 -- rem quot ; symmetric division`
Perform a signed division of double number D1 by single number N2 and return remainder and quotient using symmetric (normal) division.

`CODE /MOD  \ n1 n2 -- rem quot`
Signed symmetric division of N1 by N2 single-precision returning remainder and quotient.

`: /  \ n1 n2  -- n3`
Standard signed division operator. n3 = n1/n2.

`: MOD  \ n1 n2 -- n3`
Return remainder of division of N1 by N2. n3 = n1 mod n2.

`: M/  \ d n1 -- n2`
Signed divide of a double by a single integer.

`: MU/MOD  \ d n -- rem d#quot`
Perform an unsigned divide of a double by a single, returning a single remainder and a double quotient.

3.9 Scaling - multiply then divide

These operations perform a multiply followed by a divide. The intermediate result is in an extended form. The point of these operations is to avoid loss of precision.

```
: */MOD          \ n1 n2 n3 -- n4 n4
```
Multiply n1 by n2 to give a double precision result, and then divide it by n3 returning the remainder and quotient.

```
: */             \ n1 n2 n3 -- n4
```
Multiply n1 by n2 to give a double precision result, and then divide it by n3 returning the quotient.

```
: m*/            \ d1 n2 n3 -- dquot
```
The result dquot=(d1*n2)/n3. The intermediate value d1*n2 is triple-precision to avoid loss of precision. In an ANS Forth standard program n3 can only be a positive signed number and a negative value for n3 generates an ambiguous condition, which may cause an error on some implementations, but not in this one.

3.10 Stack manipulation

```
: NIP            \ x1 x2 -- x2
```
Dispose of the second item on the data stack.

```
: TUCK           \ x1 x2 -- x2 x1 x2
```
Insert a copy of the top data stack item underneath the current second item.

```
CODE PICK        \ xu .. x0 u -- xu .. x0 xu
```
Get a copy of the Nth data stack item and place on top of stack. 0 PICK is equivalent to DUP.

```
CODE ROLL        \ xu xu-1 .. x0 u -- xu-1 .. x0 xu
```
Rotate the order of the top N stack items by one place such that the current top of stack becomes the second item and the Nth item becomes TOS. See also `ROT`.

```
: ROT   \ x1 x2 x3 -- x2 x3 x1
```
ROTate the positions of the top three stack items such that the current top of stack becomes the second item. See also `ROLL`.

```
: -ROT  \ x1 x2 x3 -- x3 x1 x2
```
The inverse of `ROT`.

```
CODE >R          \ x -- ; R: -- x
```
Push the current top item of the data stack onto the top of the return stack.

```
CODE R>          \ -- x ; R: x --
```
Pop the top item from the return stack to the data stack.

```
CODE R@          \ --  x  ; R:  x -- x
```
Copy the top item from the return stack to the data stack.

```
CODE 2>R         \ x1 x2 -- ; R:  -- x1 x2
```
Transfer the two top data stack items to the return stack.

```
CODE 2R>         \ -- x1 x2 ; R: x1 x2 --
```
Transfer the top two return stack items to the data stack.

```
CODE 2R@         \ --  x1 x2  ; R:  x1 x2 -- x1 x2
```
Copy the top two return stack items to the data stack.

```
CODE 2ROT        \ x1 x2 x3 x4 x5 x6 -- x3 x4 x5 x6 x1 x2
```

Perform the ROT operation on three cell-pairs.

`: SWAP \ x1 x2 -- x2 x1`
Exchange the top two data stack items.

`: DUP \ x -- x x`
DUPlicate the top stack item.

`: OVER \ x1 x2 -- x1 x2 x1`
Copy NOS to a new top-of-stack item.

`: DROP \ x --`
Lose the top data stack item and promote NOS to TOS.

`: 2DROP \ x1 x2 -- )`
Discard the top two data stack items.

`: 2SWAP \ x1 x2 x3 x4 -- x3 x4 x1 x2`
Exchange the top two cell-pairs on the data stack.

`CODE ?DUP \ x -- | x`
DUPlicate the top stack item only if it non-zero.

`CODE 2DUP \ x1 x2 -- x1 x2 x1 x2`
DUPlicate the top cell-pair on the data stack.

`CODE 2OVER \ x1 x2 x3 x4 -- x1 x2 x3 x4 x1 x2`
As OVER but works with cell-pairs rather than single-cell items.

`CODE SP@ \ -- x`
Get the current address value of the data-stack pointer.

`CODE SP! \ x --`
Set the current address value of the data-stack pointer.

`CODE RP@ \ -- x`
Get the current address value of the return-stack pointer.

`CODE RP! \ x --`
Set the current address value of the return-stack pointer.

3.11 String and memory operators

`: COUNT \ c-addr1 -- c-addr2' u`
Given the address of a counted string in memory this word will return the address of the first character and the length in characters of the string.

`CODE /STRING \ c-addr1 u1 n -- c-addr2 u2`
Modify a string address and length to remove the first N characters from the string.

`CODE SKIP \ c-addr1 u1 char -- c-addr2 u2`
Modify the string description by skipping over leading occurrences of 'char'.

`CODE SCAN \ c-addr1 u1 char -- c-addr2 u2`
Look for first occurrence of *char* in the string and return a new string. *C-addr2/u2* describes the string with *char* as the first character.

`CODE S= \ c-addr1 c-addr2 u -- flag`
Compare two same-length strings/memory blocks, returning TRUE if they are identical.

`: compare \ c-addr1 u1 c-addr2 u2 -- n` 17.6.1.0935

Compare two strings. The return result is 0 for a match or can be -ve/+ve indicating string differences. If the two strings are identical, n is zero. If the two strings are identical up to the length of the shorter string, n is minus-one (-1) if u1 is less than u2 and one (1) otherwise. If the two strings are not identical up to the length of the shorter string, n is minus-one (-1) if the first non-matching character in the string specified by c-addr1 u1 has a lesser numeric value than the corresponding character in the string specified by c-addr2 u2 and one (1) otherwise.

`: SEARCH ( c-addr1 u1 c-addr2 u2 -- c-addr3 u3 flag )`
Search the string c-addr1/u1 for the string c-addr2/u2. If a match is found return c-addr3/u3, the address of the start of the match and the number of characters remaining in c-addr1/u1, plus flag f set to true. If no match was found return c-addr1/u1 and f=0.

`code cmove \ asrc adest len --`
Copy *len* bytes of memory forwards from *asrc* to *adest*. If the performance of `CMOVE` is important in your application, set the equate `FastCmove?` non-zero and a much faster (four to five times) but much larger (~900 bytes) version will be compiled. See *Cortex\fcmove.fth* for the details.

`CODE CMOVE> \ c-addr1 c-addr2 u --`
As `CMOVE` but working in the opposite direction, copying the last character in the string first.

`: ON \ a-addr --`
Given the address of a CELL this will set its contents to TRUE (-1).

`: OFF \ a-addr --`
Given the address of a CELL this will set its contents to FALSE (0).

`CODE C+! \ b c-addr --`
Add N to the character (byte) at memory address ADDR.

`CODE 2@ \ a-addr -- x1 x2`
Fetch and return the two CELLS from memory ADDR and ADDR+sizeof(CELL). The cell at the lower address is on the top of the stack.

`CODE 2! \ x1 x2 a-addr --`
Store the two CELLS x1 and x2 at memory ADDR. X2 is stored at ADDR and X1 is stored at ADDR+CELL.

`CODE FILL \ c-addr u char --`
Fill LEN bytes of memory starting at ADDR with the byte information specified as CHAR.

`CODE +! \ n|u a-addr --`
Add N to the CELL at memory address ADDR.

`CODE INCR \ a-addr --`
Increment the data cell at a-addr by one.

`CODE DECR \ a-addr --`
Decrement the data cell at a-addr by one.

`CODE @ \ a-addr -- x`
Fetch and return the CELL at memory ADDR.

`CODE W@ \ a-addr -- w`
Fetch and 0 extend the word (16 bit) at memory ADDR.

`CODE C@ \ c-addr -- char`
Fetch and 0 extend the character at memory ADDR and return.

`CODE ! \ x a-addr --`

Store the CELL quantity X at memory A-ADDR.

`CODE W!  \ w a-addr --`
Store the word (16 bit) quantity w at memory ADDR.

`CODE C!  \ char c-addr --`
Store the character CHAR at memory C-ADDR.

`: UPC  \ char -- char'`
Convert supplied character to upper case if it was alphabetic otherwise return the unmodified character. `UPC` is English language specific.

`CODE UPPER  \ c-addr u --`
Convert the ASCII string described to upper-case. This operation happens in place.

`CODE TEST-BIT  \ mask c-addr -- flag`
AND the mask with the contents of addr and return true if the result is non-zero (-1) or false (0) if the result is zero. Byte operation.

`CODE SET-BIT  \ mask c-addr --`
Apply the mask ORred with the contents of c-addr. Byte operation.

`CODE RESET-BIT  \ mask c-addr --`
Apply the mask inverted and ANDed with the contents of c-addr. Byte operation.

`CODE TOGGLE-BIT \ u c-addr --`
Invert the bits at c-addr specified by the mask. Byte operation.

3.12 Miscellaneous words

`CODE NAME>  \ nfa -- cfa`
Move a pointer from an NFA to the XT..

`CODE >NAME  \ cfa -- nfa`
Move a pointer from an XT back to the NFA or name-pointer. If the original pointer was not an XT or if the definition in question has no name header in the dictionary the returned pointer will be useless. Care should be taken when manipulating or scanning the Forth dictionary in this way.

`: SEARCH-WORDLIST  \ c-addr u wid -- 0|xt 1|xt -1`
Search the given wordlist for a definition. If the definition is not found then 0 is returned, otherwise the XT of the definition is returned along with a non-zero code. A -ve code indicates a "normal" definition and a +ve code indicates an `IMMEDIATE` word.

`CODE DIGIT  \ char base -- 0|n true`
If the ASCII value *CHAR* can be treated as a digit for a number within the radix *base* then return the digit and a TRUE (-1) flag, otherwise return FALSE (0).

3.13 Portability helpers

Using these words will make code easier to port between 16, 32 and 64 bit targets.

`CODE CELL+  \ a-addr1 -- a-addr2`
Add the size of a CELL to the top-of stack.

`CODE CELLS  \ n1 -- n2`
Return the size in address units of N1 cells in memory.

`CODE CELL-  \ a-addr1 -- a-addr2`

Decrement an address by the size of a cell.

`CODE CELL \ -- n`
Return the size in address units of one CELL.

`CODE CHAR+ \ c-addr1 -- c-addr2`
Increment an address by the size of a character.

`: CHARS \ n1 -- n2`
Return size in address units of N1 characters.

3.14 Runtime for VALUE

`CODE VAL! \ n -- ; store value address in-line`
Store n at the inline address following this word. INTERNAL.

`CODE VAL@ \ -- n ; read value data address in-line`
Read n from the inline address following this word. INTERNAL.

3.15 Supporting complation on the target

Compilation on the target is supported for compilation into RAM. The target's compiler is simplistic and gives neither the code size nor the performance of cross-compiled code. The support words are compiled without heads.

Direct compilation into Flash requires additional code. If you need it and want support, please contact MPE.

`: !scall \ dest addr --`
Patch a BL DEST opcode at addr.

`: !lcall \ dest addr --`
Patch n MVL32 R0, # DEST+1 opcode at addr.

3.16 Defining words and runtime support

`L: DOCREATE \ -- addr`
The run time action of `CREATE`. The call must be on a four byte boundary. INTERNAL.

`CODE LIT \ -- x`
Code which when CALLED at runtime will return an inline cell value. The call must be at a four byte boundary. INTERNAL.

`CODE (") \ -- a-addr ; return address of string, skip over it`
Return the address of a counted string that is inline after the CALLING word, and adjust the CALLING word's return address to step over the inline string. The adjusted return address will be at a four byte boundary. See the definition of `(.")` for an example.

`: aligned \ addr -- addr'`
Given an address pointer this word will return the next ALIGNED address subject to system wide alignment restrictions.

`: compile, \ xt --`
Compile the word specified by xt into the current definition.

`: >BODY \ xt -- a-addr`
Move a pointer from a CFA or "XT" to the definition BODY. This should only be used with children of `CREATE`. E.g. if `FOOBAR` is defined by `CREATE foobar`, then the phrase `' foobar >body` would yield the same result as executing `foobar`.

`: DOCREATE,  \ --`
Compile the run time action of `CREATE`. INTERNAL.

`: (;CODE)  \ -- ; R: a-addr --`
Performed at compile time by `;CODE` and `DOES>`. Patch the last word defined (by `CREATE`) to have the run time actions that follow immediately after `(;CODE)`. INTERNAL.

`: (;CODE)  \ -- ; R: a-addr --`
Performed at compile time by `;CODE` and `DOES>`. Patch the last word defined (by `CREATE`) to have the run time actions that follow immediately after `(;CODE)`. INTERNAL.

`: CONSTANT  \ x "<spaces>name" -- ; Exec: -- x`
Create a new `CONSTANT` called `name` which has the value x. When `NAME` is executed x is returned.

`: 2CONSTANT  \ Comp: x1 x2 "<spaces>name" -- ; Run: -- x1 x2`
A two-cell equivalent of `CONSTANT`.

`: VARIABLE  \ "<spaces>name" -- ; Exec: -- a-addr`
Create a new variable called `name`. When `name` is executed the address of the data-cell is returned for use with `@` and `!` operators.

`: 2VARIABLE  \ Comp: "<spaces>name" -- ; Run: -- a-addr`
A two-cell equivalent of `VARIABLE`.

`: USER  \ u "<spaces>name" -- ; Exec: -- addr ; SFP009`
Create a new `USER` variable called `name`. The u parameter specifies the index into the user-area table at which to place the* data. `USER` variables are located in a separate area of memory for each task or interrupt. Use in the form:

```
$400 USER TaskData
```

`: u#  \ "<name>"-- u`
An `INTERPRETER` word that returns the index of the `USER` variable whose name follows, e.g.

```
u# S0
```

`: :  \ C: "<spaces>name" -- colon-sys ; Exec: i*x -- j*x ; R: -- nest-sys`
Begin a new definition called `name`.

`: :NONAME  \ C: -- colon-sys ; Exec: i*x -- i*x  ; R: -- nest-sys`
Begin a new code definition which does not have a name. After the definition is complete the semi-colon operator returns the XT of newly compiled code on the stack.

`: DOES>  \ C: colon-sys1 -- colon-sys2 ; Run: -- ; R:  nest-sys --`
Begin definition of the runtime-action of a child of a defining word. See the section about defining words in *Programming Forth*. You should not use `RECURSE` after `DOES>`.

`: CRASH  \ -- ; used as action of DEFER`
The default action of a `DEFER`ed word, which is to perform `#12 THROW`. INTERNAL.

`: DEFER  \ Comp: "<spaces>name" -- ; Run:  i*x -- j*x`
Creates a new `DEFER`ed word. A default action, `CRASH`, is assigned.

`: (TO-DO)  \ -- ; R: xt -- a-addr'`
The run-time action of `TO-DO`. It is followed by the data address of the `DEFER`red word at which the xt is stored.

`: FIELD  \ size n "<spaces>name" -- size+n ; Exec: addr -- addr+n`
Create a new field of n bytes within a structure so far of *size* bytes.

3.17 Structure compilation

These words define high level branches. They are used by the structure words such as `IF` and `AGAIN`.

`: >mark \ -- addr`
Mark the start of a forward branch. HIGH LEVEL CONSTRUCTS ONLY. INTERNAL.

`: >resolve \ addr --`
Resolve absolute target of forward branch. HIGH LEVEL CONSTRUCTS ONLY. INTERNAL.

`: <mark \ -- addr`
Mark the start (destination) of a backward branch. HIGH LEVEL CONSTRUCTS ONLY. INTERNAL.

`: <resolve \ addr --`
Resolve a backward branch to addr. HIGH LEVEL CONSTRUCTS ONLY. INTERNAL.

`synonym >c_res_branch >resolve \ addr -- ; fix up forward referenced branch`
See >RESOLVE. INTERNAL.

`synonym c_mrk_branch< <mark \ -- addr ; mark destination of backward branch`
See >MARK. INTERNAL.

3.18 Branch constructors

Used when compiling code on the target.

`: c_branch< \ addr --`
Lay the code for an unconditional backward branch. INTERNAL.

`: c_?branch< \ addr --`
Lay the code for a conditional backward branch.

`: c_branch> \ -- addr`
Lay the code for a forward referenced unconditional branch. INTERNAL.

`: c_?branch> \ -- addr`
Lay the code for a forward referenced conditional branch. INTERNAL.

3.19 Main compilers

`: c_lit \ lit --`
Compile the code for a literal of value *lit*. INTERNAL.

`: c_drop \ --`
Compile the code for `DROP`. INTERNAL.

`: c_exit \ --`
Compile the code for `EXIT`. INTERNAL.

`: c_do \ C: -- do-sys ; Run: n1|u1 n2|u2 -- ; R: -- loop-sys`
Compile the code for `DO`. INTERNAL.

`: c_?DO \ C: -- do-sys ; Run: n1|u1 n2|u2 -- ; R: -- | loop-sys`
Compile the code for `?DO`. INTERNAL.

`: c_LOOP \ C: do-sys -- ; Run: -- ; R: loop-sys1 -- | loop-sys2`
Compile the code for `LOOP`. INTERNAL.

`: c_+LOOP \ C: do-sys -- ; Run: -- ; R: loop-sys1 -- | loop-sys2`
Compile the code for `+LOOP`. INTERNAL.

`variable NextCaseTarg \ -- addr`
Holds the entry point of the current `CASE` structure. INTERNAL.

`: c_case \ -- addr`
Compile the code for `CASE`. INTERNAL.

`: c_OF \ C: -- of-sys ; Run: x1 x2 -- | x1`
Compile the code for `OF`. INTERNAL.

`: c_ENDOF \ C: case-sys1 of-sys -- case-sys2 ; Run: --`
Compile the code for `ENDOF`. INTERNAL.

`: FIX-EXITS \ n1..nn --`
Compile the code to resolve the forward branches at the end of a `CASE` structure. INTERNAL.

`: c_ENDCASE \ C: case-sys -- ; Run: x --`
Compile the code for `ENDCASE`. INTERNAL.

`: c_END-CASE \ C: case-sys -- ; Run: x --`
Compile the code for `END-CASE`. INTERNAL. Only compiled if the equate `FullCase?` is non-zero.

`: c_NEXTCASE \ C: case-sys -- ; Run: x --`
Compile the code for `NEXTCASE`. INTERNAL. Only compiled if the equate `FullCase?` is non-zero.

`: c_?OF \ C: -- of-sys ; Run: flag --`
Compile the code for `?OF`. INTERNAL. Only compiled if the equate `FullCase?` is non-zero.

3.20 Miscellaneous

`code di \ --`
Disable interrupts.

`code ei \ --`
Enable interrupts.

`code dfi \ --`
Disable fault exceptions.

`code efi \ --`
Enable fault exceptions.

`code [I \ R: -- x1 x2`
Preserve interrupt/exception status on the return stack, and disable interrupts/exceptions except reset, NMI and HardFault. The state is restored by `I]`.

`code I] \ R: x1 x2 --`
Restore interrupt status saved by `[I` from the return stack.

`code clz \ x -- #lz`
Count the number of leading zeros in x.

`code dsqrt \ +d -- n`
Single square root of a double number. 62 bits -> 31 bits. The equate `DSQRT?` must be set true to compile this word.

`: setMask \ value mask addr -- ; cell operation`

Clear the *mask* bits at *addr* and set (or) the bits defined by *value*.

`: init-io \ addr --`
Copy the contents of the I/O set up table to an I/O device. Each element of the table is of the form addr (cell) followed by data (cell). The table is terminated by an address of 0.

4 High level kernel kernel62.fth.

4.1 Configuration options

The following option equates are set if not already before *\i{kernel62.fth) is compiled.

```
1 equ BigKernel?        \ --
```
Set this false to reduce the kernel size and lose some words.

4.2 User variables

```
variable next-user      \ -- addr
```
Next valid offset for a `USER` variable created by `+USER`.

```
: +user         \ size --
```
Creates a `USER` variable size bytes long at the offset given by `NEXT-USER` and updates it.

```
tcb-size +user SELF             \ task identifier and TCB
```
When multitasking is enabled by setting the equate `TASKING?` the task control block for a task occupies `TCB-SIZE` bytes at the start of the user area. Thus the user area pointer also acts as a pointer to the task control block.

```
cell +user S0           \ base of data stack
```
Holds the initial setting of the data stack pointer. N.B. `S0`, `R0`, `#TIB` and `'TIB` must be defined in that order.

```
cell +user R0           \ base of return stack
```
Holds the initial setting of the return stack pointer.

```
cell +user #TIB         \ number of chars currently in TIB
```
Holds the number of characters currently in `TIB`.

```
cell +user 'TIB         \ address of TIB
```
Holds the address of `TIB`, the terminal input buffer.

```
cell +user >IN          \ offset into TIB
```
Holds the current character position being processed in the input stream.

```
cell +user XON/XOFF     \ true if XON/XOFF protocol in use
```
True when console is using XON/XOFF protocol.

```
cell +user ECHOING      \ true if echoing
```
True when console is echoing input characters.

```
cell +user OUT          \ number of chars displayed on current line
```
Holds the number of chars displayed on current output line. Reset by CR.

```
cell +user BASE         \ current numeric conversion base
```
Holds the current numeric conversion base

```
cell +user HLD          \ used during number formatting
```
Holds data used during number formatting

```
cell +user #L           \ number of cells converted by NUMBER?
```
Holds the number of cells converted by `NUMBER?`

```
cell +user #D           \ number of digits converted by NUMBER?
```
Holds the number of digits converted by `NUMBER?`

```
cell +user DPL          \ position of double number character id
```
Holds the number of characters after the double number indicator character. DPL is initialised to -1, which indicates a single number, and is incremented for each character after the separator.

```
cell +user HANDLER      \ used in catch and throw
```
Holds the address of the previous exception frame.

```
cell +user OPVEC        \ output vector
```
Holds the address of the I/O vector for the current output device.

```
cell +user IPVEC        \ input vector
```
Holds the address of the I/O vector for the current input device.

```
cell +user 'AbortText   \ Address of text from ABORT"
```
Set by the run-time action of ABORT" to hold the address of the counted string used by ABORT" <text>".

```
#64 chars dup +user PAD
```
A temporary string scratch buffer.

4.3 System Constants

Various constants for the internal system.

FALSE	The well formed flag version for a logical negative
TRUE	The well formed flag version for a logical positive
BL	An internal constant for blank space
C/L	Max chars/line for internal displays under C/LINE
#VOCS	Maximum number of Vocabularies in search order
VSIZE	Size of CONTEXT area for search order
XON	XON character for serial line flow control
XOFF	XOFF character for serial line flow control

4.4 System VARIABLEs and Buffers

4.4.1 Variables

Note that FENCE, DP and VOC-LINK must be declared in that order.

WIDTH	maximum target name size
FENCE	protected dictionary
DP	dictionary pointer
VOC-LINK	links vocabularies
RP	Harvard targets only. The equivalent of DP for DATA space.
SCR	If BLOCKS? true; for mass storage
BLK	If BLOCKS? true; user input dev: 0 for keyboard, >0 for block
CURRENT	Vocabulary/wordlist in which to put new definitions
STATE	Interpreting=0 or compiling=-1

CSP	Preserved stack pointer for compile time error checking
CONTEXT	Search order array
LAST	Points to name field of last definition
#THREADS	Default number of threads in new wordlists

4.5 Deferred words

`defer NUMBER?  \ addr -- d/n/- 2/1/0`
Attempt to convert the counted string at 'addr' to an integer. The return result is either 0 for failed, 1 for a single-cell return result (followed by that cell) or 2 for a double-cell return. The ASCII number string supplied can also contain implicit radix over-rides. A leading $ enforces hexadecimal, a leading # enforces decimal and a leading % enforces binary. Hexadecimal numbers can can also be specified by a leading '0x' or trailing 'h'. When one of the floating point packs is compiled, the action of `NUMBER?` is changed.

`defer ERROR  \ n -- ; error handler`
The standard error handler reports error n. If the system is loading, the offending line will be displayed. Now implemented by default as a synonym for `THROW`. Removed from v6.2 onwards. Use `THROW` instead.

4.6 Predefined Vocabularies

FORTH	Is the standard general purpose vocabulary
ROOT	This vocabulary holds the bare minimum functions

4.7 Vectored I/O handling

4.7.1 Introduction

The standard console Forth I/O words (`KEY?`, `KEY`, `EMIT`, `TYPE` and `CR`) can be used with any I/O device by placing the address of a table of xts in the `USER` variables `IPVEC` and `OPVEC`. `IPVEC` (input vector) controls the actions of `KEY?` and `KEY`, and `OPVEC`(output vector) controls the actions of `EMIT`, `TYPE` and `CR`. Adding a new device is matter of writing the five primitives, building the table, and storing the address of the table in the pointers `IPVEC` and `OPVEC` to make the new device active. Any initialisation must be performed before the device is made active.

Note that for the output words (`EMIT`, `TYPE` and `CR`) the `USER` variable `OUT` is handled in the kernel before the funtion in the table is called.

4.7.2 Building a vector table

The example below is taken from an ARM implementation.

```
create Console1      \ -- addr
  ' serkey1i ,             \ -- char
  ' serkey?1i ,            \ -- flag
  ' seremit1 ,             \ char --
  ' sertype1 ,             \ c-addr len --
  ' serCR1 ,               \ --

Console1 opvec !  Console1 ipvec !
```

4.7.3 Generic I/O words

`: KEY?  \ -- flag ; check receive char`
Return true if a character is available at the current input device.

`: KEY  \ -- char ; receive char`
Wait until the current input device receives a character and return it.

`: EMIT  \ -- char ; display char`
Display char on the current I/O device. `OUT` is incremented before executing the vector function.

`: TYPE  \ caddr len -- ; display string`
Display/write the string on the current output device. Len is added to `OUT` before executing the vector function.

`: CR  \ -- ; display new line`
Perform the equivalent of a CR/LF pair on the current output device. `OUT` is zeroed. before executing the vector function.

`: TYPEC  \ caddr len -- ; display string`
Display/write the string from CODE space on the current output device. Len is added to `OUT` before executing the vector function. N.B. Harvard targets only. In non-Harvard targets, this is a synonym for `TYPE`.

`: SPACE  \ --`
Output a blank space (ASCII 32) character.

`: SPACES  \ n --`
Output 'n' spaces, where 'n' > 0. If 'n' < 0, no action is taken.

`: FlushKeys  \ --`
Compiled for 32 bit systems to flush any pending input that might be returned by `KEY`.

`: SetConsole  \ device --`
Sets `KEY` and `EMIT` and friends to use the given device for terminal I/O. Compiled for 32 bit systems, but is also part of *LIBRARY.FTH*.

`: [io  \ -- ; R: -- ipvec opvec`
Save the current I/O devices on the return stack.

`: io]  \ -- ; R: ipvec opvec --`
Restore the I/O devices from the return stack.

4.8 String and memory operations

Some of these words may be coded for performance. If they are predefined, the high level versions will not be compiled.

For byte-addressed CPUs (nearly all except DSPs) this kernel assumes that a character is an 8 bit byte, i.e. that:

```
char = byte = address-unit
```

`: PLACE  \ c-addr1 u c-addr2 -- ; copies uncounted string to counted`
Place the string c-addr1/u as a counted string at c-addr2.

`: BOUNDS  \ addr len -- addr+len addr`

Modify the address and length parameters to provide an end-address and start-address pair suitable for a `DO ... LOOP` construct.

`: UPC \ char -- char'`
If char is in the range 'a' to 'z' convert it to upper case. `UPC` is English language specific.

`: UPPER \ c-addr u --`
Convert the ASCII string described to upper-case. This operation happens in place. Note that this word is language specific and is written to handle English only.

`: ERASE \ a-addr u --`
Erase U bytes of memory from A-ADDR with 0.

`: BLANK \ a-addr u --`
Blank U bytes of memory from A-ADDR using ASCII 32 (space).

4.9 Dictionary management

`: HERE \ -- addr`
Return the current dictionary pointer which is the first address-unit of free space within the system.

`: ALLOT \ n --`
Allocate N address-units of data space from the current value of `HERE` and move the pointer.

`: aligned \ addr -- addr'`
Given an address pointer this word will return the next `ALIGNED` address subject to system wide alignment restrictions.

`: ALIGN \ --`
`ALIGN` dictionary pointer using the same rules as `ALIGNED`.

`: LATEST \ -- c-addr`
Return the address of the name field of the last definition.

`: SMUDGE \ --`
Toggle the `SMUDGE` bit of the latest definition.

`: , \ x --`
Place the CELL value X into the dictionary at `HERE` and increment the pointer.

`: W, \ w --`
Place the WORD value X into the dictionary at `HERE` and increment the pointer. This word is not present on 16 bit implementations.

`: C, \ char --`
Place the CHAR value into the dictionary at `HERE` and increment the pointer.

`: there \ -- addr`
Harvard targets only: Return the DATA space pointer.

`: allot-ram \ n --`
Harvard targets only: `ALLOT` DATA space.

`: c,(r) \ b --`
Harvard targets only: The equivalent of `C,` for DATA space.

`: ,(r) \ n --`
Harvard targets only: The equivalent of `,` for DATA space.

`: N>LINK \ a-addr -- a-addr'`

Move a pointer from a NFA field to the Link Field.

```
: LINK>N        \ a-addr -- a-addr'
```
The inverse of N>LINK.

```
: >LINK         \ a-addr -- a-addr'
```
Move a pointer from an XT to the link field address.

```
: LINK>         \ a-addr -- a-addr'
```
The inverse of >LINK.

```
: >VOC-LINK     \ wid -- a-addr
```
Step from a wordlist identifier, wid, to the address of the field containg the address of the previously defined wordlist.

```
: >#THREADS     \ wid -- a-addr ; for XC5 compatibility
```
Step from a wordlist identifier, wid, to the address of the field containg the number of threads in the wordlist.

```
: >THREADS      \ wid -- a-addr
```
Step from a wordlist identifier, wid, to the address of the array containing the top NFA for each thread in the wordlist.

```
: >VOCNAME      \ wid -- a-addr
```
Step from a wordlist identifier, wid, to the address of the field pointing to the vocabulary name field.

```
: FIND          \ c-addr -- c-addr 0|xt 1|xt -1
```
Perform the SEARCH-WORDLIST operation on all wordlists within the current search order. This definition takes a counted string rather than a *c-addr/u* pair. The counted string is returned as well as the 0 on failure.

```
: .NAME         \ nfa --
```
The correct way to display a definition's name given an NFA. string for a word name, return the address of the dictionary name thread that will contain the name.

```
: makeheader    \ c-addr len --
```
Given a word name as a string in *addr/len* form, build a dictionary header for the word.

```
: $CREATE       \ c-addr --
```
Perform the action of CREATE (below) but take the name from a counted string. OBSOLETE: replace by:

```
 count makeheader docreate,
```

```
: CREATE        \ --
```
Create a new definition in the dictionary. When the new definition is executed it will return the address of the definition BODY.

4.10 String compilation

```
: (C")          \ -- c-addr
```
The run-time action for C" which returns the address of and steps over a counted string.

```
: (S")          \ -- c-addr u
```
The run-time action for S" which returns the address and length of and steps over a string.

```
: (ABORT")      \ i*x x1 -- | i*x
```
The run time action of ABORT".

`: (.")  \ --`
The run-time action of `."`.

4.11 Pre-ANS Exception handlers

Before the ANS Forth standard, these words were the primary error handlers. They are provided for compatibility, but wherever possible, the use of `CATCH` and `THROW` will be found to be more flexible.

`: ABORT  \ i*x -- ; R: j*x --`
Performs `-1 THROW`. This is a compatibility word for earlier versions of the kernel. Unfortunately, the earlier versions gave problems when `ABORT` was used in interrupt service routines or tasks. The new definition is brutal but consistent.

`: ABORT"  \ Comp: "ccc<quote>" -- ; Run: i*x x1 -- | i*x ; R: j*x -- | j*x`
If x1 is non-zero at run-time, store the address of the following counted string in `USER` variable `'ABORTTEXT`, and perform `-2 THROW`. The text interpreter in `QUIT` will (if reached) display the text.

`: (Error)  \ n --`
The default action of `ERROR`. This definition has been removed from v6.2 onwards. See the section about the changes from v6.1 to v6.2.

`: ?ERROR  \ flag n --`
If flag is true, perform "n ERROR", otherwise do nothing. This definition has been removed from v6.2 onwards. See the section about the changes from v6.1 to v6.2.

4.12 ANS words CATCH and THROW

`CATCH` and `THROW` form the basis of all Forth error handling. The following description of `CATCH` and `THROW` originates with Mitch Bradley and is taken from an ANS Forth standard draft.

`CATCH` and `THROW` provide a reliable mechanism for handling exceptions, without having to propagate exception flags through multiple levels of word nesting. It is similar in spirit to the "non-local return" mechanisms of many other languages, such as C's **setjmp()** and **longjmp()**, and LISP's **CATCH** and **THROW**. In the Forth context, `THROW` may be described as a "multi-level EXIT", with `CATCH` marking a location to which a `THROW` may return.

Several similar Forth "multi-level EXIT" exception-handling schemes have been described and used in past years. It is not possible to implement such a scheme using only standard words (other than `CATCH` and `THROW`), because there is no portable way to "unwind" the return stack to a predetermined place.

`THROW` also provides a convenient implementation technique for the standard words `ABORT` and `ABORT"`, allowing an application to define, through the use of `CATCH`, the behavior in the event of a system `ABORT`.

4.12.1 Example implementation

This sample implementation of `CATCH` and `THROW` uses the non-standard words described below. They or their equivalents are available in many systems. Other implementation strategies, including directly saving the value of `DEPTH`, are possible if such words are not available.

`SP@` (– addr) returns the address corresponding to the top of data stack.

`SP!` (addr –) sets the stack pointer to addr, thus restoring the stack depth to the same depth that existed just before addr was acquired by executing SP@.

`RP@` (– addr) returns the address corresponding to the top of return stack.

`RP!` (addr –) sets the return stack pointer to addr, thus restoring the return stack depth to the same depth that existed just before addr was acquired by executing RP@.

```
nnn USER HANDLER   0 HANDLER !  \ last exception handler
: CATCH  ( xt -- exception# | 0 ) \ return addr on stack
        SP@ >R  ( xt )  \ save data stack pointer
        HANDLER @ >R    ( xt )  \ and previous handler
        RP@ HANDLER !   ( xt )  \ set current handler
        EXECUTE ( )     \ execute returns if no THROW
        R> HANDLER !    ( )     \ restore previous handler
        R> DROP ( )     \ discard saved stack ptr
        0       ( 0 )   \ normal completion
;
: THROW  ( ??? exception# -- ??? exception# )
        ?DUP IF ( exc# )        \ 0 THROW is no-op
                HANDLER @ RP!   ( exc# )        \ restore prev return stack
                R> HANDLER !    ( exc# )        \ restore prev handler
                R> SWAP >R      ( saved-sp ) \ exc# on return stack
                SP! DROP R>     ( exc# )        \ restore stack
                \       Return to the caller of CATCH because return
                \       stack is restored to the state that existed
                \       when CATCH began execution
        THEN
;
```

The ROM PowerForth implementation is similar to the one described above, but not identical.

4.12.2 Example use

If `THROW` is executed with a non zero argument, the effect is as if the corresponding `CATCH` had returned it. In that case, the stack depth is the same as it was just before `CATCH` began execution. The values of the i*x stack arguments could have been modified arbitrarily during the execution of xt. In general, nothing useful may be done with those stack items, but since their number is known (because the stack depth is deterministic), the application may `DROP` them to return to a predictable stack state.

Typical use:

```
: could-fail    \ -- char
  KEY DUP [CHAR] Q =
  IF  1 THROW  THEN
;

: do-it         \ a b -- c
  2DROP could-fail
;

: try-it                \ --
  1 2 ['] do-it CATCH IF
    ( -- x1 x2 ) 2DROP ." There was an exception" CR
  ELSE
    ." The character was " EMIT CR
  THEN
;

: retry-it              \ --
  BEGIN
    1 2 ['] do-it CATCH
   WHILE
    ( -- x1 x2 ) 2DROP  ." Exception, keep trying" CR
  REPEAT ( char )
  ." The character was " EMIT CR
;
```

4.12.3 Gotchas

If a `THROW` is performed without a `CATCH` in place, the system will/may crash. As the current exception frame is pointed to by the `USER` variable `HANDLER`, each task and interrupt handler will need a `CATCH` if `THROW` is used inside it.

You can no longer use `ABORT` as a way of resetting the data stack and calling `QUIT`. `ABORT` is now defined as `-1 THROW`.

4.12.4 User words

`: CATCH  \ i*x xt -- j*x 0|i*x n`
Execute the code at XT with an exception frame protecting it. `CATCH` returns a 0 if no error has occurred, otherwise it returns the throw-code passed to the last `THROW`.

`: THROW  \ k*x n -- k*x|i*x n`
Throw a non-zero exception code n back to the last `CATCH` call. If n is 0, no action is taken except to `DROP` n.

`: ?throw  \ flag throw-code -- ; SFP017`
Perform a `THROW` of value throw-code if flag is non-zero, otherwise do nothing except discard flag and throw-code.

4.13 Formatted and unformatted i/o

4.13.1 Setting number bases

`: HEX  \ --`
Change current radix to base 16.

`: DECIMAL \ --`
Change current radix to base 10.

`: OCTAL \ --`
Change current radix to base 8. 32 bit targets only.

`: BINARY \ --`
Change current radix to base 2.

4.13.2 Numeric output

`: HOLD \ char --`
Insert the ascii 'char' value into the pictured numeric output string currently being assembled.

`: SIGN \ n --`
Insert the ascii 'minus' symbol into the numeric output string if 'n' is negative.

`: # \ ud1 -- ud2`
Given a double number on the stack this will add the next digit to the pictured numeric output buffer and return the next double number to work with. PLEASE NOTE THAT THE NUMERIC OP STRING IS BUILT FROM RIGHT (l.s. ddigit) to LEFT (m.s. digit).

`: #S \ ud1 -- ud2`
Keep performing `#` until all digits are generated.

`: <# \ --`
Begin definition of a new numeric output string buffer.

`: #> \ xd -- c-addr u`
Terminate defnition of a numeric output string. Return the address and length of the ASCII string.

`: -TRAILING \ c-addr u1 -- c-addr u2`
Modify a string address/length pair to ignore any trailing spaces.

`: D.R \ d n --`
Output the double number 'd' using current radix, right justified to 'n' characters. Padding is inserted using spaces on the left side.

`: D. \ d --`
Output the double number 'd' without padding.

`: . \ n --`
Output the cell signed value 'n' without justification.

`: U. \ u --`
As with . but treat as unsigned.

`: U.R \ u n --`
As with D.R but uses a single-unsigned cell value.

`: .R \ n1 n2 --`
As with D.R but uses a single-signed cell value.

4.13.3 Numeric input

`: SKIP-SIGN \ addr1 len1 -- addr2 len2 t/f ; true if sign=negative`
Inspect the first character of the string, if it is a '+' or '-' character, step over the string. Returning true if the character was a '-', otherwise return false.

`: +DIGIT \ d1 n -- d2 ; accumulates digit into double accumulator`

Multiply d1 by the current radix and add n to it.

```
: +CHAR          \ char -- flag ; true if ok
```

This routine handles non-numeric characters, returning true for valid characters. By default, the only acceptable non-numeric character is the double-number separator ','.

```
: +ASCII-DIGIT  \ d1 char -- d2 flag ; true=ok
```

Accumulate the double number d1 with the conversion of char, returning true if the character is a valid digit or part of an integer.

```
: (INTEGER?)     \ c-addr u -- d/n/- 2/1/0
```

The guts of `INTEGER?` but without the base override handling. See `INTEGER?`

```
: Check-Prefix  \ addr len -- addr' len'
```

If any `BASE` override prefices or suffices are used in the input string, set `BASE` accordingly and return the string without the override characters.

```
: Integer?       \ $addr -- n 1 | d 2 | 0
```

Attempt to convert the counted string at 'addr' to an integer. The return result is either 0 for failed, 1 for a single-cell return result followed by that cell)or 2 for a double return. The ASCII number string supplied can also contain implicit radix over-rides. A leading $ enforces hexadecimal, a leading # enforces decimal and a leading % enforces binary. The prefix '@' is supported for octal numbers in 32 bit systems, for which hexadecimal numbers can also be specified by a leading '0x' or a trailing 'h'.

```
: >NUMBER        \ ud1 c-addr1 u1 -- ud2 c-addr2 u2 ; convert all until non-digits
```

Accumulate digits from string c-addr1/u2 into double number ud1 to produce ud2 until the first non-convertible character is found. c-addr2/u2 represents the remaining string with c-addr2 pointing the non-convertible character. The number base for conversion is defined by the contents of `USER` variable `BASE`. `>NUMBER` is now case insensitive.

4.14 String input and output

```
: BS             \ -- ; destructive backspace
```

Perform a destructive backspace by issuing ASCII characters 8, 20h, 8. If `OUT` is non-zero at the start, it is decremented by one regardless of the actions of the device driver.

```
: ?BS            \ pos -- pos' step ; perform BS if pos non-zero
```

If pos is non-zero and `ECHOING` is set, perform `BS` and return the size of the step, 0 or -1.

```
: SAVE-CH        \ char addr -- ; save as required
```

Save char at addr, and output the character if `ECHOING` is set.

```
: ."             \ "ccc<quote>" --
```

Output the text upto the closing double-quotes character. Use `.( <text>)` when interpreting.

```
: $.             \ c-addr -- ; display counted string
```

Output a counted-string to the output device. Note that on Harvard targets (e.g. 8051) c-addr is in DATA space.

```
: ACCEPT         \ c-addr +n1 -- +n2 ; read up to LEN chars into ADDR
```

Read a string of maximum size n1 characters to the buffer at c-addr, returning n2 the number of characters actually read. Input may be terminated by CR. The action may be input device specific. If `ECHOING` is non-zero, characters are echoed. If `XON/XOFF` is non-zero, an XON character is sent at the start and an XOFF character is sent at the the end.

4.15 Source input control

`0 value SOURCE-ID \ -- n ; indicates input source`
Returns an indicator of which device is generating source input. See the ANS specification for more details.

`: TIB \ -- c-addr ; return address of terminal i/p buffer`
Returns the address of the terminal input buffer. Note that tasks requiring user input must initialise the `USER` variable `'TIB`. New code should use `SOURCE` and `TO-SOURCE` instead for ANS Forth compatibility.

`: TO-SOURCE \ c-addr u --`
Set the address and length of the system terminal input buffer. These are held in the user variables `'TIB` and `#TIB`.

`: SOURCE \ -- c-addr u`
Returns the address and length of the current terminal input buffer.

`: SAVE-INPUT \ -- xn..x1 n`
Save all the details of the input source onto the data stack. will do the job. If you want to move the data to the return stack, `N>R` and `NR>` are available in some 32 bit implementations.

`: RESTORE-INPUT \ xn..x1 n -- flag`
Attempt to restore input specification from the data stack. If the stack picture between `SAVE-INPUT` and `RESTORE-INPUT` is not balanced, a non-zero is returned in place of N. On success a 0 is returned.

`: QUERY \ -- ; fetch line into TIB`
Reset the input source specification to the console and accept a line of text into the input buffer.

`: REFILL \ -- flag ; refill input source`
Attempt to refill the terminal input buffer from the current source. This may be a file or the console. An attempt to refill when the input source is a string will fail. The return result is a flag indicating success with TRUE and failure with FALSE. A failure to refill when the input source is a text file indicates the end of file condition.

4.16 Text scanning

`: PARSE \ char "ccc<char>" -- c-addr u`
Parse the next token from the terminal input buffer using `<char>` as the delimiter. The next token is returned as a `c-addr/u` string description. Note that `PARSE` does not skip leading delimiters. If you need to skip leading delimiters, use `PARSE-WORD` instead.

`: PARSE-WORD \ char -- c-addr u ; find token, skip leading chars`
An alternative to `WORD` below. The return is a *c-addr/u* pair rather than a counted string and no copy has occured, i.e. the contents of `HERE` are unaffected. Because no intermediate global buffers are used `PARSE-WORD` is more reliable than `WORD` for text scanning in multi-threaded applications.

`: WORD \ char "<chars>ccc<char>" -- c-addr`
Similar behaviour to the ANS word `PARSE` but the returned string is described as a counted string.

4.17 Miscellaneous

`: HALT? \ -- flag`
Used in listed displays. This word will check the keyboard for a 'pause' key `<space>`, if the key is pressed it will then wait for a continue key or an abort key. The return flag is TRUE if abort is requested. Line Feed (LF, ASCII 10) characters are ignored.

`: origin-  \ addr -- addr'`
If addr is non-zero, subtract the start address of the first defined CDATA section. This word is only compiled if the start address of the first defined CDATA section is non-zero.

`: origin+  \ addr -- addr' ; denormalise NFA again`
If addr is non-zero, add the start address of the first defined CDATA section. This word is only compiled if the start address of the first defined CDATA section is non-zero.

`: nfa-buff  \ -- addr+len addr ; make a buffer for holding NFAs`
Form a temporary buffer for holding NFAs. A factor for `WORDS`.

`: MAX-NFA  \ -- addr c-addr ; returns addr and top nfa`
Return the thread address and NFA of the highest word in the NFA buffer. A factor for `WORDS`.

`: COPY-THREADS  \ addr --`
Copy the threads of the `CONTEXT` wordlist to a temporary NFA buffer for manipulation. A factor for `WORDS`.

`: WORDS  \ --`
Display the names of all definitions in the wordlist at the top of the search-order.

`: MOVE  \ src dest len -- ; intelligent move`
An intelligent memory move, chooses between `CMOVE` and `CMOVE>` at runtime to avoid memory overlap problems. Note that as ROM PowerForth characters are 8 bit, there is an implicit connection between a byte and a character.

`: DEPTH  \ -- +n`
Return the number of items on the data stack, excluding the count.

`: UNUSED  \ -- u ; free dictionary space`
Return the number of bytes free in the dictionary.

`: .FREE  \ --`
Return the free dictionary space.

4.18 Wordlist control

`: WORDLIST  \ -- wid`
Create a new wordlist and return a unique identifier for it.

`: VOCABULARY  \ -- ; VOCABULARY <name>`
Create a `VOCABULARY` which is implemented as a named wordlist.

`: FORTH  \ --`
Install `FORTH` wordlist into search-order.

`: FORTH-WORDLIST  \ -- wid`
Return the unique WID for the main FORTH wordlist.

`: GET-CURRENT  \ -- wid`
Return the WID for the Wordlist which holds any definitions made at this point.

`: SET-CURRENT  \ wid --`
Change the wordlist which will hold future definitions.

`: GET-ORDER  \ -- widn...wid1 n`
Return the list of WIDs which make up the current search-order. The last value returned on top-of-stack is the number of WIDs returned.

`: SET-ORDER  \ widn...wid1 n -- ; unless n = -1`

Set the new search-order. N is the number of WIDs to place in the search-order. If N is -1 then the minimum search order is inserted.

`: ONLY \ --`
Set the minimum search order as the current search-order.

`: ALSO \ --`
Duplicate the first WID in the search order.

`: PREVIOUS \ --`
Drop the current top of search-order.

`: DEFINITIONS \ --`
Set the current top WID of search-order as the current definitions wordlist.

4.19 Control structures

`: ?PAIRS \ x1 x2 --`
If x1<>x2, issue and error. Used for on-target compile-time error checking.

`: !CSP \ x --`
Save the stack pointer in `CSP`. Used for on-target compile-time error checking.

`: ?CSP \ --`
Issue an error if the stack pointer is not the same as the value previously stored in CSP. Used for on-target compile-time error checking.

`: ?COMP \ --`
Error if not in compile state.

`: ?EXEC \ --`
Error if not interpreting.

`: DO \ C: -- do-sys ; Run: n1|u1 n2|u2 -- ; R: -- loop-sys`
Begin a `DO ... LOOP` construct. Takes the end-value and start-value from the data-stack.

`: ?DO \ C: -- do-sys ; Run: n1|u1 n2|u2 -- ; R: -- | loop-sys`
Compile a `DO` which will only begin loop execution if the loop parameters are not the same. Thus `0 0 ?DO ... LOOP` will not execute the contents of the loop.

`: LOOP \ C: do-sys -- ; Run: -- ; R: loop-sys1 -- | loop-sys2`
The closing statement of a `DO..LOOP` construct. Increments the index and terminates when the index crosses the limit.

`: +LOOP \ C: do-sys -- ; Run: n -- ; R: loop-sys1 -- | loop-sys2`
As with `LOOP` except that you specify the increment on the data-stack.

`: BEGIN \ C: -- dest ; Run: --`
Mark the start of a structure of the form:

```
BEGIN..[while]..UNTIL / AGAIN / [REPEAT]
```

`: AGAIN \ C: dest -- ; Run: --`
The end of a `BEGIN..AGAIN` construct which specifies an infinite loop.)

`: UNTIL \ C: dest -- ; Run: x --`
Compile code into definition which will jump back to the matching `BEGIN` if the supplied condition flag is Zero/FALSE.

`: WHILE \ C: dest -- orig dest ; Run: x --`
Separate the condition test from the loop code in a `BEGIN..WHILE..REPEAT` block.

`: REPEAT \ C: orig dest -- ; Run: --`
Loop back to the conditional dest code in a `BEGIN..WHILE..REPEAT` construct.)

`: IF \ C: -- orig ; Run: x --`
Mark the start of an `IF..[ELSE]..THEN` conditional block.

`: THEN \ C: orig -- ; Run: --`
Mark the end of an `IF..THEN` or `IF..ELSE..THEN` conditional construct.

`: endif \ C: orig -- ; Ru: -- ; synonym for THEN`
An alias for `THEN`. Note that ANS Forth describes `THEN` not `ENDIF`.

`: AHEAD \ C: -- orig ; Run: --`
Start an unconditional forward branch which will be resolved later.

`: ELSE \ C: orig1 -- orig2 ; Run: --`
Begin the failure condition code for an `IF`.

`: CASE \ C: -- case-sys ; Run: --`
Begin a `CASE..ENDCASE` construct. Similar to C's **switch**.

`: OF \ C: -- of-sys ; Run: x1 x2 -- | x1`
Begin conditional block for `CASE`, executed when the switch value is equal to the X2 value placed in TOS.

`: ENDOF \ C: case-sys1 of-sys -- case-sys2 ; Run: --`
Mark the end of an OF conditional block within a `CASE` construct. Compile a jump past the `ENDCASE` marker at the end of the construct.

`: ENDCASE \ C: case-sys -- ; Run: x --`
Terminate a `CASE..ENDCASE` construct. `DROP`s the switch value from the stack.

`: ?OF \ C: -- of-sys ; Run: flag --`
Begin conditional block for `CASE`, executed when the flag is true.

`: END-CASE \ C: case-sys -- ; Run: --`
A Version of `ENDCASE` which does not drop the switch value. Used when the switch value itself is consumed by a DEFAULT condition.

`: NEXTCASE \ C: case-sys -- ; Run: x --`
Terminate a `CASE..NEXTCASE` construct. `DROP`s the switch value from the stack and compiles a branch back to the top of the loop at `CASE`.

`: RECURSE \ Comp: --`
Compile a recursive call to the colon definition containing `RECURSE` itself. Do not use `RECURSE` between `DOES>` and `;`. Used in the form:

```
: foo  ... recurse ... ;
```

to compile a reference to `FOO` from inside `FOO`.

4.20 Target interpreter and compiler

`: ?STACK \ --`
Error if stack pointer out of range.

`: ?UNDEF \ x --`
Word not defined error if x=0.

`: POSTPONE \ Comp: "<spaces>name" --`

Compile a reference to another word. POSTPONE can handle compilation of IMMEDIATE words which would otherwise be executed during compilation.

```
: S"            \ Comp: "ccc<quote>" -- ; Run: -- c-addr u
```
Describe a string. Text is taken upto the next double-quote character. The address and length of the string are returned.

```
: C"            \ Comp: "ccc<quote>" -- ; Run: -- c-addr
```
As S" except the address of a counted string is returned.

```
: #LITERAL      \ n1 .... nn n -- ; put in dictionary n1 first
```
Compile n1..nn as literals so that the same stack order results when the code executes. This word will be removed in a future release. INTERNAL.

```
: LITERAL       \ Comp: x -- ; Run: -- x
```
Compile a literal into the current definition. Usually used in the form [<expression] LITERAL inside a colon definition. Note that LITERAL is IMMEDIATE.

```
: 2LITERAL      \ Comp: x1 x2 -- ; Run: -- x1 x2
```
A two cell version of LITERAL.

```
: CHAR          \ "<spaces>name" -- char
```
Return the first character of the next token in the input stream. Usually used to avoid magic numbers in the source code.

```
: [CHAR]        \ Comp: "<spaces>name" -- ; Run: -- char
```
Compile the first character of the next token in the input stream as a literal. Usually used to avoid magic numbers in the source code.

```
: sliteral      \ c-addr u -- ; Run: -- c-addr2 u ; 17.6.1.2212
```
Compile the string c-addr1/u into the dictionary so that at run time the identical string c-addr2/u is returned. Note that because of the use of dynamic strings at compile time the address c-addr2 is unlikely to be the same as c-addr1.

```
: [             \ --
```
Switch compiler into interpreter state.

```
: ]             \ --
```
Switch compiler into compilation state.

```
: IMMEDIATE     \ --
```
Mark the last defined word as IMMEDIATE. Immediate words will execute whenever encountered regardless of STATE.

```
: '             \ "<spaces>name" -- xt
```
Find the xt of the next word in the input stream. An error occurs if the xt cannot be found.

```
: [']           \ Comp: "<spaces>name" -- ; Run: -- xt
```
Find the xt of the next word in the input stream, and compile it as a literal. An error occurs if the xt cannot be found.

```
: [COMPILE]     \ "<spaces>name" --
```
Compile the next word in the input stream. [COMPILE] ignores the IMMEDIATE state of the word. Its operation is mostly superceded by POSTPONE.

```
: (             \ "ccc<paren>" --
```
Begin an inline comment. All text upto the closing bracket is ignored.

```
: \             \ "ccc<eol>" --
```

Begin a single-line comment. All text up to the end of the line is ignored.

`: ",  \ "ccc<quote>" --`
Parse text up to the closing quote and compile into the dictionary at `HERE` as a counted string. The end of the string is aligned.

`: .(  \ "cc<paren>" --`
A documenting comment. Behaves in the same manner as `(` except that the enclosed text is written to the console at compile time.

`: ASSIGN  \ "<spaces>name" --`
A state smart word to get the XT of a word. The source word is parsed from the input stream. Used as part of an `ASSIGN xxx TO-DO yyy` construct.

`: (TO-DO)  \ -- ; R: xt -- a-addr'`
The run-time action of `TO-DO`. It is followed by the data address of the `DEFER`red word at which the xt is stored.

`: TO-DO  \ "<spaces>name" --`
The second part of the `ASSIGN xxx TO-DO yyy` construct. This word will assign the given XT to be the action of a `DEFER`ed word which is named in the input stream.

`: exit  \ R: nest-sys -- ; exit current definition`
Compile code into the current definition to cause a definition to terminate. This is the Forth equivalent to inserting an RTS/RET instruction in the middle of an assembler subroutine.

`: ;  \ C: colon-sys -- ; Run: -- ; R: nest-sys --`
Complete the definition of a new 'colon' word or `:NONAME` code block.

`: INTERPRET  \ --`
Process the current input line as if it is text entered at the keyboard.

`: N>R  \ xn .. x1 N -- ; R: -- x1 .. xn n`
Transfer N items and count to the return stack.

`: NR>  \ -- xn .. x1 N ; R: x1 .. xn N --`
Pull N items and count off the return stack.

`: EVALUATE  \ i*x c-addr u -- j*x ; interpret the string`
Process the supplied string as though it had been entered via the interpreter.

`: .throw  \ throw# --`
Display the throw code. Values of 0 and -1 are ignored.

`: QUIT  \ -- ; R: i*x --`
Empty the return stack, store 0 in `SOURCE-ID`, and enter interpretation state. `QUIT` repeatedly `ACCEPT`s a line of input and `INTERPRET`s it, with a prompt if interpreting and `ECHOING` is on. Note that any task that uses `QUIT` must initialise `'TIB`, `BASE`, `IPVEC`, and `OPVEC`.

4.20.1 Compilation and Caches

Because some CPUs, e.g. XScale and ARM9s, have separate instruction and data caches, self-modifying code can cause problems when code is laid down (into the data cache) and then an attempt is made to execute it (the instruction cache will not necessarily contain the code). For this reason a word is provided that will synchronise the caches. This word is CPU specific and may reference code in a CPU and/or hardware specific file.

Synchronisation will usually only be necessary when creating words, constants, variables etc.

interactively on the target and then executing them before the code has reached the instruction cache. Only executable code has to be synchronised, not data.

If the word `FLUSHCACHE ( -- )` is provided before *kernel62.fth* is compiled, it will be executed by the text interpreter before each line is processed. `FLUSHCACHE` is also executed by `;`.

4.21 Startup code

4.21.1 Cold chain

If enabled by the non-zero equate `COLDCHAIN?` the cold start code in `COLD` will walk a list and execute the xts contained in it. The xts must have no stack effect `( -- )` and are added to the list by the phrase:

```
' <wordname> AtCold
```

The list is executed in the order in which it was defined so that the last word added is executed last. This was done for compatibility with VFX Forth, which also contains a shutdown chain, in which the last word added is executed first.

If the equate `COLDCHAIN?` is not defined in the control file, a default value of 0 will be defined.

`l: ColdChainFirst` `\ -- addr`

Dummy first entry in ColdChain.

`variable ColdChain` `\ -- addr`

Holds the address of the last entry in the cold chain.

`: AtCold` `\ xt --`

Specifiy a new XT to execute when `COLD` is run. Note that the last word added is executed last. `ATCOLD` can be executed interpretively during cross-compilation. The cold chain is built in the current `CDATA` section.

`: WalkColdChain` `\ --` `MPE.0000`

Execute all words added to the cold chain. Note that the first word added is executed first.

4.21.2 The COLD sequence

At power up, the target executes `COLD` or the word specified by `MAKE-TURNKEY <name>`.

`: copyRAMtab` `\ addr --`

Copy the given RAM table from Flash into RAM. The table may initialise multiple blocks of RAM.

`: (INIT)` `\ --`

Performs the high level Forth startup. See the source code for more details.

`: COLD` `\ --`

The first high level word executed by default. This word is set to be the word executed at power up, but this may be overridden by a later use of `MAKE-TURNKEY <name>`. See the source code for more details of `COLD`.

4.22 Kernel error codes

-1	ABORT
-2	ABORT"
-4	Stack underflow
-13	Undefined word.
-14	Attempt to interpret a compile only definition.
-22	Control structure mismatch - unbalanced control structure.
-121	Attempt to remove with MARKER or FORGET below FENCE in protected dictionary.
-403	Attempt to compile an interpret only definition.
-501	Error if not LOADing from a block.

4.23 Differences between the v6.1 and 6.2 kernels

4.23.1 Error handling

All error handling in the v6.2 kernel is defined in terms of CATCH and THROW. The earlier words ERROR and ?ERROR have been removed. If you need them, define them as synonyms for THROW and ?THROW.

The definition of ABORT has changed significantly. The old version was:

```
: ABORT         \ i*x -- ; R: j*x --
\ *G Empty the data stack and perform the action of QUIT, which includes
\ ** emptying the return stack, without displaying a message.
  xon/xoff off  echoing on              \ No Xon/Xoff, do Echo
  s0 @ sp!                              \ reset data stack
  quit                                  \ start text interpreter
;
```

The new version is:

```
: ABORT         \ i*x -- ; R: j*x --
\ *G Performs "-1 THROW". This is a compatibility word for earlier
\ ** versions of the kernel. Unfortunately, the earlier versions
\ ** gave problems when ABORT was used in interrupt service routines
\ ** or tasks. The new definition is brutal but consistent.
  -1 Throw
;
```

The old version worked 99% of the time, except that in tasks or interrupt service routines, the result was unpredictable. Because modern applications are larger and more complex, ABORT has to be completely predictable. The line

```
  xon/xoff off  echoing on              \ No Xon/Xoff, do Echo
```

is now part of QUIT. The phrase SO @ SP! must now be provided by the THROW handler.

The previous definition of THROW checked for a previously defined CATCH and performed the old ABORT if no CATCH had been defined. The new version assumes that a CATCH has been defined and may/will crash if no CATCH has been performed. The result is a faster and smaller definition of CATCH. However, it is now the programmer's responsibility to provide a CATCH handler for ALL ISRs and tasks that may generate a THROW. This is actually very little different from the previous situation, except that the system is less forgiving if you forget to provide a handler.

Error codes have been made ANS compliant. It is MPE policy that all error and ior (i/o result) codes shall be distinct from now on.

4.23.2 Terminal input buffer and ACCEPT

The changes below simplifiy the source code, and permit multiple tasks to use EVALUATE without interaction. Note that compilation from multiple sources/tasks requires the interpreter/compiler to be interlocked with a semaphore.

The 2VARIABLE SOURCE-STRING has been removed, and TO-SOURCE and SOURCE use 'TIB and #TIB instead.

The state variables ECHOING and XON/XOFF are now USER variables. In most cases this will have no impact. However, tasks may now control these variables independently.

QUIT always enforces ECHOING on and disables XON/XOFF processing. QUIT does not select an I/O device. This change was made to allow the interpreter to be used on any channel in systems with several serial lines or with the Telnet service of the PowerNet TCP/IP stack. Note that any task that uses QUIT must initialise IPVEC, OPVEC, ECHOING and XON/XOFF.

Removed: ?EMIT and SOURCE-STRING.

5 Heap Memory Allocation

5.1 Heap definition

The heap is allocated from a predefined section of memory. Facilities are provided for user expansion of the heap to mass storage, although the current code makes no provision for page management. When the heap is initialised, a free block and an end block are created. The end block is of zero size, and is used only as a marker. The address returned by `ALLOCATE` and `RESIZE` is the address of the first data byte, as is the address consumed by `FREE`.

The heap **must** be initialised before use by calling `INIT-HEAP`. Heap access words return status=0 for success, and status<>0 for error.

Two equates are required during compilation to allocate a contiguous block of RAM for the heap.

```
STARTOFHEAP is the start address of the heap
SIZEOFHEAP is the size of the RAM for the heap
```

There are two versions of this code provided. *Heap32.fth* is provided for 32-bit targets and is optimised for the VFX code generator. *Heap16.fth* is for 16 bit targets, and is optimised for code density.

5.1.1 32 bit targets - HEAP32.FTH

The heap is controlled using a single cell per block. This information is used in two parts:

```
bits 31..24: $EE - End, $FF - Free, $AA - Allocated
bits 23..0: 24 bits for number of data bytes in block.
```

A consequence of this is that the maximum block size that can be allocated is 16Mb-1 bytes.

If you use a pre-emptive scheduler or need to use the heap routines inside interrupt routines, you must define suitable heap lock and unlock routines and set the equate `LOCKHEAP?` to non-zero.

`LockHeap=0`
: no heap locking

`LockHeap=1`
: heap locking by turning off interrupts

`LockHeap=2`
: heap locking by semaphore.

5.1.2 16 bit targets - HEAP16.FTH

The heap is controlled using two cells per block. This information is used in three parts:

```
cell = #bytes, number of bytes in this block
cell = flag, split between a four bit and a 12 bit field
```

The top four bits of the flag are used to indicate the block type, where $E = End, $F = Free, $A = Allocated. Others may be added later for type management.

The bottom 12 bits of the flag are currently unused, and should be set to zero.

5.2 Gotchas

The heap routines must be protected if they are to be used both in normal code and in interrupts. In this case the code must be modified to be interrupt safe, but this may have a significant impact on interrupt latency. Examples may be found in *heap32.fth*.

If you use a pre-emptive scheduler, you must also define suitable heap lock and unlock routines.

The equate `LOCKHEAP?` controls the usage:

`LockHeap=0`
: no heap locking

`LockHeap=1`
: heap locking by turning off interrupts

`LockHeap=2`
: heap locking by semaphore.

5.3 Glossary

The glossary does not include all the factors used in the code. If you are interested in the implementation, please read the sources.

`sizeofheap buffer: STARTOFHEAP  \ -- addr`
The heap memory is defined at compile time.

`STARTOFHEAP sizeofheap + equ ENDOFHEAP  \ -- addr`
The end+1 of the heap.

`: init-heap     \ --`
The heap is initialised by creating 2 blocks. Block 1 starts at the beginning and is marked as a free block. Block 2 Is a null marker at the end of heap space.

`: allocate      \ #bytes -- addr status`
Attempt to allocate some memory from the heap. Walk the heap looking for a single big enough block. If the block is larger than than required split it into two blocks. Allocate part or all of the free block. Status=0 for success.

`: free          \ addr -- status`
Attempt to free a heap block. Status=0 for success. If *addr* is zero, no action is taken and zero is returned.

`: resize        { *ptr newsize | currsize -- *newptr ior }`
Try to resize an allocated block to a new size, allowing for alignment. If the existing memory block is not big enough, the data will be copied to a new block, and the returned addr2 will not be the same as addr1. Status=0 for success.

`: size          \ *ptr -- currsize|-1`
Return the size of an allocated block or -1 if there's an error.

5.4 Diagnostics

```
: .heap          \ --
```
Walk the heap displaying block information.

```
: heapok?        \ -- t/f
```
Walk the heap and return TRUE if the heap is "well".

6 Target VALUE and local variables

The file *COMMON\METHODS.FTH* implements the compilation of `VALUE`s and the ANS Forth `LOCALS|` syntax for compilation on the target. Compilation of this file requires CPU dependent support, usually called *LOCAL.FTH* in the *%CpuDir%* directory, and MPE standard control files will compile these files if the equate `TARGET-LOCALS?` is set non-zero in the control file.

Note that this file is only provided for full ANS compliance. The MPE extended local variable syntax is provided by the cross compiler, and is much more powerful and more readable.

Note also that compilation of %CpuDir%\LOCAL.FTH may be required if you cross compile words with more than four input arguments.

`: OPERATOR \ n -- ; define an operator in the cross compiler`
An interpreter definition that build new operators such as "to" and "addr".

`: VALUE \ n -- ; -- n ; n VALUE <name>`
Creates a variable of initial value n that returns its contents when referenced. To store to a child of VALUE use "n to <child>".

`: (LOCAL) \ Comp: c-addr u -- ; Exec: -- x ; define local var`
When executed during compilation, defines a local variable whose name is given by c-addr/u. If u is zero, c-addr is ignored and compilation of local variables is assumed to finish. When the word containing the local variable executes, the local variable is initialised from the stack. When the local variable executes, its value is returned. The local variable may be written to by preceding its name with TO. This word is provided for the construction of user-defined local variable notations. This word is only provided for ANS compatibility, and locals created by it cannot be optimised by the VFX code generator.

`: LOCALS| \ "name...name |" --`
Create named local variables <name1> to <namen>. At run time the stack effect is (xn..x1 –), such that <name1> is initialised with x1 and <namen> is initialised with xn. Note that this means that the order of declaration is the reverse of the order used in stack comments! When referenced, a local variable returns its value. To write to a local, precede its name with TO.
In the example below, a and b are named inputs.

```
: foo      \ a b --
  locals| b a |
  a b +  cr .
  a b *  cr .
;
```

7 Exception and Interrupt handlers

The file *Cortex\IntCortex.fth* contains generic interrupt handlers for Cortex-M0/M1/M3/M4 processors. *IntCortex.fth* requires *Cortex\CortexDef* to be compiled before the *SFRxxxx* file for your particular CPU. The file *Cortex/StackDef.fth* provides default main task and stack layouts and should be compiled from the control file or copied into your control file. Default stack initialisation code is provided in *Cortex/StackDef.fth*.

The high-level interrupt handlers all share a common "stack of stacks". On entry to the interrupt handler, Forth system registers are allocated. Your initialisation code **must** set up R13. This is normally provided by the MPE wrapper code for each exception.

In order to support exception nesting the equate `#IRQs` in the control file must be set to the maximum number of nestings required. The equate `#IRQs` is used to calculate the size of the required exception stack, together with the equate `#SVCs`. in the control file.

Each interrupt has its own `USER` area. No user variables are initialised except for `S0` and `R0` in SVC handlers.

It is assumed that the banked stack pointers have already been set up by the hardware initialisation code.

It is implicit throughout the code that the three system registers UP, PSP, RSP are always set so that:

```
UP > PSP > RSP
```

Because of this layout, data stack underflows may corrupt the first part of the `USER` area. You have been warned. During testing, it may be as well to set the equate `SP-GUARD` to 2 or 3 in your control file to leave a few guard cells on the data stack.

7.1 Cortex-M3 NVIC Exception handlers

This section is not a treatise on the Nested Vectored Interrupt Controller. These words provide a fairly basic set of tools for using the NVIC, which is fully documented in the Cortex-M3 Technical Reference Manual (ARM DDI 0337)from *www.arm.com*.

This code requires *Kernel62.fth* and the equate `COLDCHAIN?` must be set non-zero in the control file. The NVIC address and register offsets are defined in the file *Cortex/CortexDef.fth*.

The MPE code works in terms of vector numbers, which are 16 greater than the external interrupt numbers.

`PROC ExcEntry  \ --`
This is the template code for M3+ vectored exception handlers.

`SP-SIZE #256 u< 0=  UP-SIZE #256 u< 0= or  equ LargeISR?  \ -- flag`
The interrupt is often larger than 256 bytes, which requires larger ISR entry code.

`PROC ExcEntry  \ --`
This is the template code for M0/M1 vectored exception handlers with a large ISR frame.

`PROC ExcEntry  \ --`
This is the template code for M0/M1 vectored exception handlers with a small ISR frame.

`: EXC:  \ xt vec# -- ; -- isr`
Creates an exception handler that runs the given Forth word. At run time the entry point of the ISR is returned. Use in the form:

```
' <action> <vec#> EXC: <actionISR>
```

The example below will define a handler for the SysTick interrupt/exception that runs the Forth word `SysTickHandler`.

```
' SysTickHandler #15 EXC: SysTickEntry
```

`: EnInt  \ vec# --`
Enable the requested NVIC interrupt source. NVIC interrupts have vector numbers in the range 16..255.

`: DisInt  \ vec# --`
Disable the requested NVIC interrupt source. NVIC interrupts have vector numbers in the range 16..255.

`: SetPri  \ pri vec#`
Set the priority of the given NVIC interrupt source. NVIC interrupts have vector numbers in the range 16..255. The priority numbers are in the range 0..255, where 0 is the highest priority. The number of bits actually used is implementation dependent, but unused bits are always the low-order bits. Hence, using $10, $20 and so on is fairly portable.

`: SWINT  \ vec# --`
Generate interrupt from software.

8 Exception Fault handling

The code in *Cortex/FaultCortex.fth* provides debugging facilities for when a fault occurs that triggers an exception.

8.1 Fault handler framework

`struct /AppFrame \ -- len`
Structure defining what is saved on the application's stack.

`struct /ExData \ -- len`
A structure holding data saved on entry to the exception routine

`/ExData buffer: ExData \ -- addr`
Holds additional data about the fault.

`PROC FltEntry \ --`
This is the template code for Cortex-M3+ fault handlers. R0..R3 have already been saved by the hardware

`PROC FltEntry \ --`
This is the template code for Cortex-M0/M1 fault handlers. R0..R3 have already been saved by the hardware

`: FLT: \ xt vec# -- ; -- isr`
Creates a fault handler that runs the given Forth word. At run time the entry point of the ISR is returned. Use in the form:

```
' <action> <vec#> FLT: <actionISR>
```

The example below will define a handler for the HardFault exception that runs the Forth word `HFhandler`.

```
' HFhandler 3 FLT: HFentry
```

`: .item \ addr -- addr+4`
Display a cell item at addr and increment addr.

`: .items \ addr n -- addr+4n`
Display n items at addr and increment addr.

`: AppItems \ -- addr`
The address of fault data saved on the application's stack.

`: MainItems \ -- addr`
The address of fault data saved on SP_main.

`: AppRSP \ -- addr`
The Forth return stack pointer at the fault.

`: AppPSP \ -- addr`
The Forth data stack pointer at the fault for Cortex-M3+.

`: AppPSP \ -- addr`
The Forth data stack pointer at the fault for Cortex-M3+.

`: AppUP \ -- addr`
The Forth UP at the fault.

`: AppTOS \ -- x`
The Forth TOS at the fault.

`: AppPC \ -- x`
The PC at the fault.

`: .FltFrame \ --`
Display the CPU state pointed to by the data frame.

`: name? \ addr -- flag`
Check to see if the supplied address is a valid NFA. This word is implementation dependent. A valid NFA for MPE embedded systems satisfies the following:

- All characters within string are printable ASCII within range 33..126.
- String Length is non-zero in range 1..31 and bit 7 is set, ignore bits 6, 5.

`: ip>nfa \ addr -- nfa`
Attempt to move backwards from an address within a definition to the relevant NFA.

`: check-aligned \ addr -- addr'`
Check addr for cell alignment, report and correct if misaligned.

`: ?Clip32 \ xsp xspTop -- xsp xspTop'`
Clip the stack display to 32 items.

`: .rsFault \ --`
Display the return stack of the fault, assuming the Forth RSP=R13 and RUP=R11.

`: .psFault \ --`
Display the data stack indicated by the frame, assuming the Forth RSP=AppPSP and RUP=R11.

8.2 Default Fault handlers

The interrupt structure of the Cortex-M3 is such that the simplest way to display exception information without large code and RAM overheads is to use polled operation of the comms link without interrupts. By default, the fault handler only transmits, it does not use `KEY`. If your serial driver normally uses interrupt-driven transmit routines, you must provide the word `+FaultConsole` which switches it into polled operation. An example can be found in *Cortex/Drivers/serLPC17xxqi.fth*.

By default, there is no return from a fault handler, the system is reset using `REBOOT`. From experience, attempting to restart the system by executing the boot vector is not enough. Rebooting by triggering the watchdog always works.

`: showFault \ --`
Generic fault handler.

8.3 Intepreting the crash dump

The example below comes from typing

```
55 0 !
```

on the Forth console. The device is an NXP LPC2388 and address zero is Flash to which writes have not been permitted. There are only minor differences between the ARM example here and the fault display for Cortex-M devices.

```
55 0 !
DAbort exception at PC = 0000:1C64
=== Registers ===
CPSR = 6000:001F
R0  = 0000:0037 0000:1C60 0000:0021 0000:0021
R4  = 0000:0070 0005:FD8C 4000:0298 0000:000C
R8  = 0000:1C60 0000:0000 0000:0000 4000:FEE0
R12 = 4000:FED4 4000:FDCC 0000:4FAC 0000:1C6C

RSP = 4000:FDCC   R0 = 4000:FDE0
--- Return stack high ---
4000:FDDC 0000:51E0 QUIT
4000:FDD8 4000:FED0
4000:FDD4 0000:0000
4000:FDD0 4000:FD94
4000:FDCC 0000:3244 CATCH
--- Return stack low ---

PSP = 4000:FED4   S0 = 4000:FED4
--- Data stack high ---
--- Data stack low ---
rTOS/R10  0000:0000
Restarting system ...
```

The exception occurred in the instruction at $1C64. It was a data abort exception, which means that it was a data load or store at an invalid address.

Assuming that restart is to a Forth console, you can find out where the fault occurred if you have compiled the file *Common\DebugTools.fth* or *Powernet\DebugTools.fth*.

```
$1C64 ip>nfa .name<Enter> !  ok
```

The return stack dump shows that `CATCH` was used, in turn called by `QUIT`, the text interpreter.

Further interpretation requires some knowledge of the use of the CPU registers.

8.3.1 Register usage

Cortex

For Cortex-M the following register usage is the default:

```
r15         pc      program counter
r14         link    link register; bit0=1=Thumb, usually set
r13         rsp     return stack pointer
r12         psp     data stack pointer
r11         up      user area pointer
r10         --
r9          lp      locals pointer
r8          --
r7          tos     cached top of stack
r0-r6       scratch
```

The VFX optimiser reserves R0 and R1 for internal operations. `CODE` definitions must use R10 as TOS with NOS pointed to by R12 as a full descending stack in ARM terminology. R0..R8 are free for use by `CODE` definitions and need not be preserved or restored. You should assume that any register can be affected by other words.

ARM32

On the ARM the following register usage is the default:

```
r15         pc      program counter
r14         link    link register
r13         rsp     return stack pointer
r12         psp     data stack pointer
r11         up      user area pointer
r10         tos     cached top of stack
r9          lp      locals pointer
r0-r8       scratch
```

The VFX optimiser reserves R0 and R1 for internal operations. `CODE` definitions must use R10 as TOS with NOS pointed to by R12 as a full descending stack in ARM terminology. R0..R8 are free for use by `CODE` definitions and need not be preserved or restored. You should assume that any register can be affected by other words.

8.3.2 Interpreting the registers

Using the ARM example above, we can learn more.

```
DAbort exception at PC = 0000:1C64
=== Registers ===
CPSR = 6000:001F
R0  = 0000:0037 0000:1C60 0000:0021 0000:0021
R4  = 0000:0070 0005:FD8C 4000:0298 0000:000C
R8  = 0000:1C60 0000:0000 0000:0000 4000:FEE0
R12 = 4000:FED4 4000:FDCC 0000:4FAC 0000:1C6C
```

The exception occurred in the instruction at $1C64. It was a data abort exception, which means that it was a data load or store at an invalid address.

```
TOS = R10 = 0000:0000
UP  = R11 = 4000:FEE0
PSP = R12 = 4000:FED4
RSP = R13 = 4000:FDCC
```

In general, UP > PSP > RSP. In this case that's good. TOS=0, which we would expect from the phrase:

```
55 0 !
```

We now switch back to the cross compiler, which you did leave running, didn't you? Since we now know that $1C64 is in !, we can disassemble it.

```
dis !
!
( 0000:1C60 0100BCE8 ..<h )  ldmia r12 ! { r0 }
( 0000:1C64 00008AE5 ...e )  str r0, [ r10, # $00 ]
( 0000:1C68 0004BCE8 ..<h )  ldmia r12 ! { r10 }
( 0000:1C6C 0EF0A0E1 .p a )  mov PC, LR
16 bytes, 4 instructions.
 ok
```

From this, we can see that the offending instruction is

```
( 0000:1C64 00008AE5 ...e )  str r0, [ r10, # $00 ]
```

Since R10 is 0, we now know that it was attempting a write to 0, which is not permitted.

Provided that you keep words small, the register contents at the crash point, with the stack contents and the disassembly often provide enough information to reconstruct the state of stack on entry to the word.

9 ARM Cortex multitasker

The ARM Cortex multitasker follows the model introduced with the v6.1 compilers. A few extensions are also provided.

9.1 Configuration - normally performed earlier

```
0 equ test-multi?       \ true to compile test code
```

If previously undefined, TEST-MULTI? is set to zero and test code is not compiled.

9.2 TCB data structure layout

```
cell       LINK    link to next task
cell       SSP     Saved Stack Pointer
cell       STAT    Bit 0     1 = running, 0 = halted
                   Bit 1     1 = message pending
                   Bit 2     1 = event triggered
                   Bit 3     1 = event handler run
                   Bit 4..7  Reserved
                others  1 = set to run task, available to user
cell       TASK    Task that sent message here
cell       MESG    Message address
cell       EVNTw   CFA of word run as event handler
```

This structure is allocated at the start of the USER area. Consequently the TCB of the current task is given by UP.

```
struct /TCB     \ -- size
```

The structure used by the code that matches the description above.

9.3 Task handling primitives

```
init-u0 constant main   \ -- addr ; tcb of main task
```

Returns the base address of the main task's USER area.

```
0 value multi?  \ -- flag
```

Returns true if the tasker is enabled.

```
: single        \ --
```

Disable scheduler.

```
: multi         \ --
```

Enable scheduler.

```
CODE pause      \ -- ; the scheduler itself
```

The software scheduler itself.

```
: status        \ -- task-status
```

Returns the current task's status cell, but with the run bit masked out.

```
: restart       \ task -- ; mark task TCB as running
```

Sets the RUN bit in the task's status cell.

```
: halt          \ task -- ; reset running bit in TCB
```

Clears the RUN bit in the task's status cell.

```
: stop          \ -- ; halt oneself
```
HALTs the current task, and executes PAUSE.

9.4 Event handling

Event handling is only compiled if the equate EVENT-HANDLER? is set non-zero in the control file.

```
: set-event     \ task --
```
Set the event trigger in task TCB.

```
: event?        \ task -- flag
```
Returns true if true if task has received an event trigger which has not been cleared yet.

```
: clr-event-run \ --
```
Reset the current task's EVENT_RUN flag.

```
: to-event      \ xt task -- ; define action of a task
```
Sets XT as the event handler for the task.

9.5 Message handling

Message handling is only compiled if the equate MESSAGE-HANDLER? is set non-zero in the control file.

```
: msg?          \ task -- flag
```
Returns true if task has received a message.

```
: send-message  \ addr task --
```
Send a message to a task.

```
: get-message   \ -- addr task
```
Wait for any message and return the message and the task it came from.

```
: wait-event/msg        \ --
```
Wait for a message or an event trigger.

9.6 Task structure management

```
code init-task  \ xt task -- ; Initialise a task stack
```
Initialise a task's stack before running it and set it to execute the word whose XT is given.

```
: add-task      \ task -- ; insert into list
```
Add the task to the list of tasks after the current task.

```
: sub-task      \ task -- ; remove task from chain
```
Remove the task from the task list.

```
: initiate      \ xt task -- ; start task from scratch
```
Start the given task executing the word whose XT is given, e.g.

```
  ['] <name> <task> INITIATE
```

```
: sleeper       \ xt task --
```
Start task from scratch, but leave it HALTed. Use in the form:

```
  ['] <action> <taskname> SLEEPER
```

to put a task on the active task list, but as if HALTed. SLEEPER allows you to make a task ready

for waking up later, perhaps by another task. This avoids having to put `STOP` as the first word in a task. Note that `SLEEPER` does not call `PAUSE`. See also `INITIATE`.

```
: terminate     \ task --
```

Stop a task, and remove it from the list.

```
: init-multi    \ -- ; initialisation with multi-tasking
```

Initialise the multitasker and start it. If tasking is selected by setting the equate `TASKING?` in the control file, *KERNEL62.FTH* will automatically run this word. Make sure that your initialisation code includes `INIT-MULTI` or your code will crash.

```
: his           \ task uservar -- addr
```

Given a task id and a `USER` variable, returns the address of that variable in the given task. This word is used to set up `USER` variables in other tasks.

9.7 Semaphores

The semaphore code is only compiled if the equate `SEMAPHORES?` is set non-zero in the control file.

A `SEMAPHORE` is an extended variable used for signalling between tasks, and for resource allocation. It contains two cells, a **counter** and an **arbiter**. The counter field is used as a count of the number of times the resource may be used, and the arbiter field contains the TCB of the task that currently owns it. This field can be used for priority arbitration and deadlock detection/arbitration. The count field allows the semaphore to be used as a**counted** semaphore or as an exclusive access semaphore.

For example a character buffer may be used where the semaphore counter contains the number of available characters.

An exclusive access semaphore is used to share resources. The semaphore is initialised to one, usually by `SIGNAL`. The first task to `REQUEST` it gains access, and all other tasks must wait until the accessing task `SIGNAL`s that it has finished with the resource.

```
: semaphore     \ -- ; -- addr [child]
```

Creates a semaphore which returns its address at runtime. The count field is initialised to zero for use as counted semaphore. Use in the form:

```
Semaphore <name>
```

If you want this to be an exclusive access semaphore, follow this with:

```
  1 <name> !
```

```
: signal        \ addr --
```

`SIGNAL` increments the counter field of a semaphore, indicating either that another item has been allocated to the resource, or that it is available for use again, 0 indicating in use by a task.

```
: request       \ sem -- ; get access to resource, wait if count = 0
```

`REQUEST` waits until the counter field of a semaphore is non-zero, and then decrements the counter field by one.

9.8 TASK and START:

`TASK <name>` builds a named task user area. The action of a task is assigned and the task started by the word `INITIATE`

```
['] <action> <task> INITIATE
```

`START:` is used inside a colon definition. The code before `START:` is the task's initialisation, performed by the current task. The code after `START:` up to the closing `;` is the action of the task. For example:

```
TASK FOO
: RUN-FOO
  ...
  FOO START:
  ...
  begin ... pause again
;
```

All tasks must run in an endless loop, except for initialisation code. When `RUN-FOO` is executed, the code after `START:` is set up as the action of task `FOO` and started. `RUN-FOO` then exits.

If you want to perform additional actions after starting the task, you should use `INITIATE` to start the task.

`variable task-chain  \ -- addr`
Anchors list of all tasks created by `TASK` and friends.

`: task  \ -- ; -- task ; TASK <name> builds a task`
Note that the cross-interpreter's version of `TASK` has been modified from v6.2 onwards to leave the current section as `CDATA`.

`: task  \ -- ; -- task ; TASK <name> builds a task`
Creates a new task and data area, returning the address of the user area at run time. The task is also linked into the task chain anchored by `TASK-CHAIN`.

`: start:  \ task -- ; exits from caller`
Used inside a colon definition. The code following `START:` up to the ending semi-colon forms the action of the task. The word containing `START:` finishes at `START:`.

9.9 Debugging tools

`: .task  \ task --`
Display task's name if it has one, otherwise display its address.

`: .tasks  \ task -- ; display all task names`
Display all the tasks anchored by `TASK-CHAIN`.

`: .running  \ --`
Display all the running tasks.

10 Development tools

The file *COMMON\DEVTOOLS.FTH* supplies words that are mostly used during development and debugging.

`1 equ HexTools? \ -- flag`
If non-zero, `.DWORD` and friends will be compiled.

`1 equ DUMP?     \ -- flag`
If non-zero, `DUMP` and friends will be compiled.

10.1 Hexadecimal display tools

`1 equ simple?   \ -- n`
Set this flag non-zero to generate `.xWORD` to avoid divisions, which are slow on some CPUs.

`: .nibble       \ n --`
Convert a nibble to a hex ASCII digit and display it.

`: .BYTE         \ b --`
Display b as two hex digits.

`: .WORD         \ w --`
Display w as four hex digits.

`: .LWORD        \ x --`
Display x as eight hex digits. The separator `":"` makes the output easier to read. Future releases of MPE Forths will treat the `":"` character as having no effect on number input parsing. This character is chosen because it does not conflict with the current use of the `"."` and `","` characters for numbers. This word is only compiled for 32 bit targets.

`: .DWORD        \ x --`
A synonym for .LWORD.

10.2 DUMP and friends

`: .ASCII        \ char --`
The top bit of char is zeroed. If char is in the range 32..126 it is displayed, otherwise a `"."` is displayed.

`: DUMP          \ addr len --`
Display (dump) len bytes of memory starting at addr.

`: LDUMP         \ addr len -- ; dump 32 bit long words`
Display (dump) len bytes of memory starting at addr as 32 bit words.

`: PDUMP         \ addr len -- ; dump 32 bit long words`
Display (dump) len bytes of memory starting at addr as 32 bit words with **no** ASCII dump. This word may be necessary when accessing peripherals that must only be addressed on 32 bit boundaries.

`: WDUMP         \ addr len -- ; dump 16 bit half words`
Display (dump) len bytes of memory starting at addr as 16 bit half-words.

`: .S            \ --`
Display the contents of the data stack without affecting it.

`: ?             \ a-addr --`
Display contents of a memory location as a cell.

11 Debugging tools

MicroProcessor Engineering
133 Hill Lane
Southampton SO15 5AF
England

tel: +44 (0)23 8063 1441
fax: +44 (0)23 8033 9691
net: mpe@mpeforth.com
tech-support@mpeforth.com
web: www.mpeforth.com

The file *Common\DebugTools.fth* provides debugging tools for MPE embedded systems created by Forth 6 Cross Compilers. The emphasis is on 32 bit systems and interactive testing. The tools can easily be ported to other systems. Copyright is retained by MPE. The code may be freely used on non-MPE systems for non-commercial use. The copyright notice must be preserved.

Porting the code to other systems is up to you. This code may require some carnal knowledge of how your system works. Most Forths contain the required words, but they may not have the same names that MPE use.

11.1 Implementation dependencies

In MPE embedded systems, the `USER` variables `IPVEC` and `OPVEC` contain the address of the device structure used for input and output by `KEY`, `EMIT` and friends. In VFX Forth for Windows/Linux, the variables are `IP-HANDLE` and `OP-HANDLE`.

```
: consoleIO     \ --
```
Select debug console for output. By default this is the `CONSOLE` device.
```
  console  dup opvec !  ipvec !
  Echoing on  Xon/Xoff off
;


: name?         \ addr -- flag                                          MPE.0000
```
Check to see if the supplied address is a valid NFA, returning true if the address appears to be a valid NFA. This word is implementation dependent. For MPE cross compilers, a valid NFA for MPE embedded systems satisfies the following:

- All characters within string are printable ASCII within range 33..126
- String Length is non-zero in range 1..31 and bit 7 is set, ignore bits 6, 5

```
  count                                 \ c-addr u --
  dup  $9F and  $81 $9F within? 0=      \ NFA first byte = 1SIxxxxx, count = xxxxx
                                        \ mask           = 10011111
  if  2drop  0  exit  then
  $01F and bounds ?do
    i c@ #33 #126 within? 0=            \ check all ascii chars
    if  unloop  FALSE exit  then
```

```
  loop
  TRUE
;

: ip>nfa        \ addr -- nfa
```
Attempt to move backwards from an address within a definition to the relevant NFA.
```
  2-                            \ NFA must be at least 'n' bytes backwards
  begin
    dup name? 0=
   while
    1-
  repeat
;

: >name         \ xt -- nfa
```
Move from a word's xt to its name field. If `>NAME` does not exist `IP>NFA` will be used.
```
  ip>nfa
;

: .name         \ nfa --
```
Given a word's NFA display its name.
```
  count $1F and type
;

: .DWORD        \ dw --
```
Display the 32 bit long word 'dw' as an 8 digit hex number.
```
  base @ hex  swap
  0 <# # # # # ascii : hold # # # # #> type
  base !
;
```

11.2 Miscellaneous

MPE systems use `TICKS ( -- ms)` to return a running time count in milliseconds. Windows systems can use the **GetTickCount** API call.

```
: times         \ n -- ; n TIMES <word>
```
Execute `<word>` n times, and display the execution time. The ticker interrupt must be running.
```
  ticks  '  rot 0                       \ -- ticks xt n 0
  ?do  dup execute  loop
  drop
  ticks swap - . ." ms"
;

: .ColdChain    \ --
```
Display all words added to the cold chain. Note that the first word added is displayed first. In VFX Forth this word is called ShowColdChain.

```
  cr  ColdChainFirst
  begin
    dup
   while
    dup cell + @ >name .name            \ execute XT
    @                                   \ get next entry
  repeat
  drop
;

: .decimal      \ n --
```

Display a value as a decimal number.

```
  base @ >r  decimal .  r> base !
;

: .hex          \ n --
```

Display a value as a hexadecimal number.

```
  base @ >r  hex u.  r> base !
;

: [con          \ -- ; R: -- consys
```

Saves `BASE` and the current i/o vectors on the return stack, and then switches to the console and `DECIMAL`.

```
   r>
   base @ >r  opvec @ >r  ipvec @ >r
   ConsoleIO decimal
   >r
;

: con]          \ -- ; R: consys --
```

Restores `BASE` and the current i/o vectors from the return stack.

```
  r>
  r> ipvec !  r> opvec !  r> base !
  >r
;

: CheckFailed   \ ip caddr len --
```

Given the address where the fault occurred and a string, output the string and some diagnostic information.

```
  [con
    cr type  ."  failed at "
    dup .dword ."  in " ip>nfa .name
  con]
;
```

11.3 Stack checking

Especially in multi-tasked systems, stack errors can be fatal. Detecting them as early as possible reduces debugging time. These words rely on Forth return stack cells containing return addresses. This is true on the vast majority of Forth systems except for some 8051 and real-mode 80x86 systems. If you find others, please let us know.

```
: ?StackDepth   \ +n --
```

If the stack depth before +n is not n, issue a console warning message and clear the stack. Note that this word is implementation dependent.

```
  dup 2+ depth =
  if  drop  exit  endif                  \ no failure
  [con
  cr ." *** Stack fault: depth = " depth 1- 0 .r ." (d) "
[ tasking? ] [if]
  ." in task " self .task                \ indicate current task
[then]
  >r  s0 @ sp!  r> 0 ?do  0  loop        \ set required depth
  cr ."   Stack updated."
  con]
;


: ?StackEmpty   \ --
```

If the stack depth is non-zero, issue a console warning message and clear the stack.

```
  0 ?StackDepth  ;


: TaskChecks    \ --
```

Use in task to check for creeping stacks and so on. This word can be extended to provide additional internal consistency checks.

```
  ?StackEmpty  ;


: SF{           \ n -- ; R: -- depth
```

n SF{ .... }SF will check for stack faults. n describes the stack change between SF{ and }SF. If the stack change is different, an error message is generated. This word will work on most systems in which the return address is held on the return stack.

```
  r>  swap depth 2- + >r  >r
;


: }SF           \ -- ; R: depth -- ; perform stack check
```

The end of an SF{ ... }SF structure. This word is not strictly portable as it assumes that the Forth return stack holds a valid return address. In the vast majority of cases the assumption is true, but beware of some 8051 implementations. See SF{

```
  r>
  r> depth 2- <> if
    dup s" Stack check" CheckFailed
  endif
  >r
;
```

11.4 Assertions

Assertions are a useful way to check that the system is behaving correctly. When the phrase:

```
[ASSERT <test> ASSERT]
```

is compiled into a piece of code, the test is performed and generates an error report if the result is false. If you do not want the performance overhead of the test, set the value `ASSERTS?` to zero. To remove even the small overhead of of testing `ASSERTS?`, comment out the line.

```
-1 value assert?        \ -- n
```
Returns non-zero if asserts will be tested.

```
: (assert)      \ flag --
```
If flag is zero, report an ASSERT error.

```
  if  exit  endif                       \ faster on some CPUs
  r@ s" ASSERT" CheckFailed
;
```

```
: [assert       \ --
```
Compile the code to start an assert.

```
  ?comp                                 \ must be compiling
  postpone assert?  postpone if
; immediate
```

```
: assert]       \ --
```
Compile the code to end an assert.

```
  ?comp                                 \ must be compiling
  postpone (assert)  postpone then
; immediate
```

Here is a simple assert that will fail if `BASE` is not `DECIMAL`.

```
: foo     \ --
  [assert  base @ #10 =  assert]
;
```

11.5 Return stack trace

```
: rpick         \ n -- x
```
Get a copy of the Nth return stack item. This works on most systems with a grow-down return stack in main memory.

```
: rdepth        \ -- n
```
Return the number of items on the return stack. This works for most systems with a grow-down stack in main memory.

11.6 Cold Chain

```
: ShowColdChain \ --
```
Display the cold chain

12 ANS Environment System

The file COMMON\ENVIRON.FTH provides the ANS ENVIRONMENT? word, and some basic environment data.

```
Vocabulary Environment                    \ used for enviroment? queries
```

The vocabulary within which environment data is kept.

```
: ENVIRONMENT?  \ c-addr u --  false | i*x true
```

The string is treated as a word name. If it is found in the ENVIRONMENT vocabulary, the word is executed the results of executing it plus true are returned, otherwise just false is returned.

13 Software Floating Point

13.1 Introduction

Although most embedded applications only require integer arithmetic, some do require floating-point. Therefore software floating-point is supplied with the cross-compiler and the target Forth. The target floating point wordset is not fully ANS compliant, but satisfies the needs of embedded systems without undue complexity. The Forth data stack and the floating point stack are the same. The floating point data storage format is not IEEE format, but is optimised for performance on small controllers. If you need a separate floating point stack or IEEE double format storage, please contact MPE. Any variations in the implementation will be documented in the target specific section of the manual.

The cross-compiler has a more limited floating-point support than the target. Some words are avaliable during compilation of colon definitions, but not while interpreting.

13.2 Source code

The source code is in two sets of files, one for 32 bit Forth targets, the other for 16 bit targets. The files are:

```
Common\sfp32hi    32 bit primitives
Common\sfp32com   32 bit high level code
Common\sfp16hi    16 bit primitives
Common\sfp16com   16 bit high level code
```

These files use no assembler definitions. Some targets have code versions of the primitives, and these will be found in the CPU specific code directory. A significant increase in performance can be obtained by using the code files.

13.3 Entering floating-point numbers

Floating point number entry is enabled by `REALS` and disabled by `INTEGERS`.

Floating-point numbers of the form 0.1234e1 are required (see `FNUMBER?`) during interpretation and compilation of source code. Floating-point numbers are compiled as literal numbers when in a colon definition (compiling) and placed on the stack when outside a definition (interpreting).

The more flexible word `>FLOAT` accepts numbers in two forms, 1.234 and 0.1234e1. Both words are documented later in this chapter. See also the section on *Gotchas* later in this chapter.

Note also that MPE Forths use ',' as the double number indicator - it makes life much easier for Europeans.

13.4 The form of floating-point numbers

A floating-point number is placed on the Forth data stack. In the Forth literature, this is referred to as a combined floating point and data stack. For 32 bit targets, a floating point

number consists of two 32-bit numbers, one for the mantissa and one for the exponent. For 16 bit targets, it consists of a 32-bit double mantissa and a single 16-bit exponent. The mantissa is normalised. The exponent is on the top of the stack. Note that for 16 bit targets, number conversion is affected by the cross-compiler directives `HOST-MATH` and `TARGET-MATH`. `HOST-MATH` leaves double numbers and floats in 32-bit form, whereas `TARGET-MATH` leaves them in 16-bit form.

13.5 Creating and using variables

To create a variable, use `FVARIABLE`. `FVARIABLE` works in the same way as `VARIABLE`. For example, to create a floating-point variable called `VAR1` you code:

```
FVARIABLE VAR1
```

When `VAR1` is used, it returns the address of the floating-point number.

Two words are used to access floating-point variables, `F@` and `F!`. These are analogous to `@` and `!`.

13.6 Creating constants

To create a floating-point constant, use `FCONSTANT`, which is analogous to `CONSTANT`. For example, to generate a floating-point constant called `CON1` with a value of 1.234, you enter:

```
1.234e0 FCONSTANT CON1
```

When `CON1` is executed, it returns 1.234 on the Forth stack.

13.7 Using the supplied words

The supplied words split into several groups:

- sines, cosines and tangents
- arc sines, cosines and tangents
- arithmetic functions
- logarithms
- powers
- displaying floating-point numbers
- inputting floating-point numbers

The following functions only exist as target words so you cannot use them in calculations in your source code when outside a colon definition.

13.7.1 Calculating sines, cosines and tangents

To calculate sine, cosine and tangent, use `FSIN`, `FCOS` and `FTAN` respectively. Angles are expressed in radians.

13.7.2 Calculating arc sines, cosines and tangents

To calculate arc sine, cosine and tangent, use `FASIN`, `FACOS`

and `FATAN` respectively. They return an angle in radians.

13.7.3 Calculating logarithms

Two words are supplied to calculate logarithms, `FLOG` and `FLN`. `FLOG` calculates a logarithm to base 10 (decimal). `FLN` calculates a logarithm to base e. Both take a floating-point number in the range from 0 to Einf.

13.7.4 Calculating powers

Three power functions are supplied:

```
FE^X F10^X X^Y
```

13.8 Degrees or radians

The angular measurement used in the trigonometric functions are in radians. To convert between degrees and radians use `RAD>DEG` or `DEG>RAD`. `RAD>DEG` converts an angle from radians to degrees. `DEG>RAD` converts an angle from degrees to radians.

13.9 Displaying floating-point numbers

Two words are available for displaying floating-point numbers, `F.` and `E.`. The word `F.` takes a floating-point number from the stack and displays it in the form xxxx.xxxxx or x.xxxxxEyy depending on the size of the number. The word `E.` displays the number in the latter form.

13.10 Number formats, ANS and Forth200x

The ANS Forth standard specifies that floating point numbers must be entered in the form 1.234e5 and must contain a point '.' and 'e' or 'E', and that double integers are terminated by a point '.'.

This situation prevents the use of the standard conversion words in international applications because of the interchangable use of the '.' and ',' characters in numbers. Because of this, the cross-compiler's host VFX Forth uses two four-byte arrays, `FP-CHAR` and `DP-CHAR`, to hold the characters used as the floating point and double integer indicator characters. By default, `FP-CHAR` is initialised to '.' and `DP-CHAR` is initialised to to ',' and '.'. For strict ANS compliance, you should set them as follows **before** `CROSS-COMPILE` is run.

```
\ ANS standard setting
  char . dp-char !
  char . fp-char !
: ans-floats    \ -- ; for strict ANS compliance
  [char] . dp-char !
  [char] . fp-char !
;
\ MPE defaults
  char , dp-char !
  char . dp-char 1+ c!
  char . fp-char !
: mpe-floats    \ -- ; for existing and most legacy code
  [char] , dp-char !
  [char] . dp-char 1+ c!
  [char] . fp-char !
;
\ Legacy defaults, including ProForth
  char , dp-char !
  char . fp-char !
: legacy-floats \ -- ; for legacy code
  [char] , dp-char !
  [char] . fp-char !
;
```

You can of course set these arrays to hold any values which suit your application's language and locale. Note that integer conversion is always attempted before floating point conversion. This means that if the `FP-CHAR` and `DP-CHAR` arrays contain the same character, floating point numbers must contain 'e' or 'E'. If the arrays are all different, a number containing the `FP-CHAR` will be successfully converted as a floating point number, even if it does not contain 'e' or 'E'.

13.11 Glossary

13.11.1 Separators

Before July 2010, the floating point separator, '.', was fixed. To ease internationalisation, it is now variable.

`variable fp-char        \ -- addr`

Holds up to four character(s) to be treated as floating point indicators. Set to '.' for ANS compatibility. Note that this should be accessed as a one to four byte array. The first character is used as the point character for output.

`0 equ SepArray? \ -- flag`

If the equate is non-zero, `fp-char` is treated as a four byte array, otherwise as a one byte array. This is a flag for future expansion.

`: isSep?        \ char addr -- flag`

Return true if *char* is one of the four bytes at *addr*. If less than than four bytes are needed, a zero byte acts as a terminator. Used when `SepArray?` is true.

`: isSep?  c@ =  ;`

A compiler macro used when `SepArray?` is false.

13.11.2 Basic stack and memory operators

`: F!  \ r addr --`
Stores r at addr

`: F@  \ addr -- r`
Fetches r from addr.

`: F,  \ r --`
Lays a real number into the dictionary, reserving 8 bytes.

`: FDUP  \ r -- r r`
Floating point equivalent of `DUP`.

`: FOVER  \ r1 r2 -- r1 r2 r1`
Floating point equivalent of `OVER`.

`: FROT  \ r1 r2 r3 -- r2 r3 r1`
Floating point equivalent of `ROT`.

`: FPICK  \ fu..f0 u -- fu..f0 fu`
Floating point equivalent of `PICK`.

`: FROLL  \ f1 f2 f3 --  f2 f3 f1`
Floating point equivalent of `ROLL`.

`: FSWAP  \ r1 r2 -- r2 r1`
Floating point equivalent of `SWAP`.

`: FDROP  \ r --`
Floating point equivalent of `DROP`.

`: FNIP  \ r1 r2 -- r2`
Floating point equivalent of `NIP`.

13.11.3 Floating point defining words

`: FVARIABLE  \ "<spaces>name" -- ; Run: -- f-addr`
Use in the form: `FVARIABLE <name>` to create a variable that will hold a floating point number.

`: FCONSTANT  \ r "<spaces>name" -- ; Run: -- r`
Use in the form: `<float> FCONSTANT <name>` to create a constant that returns a floating point number.

`: FARRAY  \ "<spaces>name" fn-1..f0 n -- ; Run: i -- ri`
Create an initialised array of floating point numbers. Use in the form:

```
fn-1 .. f1 f0 n FARRAY <name>
```

to create an array of n floating point numbers. When the array `name` is executed, the index i is used to return the address of the i'th 0 zero-based element in the array. For example:

```
4e0 3e0 2e0 1e0 0e0 5 FARRAY TEST
```

will set up an array of five elements. Note that the rightmost float (0e0) is element 0. Then `i TEST` will return the *\{i}th element. If you create this array in `IDATA`, restore `CDATA` afterwards.

`: FBUFF  \ u "name" -- ; i -- addr`
Creates a buffer of u floats in the current memory section. The child action is to return the address of the ith element (zero-based).

```
10 fbuff foo
```

Creates an buffer for ten float elements in.

```
3 foo
```

Returns the address of element 3 in the buffer.

The default section is `CDATA`, and we recommend that you leave it that way! To create a ten element array in `UDATA` space, you can use:

```
udata
10 fbuff MyFloats
cdata
```

13.11.4 Type conversions

`: NORM \ n exp -- f`
Normalise a single integer and a single exponent to produce a floating point number. INTERNAL.

`: DNORM \ d exp -- fn ; normalise a 64 bit double`
Normalise a double integer and a single exponent to produce a floating point number. INTERNAL.

`: FSIGN \ fn -- |fn| flag ; true if negative`
Return the absolute value of fn and a flag which is true if fn is negative.

`: S>F \ n -- fn`
Converts a single integer to a float.

`: F>S \ fn -- n`
Converts a float to a single integer. Note that `F>S` truncates the number towards zero according to the ANS specification. If `|fn|` is greater than maxint, `+/-maxint` is returned.

`: D>F \ d -- fn`
Converts a double integer to a float.

`: F>D \ fn -- d`
Converts a float to a double integer. Note that `F>D` truncates the number towards zero according to the ANS specification. If `|fn|` is greater than dmaxint, `+/-dmaxint` is returned.

`: FINT \ f1 -- f2`
Chop the number towards zero to produce a floating point representation of an integer.

13.11.5 Arithmetic

`: FNEGATE \ r1 -- r2`
Floating point negate.

`: ?FNEGATE \ fn n -- fn|-fn`
If n is negative, negate fn.

`: FABS \ fn -- |fn|`
Floating point absolute.

`: F* \ r1 r2 -- r3`
Floating point multiply.

```
: F/            \ r1 r2 -- r3
```

Floating point divide.

```
: F+            \ r1 r2 -- r3
```

Floating point addition.

```
: F-            \ r1 r2 -- r3
```

Floating point subtraction.

```
: FSEPARATE     \ f1 f2 -- f3 f4
```

Leave the signed integer quotient f4 and remainder f3 when f1 is divided by f2. The remainder has the same sign as the dividend.

```
: FFRAC         \ f1 f2 -- f3
```

Leave the fractional remainder from the division f1/f2. The remainder takes the sign of the dividend.

13.11.6 Relational operators

```
: F0<           \ f1 -- flag
```

Floating point 0<.

```
: F0>           \ f1 -- flag
```

Floating point 0>.

```
: F0=           \ f1 -- flag
```

Floating point 0=.

```
: F0<>          \ f1 -- flag
```

Floating point 0<>.

```
: F=            \ f1  f2 -- flag
```

Floating point =.

```
: F<            \ r1  r2 -- flag
```

Floating point <.

```
: F>            \ f1  f2 -- flag
```

Floating point >.

```
: FMAX          \ r1 r2 -- r1|r2
```

Floating point MAX.

```
: FMIN          \ r1 r2 -- r1|r2
```

Floating point MIN.

13.11.7 Rounding

```
f# 1.0 fconstant %ONE
```

Floating point 1.0.

```
: FLOOR         \ r1 -- r2
```

Floored round towards -infinity.

```
: FROUND        \ r1 --  r2
```

Round the number to nearest or even.

13.11.8 Miscellaneous

`: FALIGNED \ addr -- f-addr`
Aligns the address to accept an 8-byte float.

`: FALIGN \ --`
Aligns the dictionary to accept an 8-byte float.

`: FDEPTH \ -- +n`
Returns the number of floats on the stack.

`: FLOAT+ \ f-addr1 -- f-addr2`
Increments addr by 8, the size of a float.

`: FLOATS \ n1 -- n2`
Returns n2, the size of n1 floats.

13.11.9 Floating point output

`1 s>f 10 s>f f/ fconstant %.1`
Floating point 0.1.

`1 s>f fconstant %1`
Floating point 1.0.

`10 s>f fconstant %10`
Floating point 10.0.

`1250000000 34 fconstant %10^10`
Floating point 10^10.

`1844674407 -33 fconstant %10^-10`
Floating point 10^-10.

`F# 1.0E256 FCONSTANT %10^256`
Floating point 10^256.

`F# 1.0E-1 FCONSTANT %10E-1`
Floating point 10^-1.

`F# 1.0E-10 FCONSTANT %10E-10`
Floating point 10^-10.

`F# 1.0E-256 FCONSTANT %10^-256`
Floating point 10^-256.

`16 FARRAY POWERS-OF-10E1`
An array of 16 powers of ten starting at 10^0 in steps of 1.

`17 FARRAY POWERS-OF-10E16`
An array of 17 powers of ten starting at 10^0 in steps of 16.

`16 FARRAY POWERS-OF-10E-1`
An array of 16 powers of ten starting at 10^0 in steps of -1.

`17 FARRAY POWERS-OF-10E-16`
An array of 17 powers of ten starting at 10^0 in steps of -16.

`: RAISE_POWER \ mant exp -- mant' exp'`
Raise the power in preparation for number formatting.

`: SINK_FRACTION \ mant exp -- mant' exp'`

Reduce the power in preparation for number formatting.

`variable places  8 places !     \ -- addr`
Number of digits output after the decimal point.

`: ROUND          \ f1 -- f2`
Rounds least significant eight bits to 0 if higher 2 bits are all 0s or all 1s.

`: ?10PWR         \ exp[2] -- exp[2] exp[10]`
Generate the power of ten corresponding to the power of two. INTERNAL.

`: SIGFIGS        \ fn n -- d dec_exponent`
From fn, generate a double number corresponding to n significant digits and a decimal exponent. INTERNAL.

`: op-prepare     \ fn -- d exp sign`
From fn, generate a double number corresponding to n significant digits, a decimal exponent and a sign indicator (nz=negative). INTERNAL.

`: .EXP           \ exp --`
Display the exponent. INTERNAL.

`: N#             \ d n -- d'`
Convert n digits. INTERNAL.

`: .FPsign        \ flag --`
If flag is non-zero, generate a '-' otherwise a space.

`: .FPsep         \ --`
Issue the FP separator, usually '.'.

`: E.             \ n exp --`
Print the f.p. number on the stack in exponential form, x.xxxxxEyy.

`: REPRESENT      \ r c-addr u -- n flag1 flag2`
Assume that the floating number is of the form +/-0.xxxxEyy. Place the significand xxxxx at c-addr with a maximum of u digits. Return n the signed integer version of yy. Return flag1 true if f is negative, and return flag2 true if the results are valid. In this implementation all errors are handled by exceptions, and so flag2 is always true.

`: F.             \ f --`
Print the f.p. number in free format, xxxx.yyyy, if possible. Otherwise display using the x.xxxxEyy format.

13.11.10 Floating point input

Note that number conversion takes place in `PAD`.

`: FLITERAL       \ Comp: r -- ; Run: -- r`
Compiles a float as a literal into the current definition. At execution time, a float is returned. For example, `[ %PI F2* ] FLITERAL` will compile 2PI as a floating point literal. Note that `FLITERAL` is immediate.

`: CONVERT-EXP    \ c-addr --`
If the character at c-addr is 'D' convert it to 'E'. INTERNAL.

`: CONVERT-FPCHAR         \ c-addr --`
Convert the f.p. char '.' to the double char ',' for conversion. INTERNAL.

`: ALL-BLANKS?    \ c-addr len -- flag`

Return true if string is all blanks (spaces). INTERNAL.

```
: FCHECK        \ -- am lm ae le e-flag .-flag
```

Check the input string at PAD, returning the separated mantissa and exponent flags. The e-flag is returned true if the string contained an exponent indicator 'E' and the .-flag is returned true if a '.' was found. INTERNAL.

```
: MNUM          \ c-addr u -- d 2 | 0
```

Convert the mantissa string to a double number and 2. If conversion fails, just return 0. INTERNAL.

```
: ENUM          \ c-addr u -- n 1 | 0 ; str as above
```

Convert the exponent string to a single number and 1. If conversion fails, just return 0. INTERNAL.

```
: *10^X         \ float dec_exponent -- float'
```

Generate float' = float *10^dec_exp. INTERNAL.

```
: FIXEXP     \ dmant exp -- mant' exp'
```

Convert a double integer mantissa and a single integer exponent into a floating point number. INTERNAL.

```
: FNUMBER?      \ addr --  0/.../mant exp 2
```

Behaves like the integer version of NUMBER? except that if the number is in F.P. format and BASE is decimal, a floating point conversion is attempted. If conversion is successful, the floating point number is left on the float stack and the result code is 2. This word only accepts words with an 'E' as a floating point indicator, e.g, 1.2345e0. If BASE is not decimal all numbers are treated as integers. The integer prefixes '#','$','0x' etc. are recognised and cause integer conversion to be used.

```
: >FLOAT        \ c-addr u -- r true | false
```

Try to convert the string at c-addr/u to a floating point number. If conversion is successful, flag is returned true, and a floating number is returned on the float stack, otherwise just flag=0 is returned. This word accepts several forms, e.g. 1.2345e0, 1.2345, 12345 and converts them to a float. Note that double numbers (containing a ',') cannot be converted. Number conversion is decimal only, regardless of the current BASE.

```
: (F#)          \ addr -- fn 2 | 0
```

The primitive for F# and F#IN below.

```
: F#IN          \ -- fn 2 | 0
```

Attempts to convert a token from the input stream to a floating-point number. Numbers in integer format will be converted to floating-point. An indicator (0 or 2/3) is returned in the same way as an indicator is returned by FNUMBER?.

```
: F#            \ -- [f] ; or compiles it [ state smart ]
```

If interpreting, takes text from the input stream and, if possible converts it to a f.p. number on the stack. Numbers in integer format will be converted to floating-point. If compiling, the converted number is compiled.

```
: REALS         \ -- ; allow f.p input
```

Switch NUMBER? to permit floating point input using FNUMBER?. This action can be reversed by INTEGERS. Both REALS and INTEGERS are in the FORTH vocabulary.

```
: INTEGERS      \ -- ; no f.p input
```

Switch NUMBER? to restore integer only input.

13.11.11 Trigonmetric functions

N.B. All angles are in radians.

```
: DEG>RAD       \ n1 -- n2
```
Convert degrees to radians.

```
: RAD>DEG       \ n1 -- n2
```
convert radians to degrees.

```
: FSQR          \ f1 -- f2 ; FSQR by Heron's formula
```
F2=sqrt(f1) by Heron's formula.

```
: FSIN          \ f1 -- f2
```
f2=sin(f1).

```
: FCOS          \ f1 -- f2
```
f2=cos(f1).

```
: FTAN          \ f1 -- f2
```
f2=tan(f1).

```
: FASIN         \ f1 -- f2
```
f2=arcsin(f1).

```
: FACOS         \ f1 -- f2
```
f2=arccos(f1).

```
: FATAN         \ f1 -- f2
```
f2=arctan(f1).

13.11.12 Power and logarithmic functions

```
: FLN           \ f1 -- f2
```
Take the logarithm of f1 to base e and return the result.

```
: FLOG          \ f1 -- f2
```
Take the logarithm of f1 to base 10 and return the result.

```
: FE^X          \ f1 -- f2
```
f2=e^f1.

```
: F10^X         \ f1 -- f2
```
f2=10^f1

```
: FX^N          \ x-real n-integer -- fx^n
```
fx^n=x^n where x is a float and n is an integer.

```
: FX^Y          \ x-real y-real -- fn
```
fn=X^Y where Y and Y are both floats.

13.11.13 IEEE format conversion

```
: FP>IEEE       \ fp -- ieee32
```
Convert native FP value to IEEE 32 bit format.

```
: IEEE>FP       \ ieee32 -- fp
```
Convert IEEE 32 bit float to native format.

13.12 Gotchas

The ANS and Forth200x specifications define the format of floating point numbers during text interpretation as:

```
Convertible string := <significand><exponent>

<significand> := [<sign>]<digits>[.<digits0>]
<exponent>    := E[<sign>]<digits0>
<sign>        := { + | - }
<digits>      := <digit><digits0>
<digits0>     := <digit>*
<digit>       := { 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 }
```

This format is handled by the word `FNUMBER?`. The word `>FLOAT` accepts a more relaxed format.

```
Convertible string := <significand>[<exponent>]

<significand> := [<sign>]{<digits>[.<digits0>] | .<digits> }
<exponent>    := <marker><digits0>
<marker>      := {<e-form> | <sign-form>}
<e-form>      := <e-char>[<sign-form>]
<sign-form>   := { + | - }
<e-char>      := { D | d | E | e }
```

This restriction makes it difficult to use the text interpreter during program execution as it requires floating point numbers to contain 'D' or 'E' indicators, which is not profane practice. A quick kluge to fix this is to change `FNUMBER?` as below.

```
Replace:
  fcheck drop if                        \ valid f.p. number?
with:
  fcheck or if                          \ valid f.p. number?
```

Note that this change can/will cause problems if number base is not `DECIMAL`.

13.13 Changes from v6.0 to v6.1

Renamed `DINT` to `F>D` for consistency. `F>D` is the ANS word. The original `F>D` was just a synonym. Similarly `SINT` was renamed to `F>S`.

The word `FLOATS` that enabled floating point number conversion has been renamed to `REALS` to avoid a name conflict with the ANS word of the same name.

The `F-PACK` vocabulary has been removed as no one liked it, and it could be considered contrary to the ANS Forth specification. If you wish to retain the `F-PACK` vocabulary, add the following lines before and after the compilation of the floating point code:

```
only forth definitions          \ *** added ***
vocabulary f-pack               \ *** added ***
also f-pack definition          \ *** added ***
include %CommonDir%\Sfp32Hi     \ primitives
include %CommonDir%\Sfp32Com    \ common high level code
previous definitions            \ *** added ***
```

The code enabling floating point to work in degrees or radians has been commented out for ANS compatibility. All trig functions now operate in radians. The commented out code may be uncommented if you need backward compatibility.

13.13.1 32 bit targets: software floating point

Overhauled 32 bit software floating point and incorporated improvements contributed by Hiden Analytical. These include more complete special case detection, faster high level code, and more accurate number input and output.

Removed all use of global variables except `PLACES` to make the floating point code usable in interrupt routines and in multitasked systems. If the output routines are to be multitasked, change the definition of `PLACES` from:

```
VARIABLE PLACES  8 PLACES !
```

to:

```
CELL +USER PLACES
```

and remember to initialise `PLACES` before using the floating point output routines.

Many words that are only useful as factors have been made headerless to save target memory space.

13.13.2 16 bit targets: software floating point

Note that the 16 bit floating point pack is not re-entrant. If you need to use the floating point pack in a multitasking system, you should convert the global variables to `USER` variables. The word `+USER` can be used

```
<size> +USER <name>
```

to define a `USER` variable of a given size (normally a CELL) at the next free offset in the USER area. Only `PLACES` will need initialisation.

13.14 High Level primitives

The software floating point pack requires several support primitives. High level versions are provided in *SFP16HI.FTH* and *SFP32HI.FTH* for 16 and 32 bit targets. Some targets have coded versions in the CPU directory and these will provide much better performance. The support file should be compiled before the common file.

```
: <<1           \ n -- n<<1
```

A compiler synonym for `2*` or `1 LSHIFT`.

```
: >>1           \ n -- n>>1
```

A compiler synonym for `u2/` or `1 RSHIFT`.

`: S->  \ n1 carry-in-flag --- n2 carry-out-flag`
Perform a right shift, applying the carry in to the m.s. bit and returning the carry out as 1 or 0.

`: <-S  \ n1 carry-in-flag --- n2 carry-out-flag`
Perform a left shift, applying the carry in to the l.s. bit and returning the carry out as 1 or 0.

`: d<<1  \ xd -- xd<<1`
One bit double left shift.

`: d>>1  \ xd -- xd>>1`
One bit double right logical shift.

`: D>>N  \ d m -- d>>m`
M bit double right logical shift.

14 Time Delays

The code in *Common\Delays.fth* allows you to handle time delays specified in milliseconds. If you use the multitasker or *Common\Timebase.fth*, *Common\Delays.fth* should be compiled after them.

```
: pause          \ -- ; multitasker hook
```
Allows the sytem multitasker to get a look in. If the multitasker has not been compiled, `PAUSE` is defined as a compiler `NOOP`.

```
0 value ticks    \ -- n
```
Return current clock value in milliseconds. This value can treated as a 32 bit unsigned value that will wrap when it overflows.

```
: later          \ n -- n'
```
Generates the timebase value for termination in n millseconds time.

```
: expired        \ n -- flag ; true if timed out
```
Flag is returned true if the timebase value n has timed out. N.B. Calls `PAUSE`.

```
: timedout?      \ n -- flag ; true if timed out
```
Flag is returned true if the timebase value n has timed out. `TIMEDOUT?` does not call `PAUSE`, so it can be used in interrupt handlers. In particular, `TIMEDOUT?` should be used rather than `EXPIRED` inside timer action words to reduce timer jitter.

```
: ms             \ n --
```
Waits for n milliseconds. Uses `PAUSE` through `EXPIRED`.

15 Periodic Timers

This code provides a timer system that allows many timers The Forth words in the user accessible group documented below are compatible with the token definitions for the PRACTICAL virtual machine, with the code supplied with MPE's embedded targets, and with VFX Forth. This code assumes the presence of a global value `TICKS` which holds a time value incremented in milliseconds. The timebase is approximate. Granularity and jitter are affected by the timer ISR and the time taken by your own code to execute. By default, the timer is set to run every 10..100 ms. The source code is in the the file *TIMEBASE.FTH*.

The file *DELAYS.FTH* should be compiled after *TIMEBASE.FTH*. The code to start and stop the timebase system is part of the ticker interrupt system, which is compiled after *DELAYS.FTH*. If you need to write a new ticker interrupt handler, there will be examples to start from in the <*CPU*>*DRIVERS* folder. The required compilation order is this:

```
multitasker (optional)
TIMEBASE.FTH (optional)
DELAYS.FTH
Ticker driver
```

The timer chain is built using a buffer area, and two chain pointers. Each timer is linked into either the free timer chain, or into the active timer chain.

All time periods are in milliseconds. Note that on a 32 bit system, these time periods must be less than 2^31-1 milliseconds, say 596 hours or 24 days, whereas if the code is on a 16 bit system, time periods must be less than 2^15-1 milliseconds, say 32 seconds.

15.1 The basics of timers

These basic words are defined for applications to use the timer system. Other words are detailed elswhere in this chapter.

```
START-TIMERS       \ -- ; must do this first
STOP-TIMERS        \ -- ; closes timers
AFTER              \ xt period -- timerid/0 ; runs xt once after period ms
EVERY              \ xt period -- timerid/0 ; runs xt every period ms
TSTOP              \ timerid -- ; stops the timer
MS                 \ period -- ; wait for period ms
```

After the timers have been started, actions can be added. The example below starts a timer which puts a character on the debug console every two seconds. Note that when using generic I/O, the output and input devices MUST be specified.

```
start-timers
: t        \ -- ; will run every 2 seconds
  console opvec !
  [char] * emit
;
' t 2 seconds every  \ returns timer id, use TSTOP to stop it
```

The item on stack is a timer handle, use TSTOP to halt this timer.

AFTER is very useful for creating timeouts, such as required to determine if something has happened in time. AFTER returns a timerid. If the action you are protecting happens in time, just use TSTOP when the action happens, and the timer will never trigger. If the action does not happen, the timer event will be triggered.

15.2 Considerations when using timers

All timers are executed within a single interrupt, and so all timer action words share a common user area. This has some impact on timer action words. Since you do not know in which order timer action words are executed, you must set up any USER variables such as BASE that you use, either directly or indirectly.

The interrupt that handles all the timers does not set IPVEC and OPVEC to a default value. If you use I/O words such as EMIT and TYPE within a timer action, you MUST set IPVEC and OPVEC before using the I/O. For the sake of other timer action routines that may still be using default I/O, it is polite to save and restore IPVEC and OPVEC in your timer action words.

Do not worry about calling TSTOP with a timerid that has already been executed and removed from the active timer chain; if TSTOP cannot find the timer, it will ignore the request.

Under some conditions, the execution time of all the timer routines may be longer than the requested period of the timer. In addition, the timer interrupt may be subject to jitter.

15.3 Implementation issues

The following discussion is relevant if you want to modify this code. Functionally equivalent code is provided with MPE's VFX Forth systems. In the Windows environment, timer interrupts are implemented by callbacks and critical sections.

By default, the word DO-TIMERS is run from within the periodic timer interrupt. If interrupts are not re-enabled after resetting the timer interrupt, you may have latency issues if a number of timers is used, or if one of the timer routines takes a considerable time. In this case, it would be better to set up the timer routine to RESTART a task which calls DO-TIMERS,e.g.

```
: TIMER-TASK    \ --
  <initialise>
  BEGIN
    DO-TIMERS STOP
  AGAIN
;
```

Such a strategy also permits you to use a fast interrupt, say 1ms, for the clock, and to trigger the TIMER-TASK every say 32 ms.

15.4 Timebase glossary

```
0 value ticks   \ -- addr ; holds timer count
```
Get current clock value in milliseconds.

```
#8 constant #timers     \ -- n ; maximum number of timers
```
A constant used at compile time to set the maximum number of timers required. Each timer requires RAM as defined by the ITIMER structure.

```
: do-timers     \ --
```
Process all the timers in the chain

```
: after         \ xt period -- timerid/0 ; xt is executed once,
```
Starts a timer that executes once after the given period. A timer ID is returned if the timer could be started, otherwise 0 is returned.

```
: every         \ xt period -- timerid/0 ; periodically
```
Starts a timer that executes every given period. A timer ID is returned if the timer could be started, otherwise 0 is returned. The returned timerID can be used by TSTOP to stop the timer.

```
: tstop         \ timerid --
```
Removes the given timer from the active list.

16 Vocabulary and wordlist tools

`: VOC?  \ wid -- flag`
Return TRUE if 'wid' is actually a vocabulary.

`: .VOC  \ wid --`
If wid represents a vocabulary, display its name, otherwise just display its value.

`: ORDER  \ --`
Display the current search order and definitions vocabularies.

`: VOCS  \ --`
Display all vocabularies.

`: $FORGET  \ c-addr --`
Forgets word name in given string. See FORGET.

`: FORGET  \ "<spaces>name" --`
Used in the form "FORGET <name>", <name> and all following words are removed from the dictionary. This word is marked obsolescent in the ANS specification, and is replaced by MARKER.

`: MARKER  \ "<spaces>name" -- ; Exec: --`
MARKER <name> creates a word that when executed removes itself and ALL following definitions from the dictionary. MARKER is the ANS replacement for FORGET. MARKER automatically trims all wordlist and vocabulary based chains.

17 Character Queues

The file *Common\Cqueues.fth* provides circular character (byte) queues. If the equate `TASKING?` is non-zero, the blocking routines will use `PAUSE`. Interrupts are disabled for the queue empty/full checks.

17.1 Queue data structure

`struct /cqueue  \ -- size ; character queue structure in idata, buffer in udata`
Circular queue data structure.

```
  int >qhead            \ Offset of head, where characters are taken from
  int >qtail            \ Offset of tail, where characters are put
  int >qchars           \ Number of characters in the queue
  int >qmask            \ Mask to apply to pointers
  ptr >qbuffer          \ Base address of character buffer
end-struct
```

`: cqueue:       \ size -- ; -- cqueue ; size CQUEUE: <name>`
An interpreter definition to build a character queue of the specified size. The queue data structure is built in the current `IDATA` space, and the buffer itself is in the current `UDATA` space. Executing `<name>` returns the address of the queue data structure. N.B. The size of a queue must be a power of two, e.g. 32, 64 ...

`: init-cqueue   \ cqueue -- ; initialise queue`
Initialise the specified queue created by `CQUEUE:`.

`: init-hcqueue  \ size cqueue -- ; SFP002`
Initialise the specified queue created by `ALLOCATE`. When a queue is allocated from the heap by a phrase of the form `/CQUEUE <size> + ALLOCATE`, this word must be used.

17.2 Queue primitives

`: (>cqueue)     \ char cqueue -- ; put character on cqueue`
Put char into the queue with no checks.

`: (cqueue>)     \ cqueue -- char ; get next character from queue`
Get the next character from the queue with no checks.

`: (cqfull?)     \ queue -- flag ; TRUE if queue full`
Return true if the queue is full. No interrupt protection is provided.

`: cqfull?       \ queue -- flag ; TRUE if queue full`
Return true if the queue is full. Interrupt protection is provided.

`: cqchars       \ queue -- n`
Return the number of characters in the queue.

`: cqempty?      \ queue -- flag ; TRUE if queue empty`
Return true if the queue is empty.

`: cqnotempty?   \ queue -- flag ; TRUE if queue not empty`
Return true if the queue is not empty, i.e. if it contains any characters.

`: cqRoom        \ queue -- u`
Return the number of free bytes in the queue.

`: cqroom?  \ +n queue -- flag`
Return true if there is enough room in the queue for +n more characters.

`: >cqueue  \ char cqueue -- ; spins if full`
Put a character into the queue. If the queue is full, the system waits (blocks) until there is enough space.

`: cqueue>  \ queue -- char ; spins while queue empty`
Remove the next character, waiting if the queue is empty.

17.3 Extended queue operations

When queues are filled and emptied in blocks rather than character by character, faster operation can be achieved using these words. You may see additional benefit using the fast version of `CMOVE`.

The extended operations are compiled if the equate `extCQ?` is set non-zero before *Cqueues.fth* is compiled.

`0 equ extCQ?  \ -- flag`
If set non-zero here or before *Cqueues.fth* is compiled, the extended wordset below will be compiled.

17.3.1 Primitives

`: /tailw  \ queue -- len`
Number of bytes from the tail to the end.

`: /gapw  \ queue -- len`
Number of bytes from the tail to the head, assuming no wrap.

`: +tail  \ n queue --`
Add n characters to the tail pointer and count.

`: $>cq  \ caddr len n queue -- caddr' len' queue`
Copy *n* bytes of the the source block into the buffer.

`: $>tail  \ caddr len queue -- caddr' len' queue`
Add block in space between tail and end

`: $>gap  \ caddr len queue -- caddr' len' queue`
Add block in space between tail and head.

`: /headr  \ queue -- len`
Number of bytes from the head to the end.

`: /gapr  \ queue -- len`
Number of bytes from the head to the tail, assuming no wrap.

`: -head  \ n queue --`
Remove n characters from the head pointer and count.

`: cq>$  \ caddr len n queue -- caddr' len' queue`
Copy *n* bytes from the queue into the buffer *caddr/len.*

`: head>$  \ caddr len queue -- caddr' len' queue`
Copy queue block in space between head and end

`: gap>$  \ caddr len queue -- caddr' len' queue`
Add block in space between head and tail.

17.3.2 User words

```
: cqWrite       \ caddr len queue --
```
Write the given string to the queue. This word blocks until the operation is complete.

```
: cqStuff       \ caddr len queue -- n
```
Write as much data from the given string to the queue as there is room for. Return the number of bytes written.

```
: cqRead        \ caddr len queue --
```
Read the given string from the queue. This word blocks until the operation is complete.

```
: cqExtract     \ caddr len queue -- n
```
Read data from the queue to the buffer. The amount read is limited by the size of the buffer and the number of bytes in the queue. The number of bytes read is returned.

18 Kinetis K60 interrupt queued serial driver

The file *Cortex\Drivers\serK60qi.fth* provides interrupt and queued serial drivers for the on-chip UARTs.

Primary testing was performed using UART3 on a Tower system with a TWR-K60N512 CPU module and a TWR-SER module. Make sure that your `init-sern` words initialise the pin registers to the configuration that you are using.

18.1 Initialisation

The baud rate generator value is defined by:

```
baudrate=moduleclock/(16*(SBR+BRFD))
```

where SBR is a 13 bit integer and BRFD is a 5 bit fraction. To use this, we rarrange the equation to the form:

```
(SBR+BRFD)=moduleclock/(16*baudrate)
```

In order to use integer operations, we scale this.

```
brgval=2*moduleclock/baudrate
```

`: genBRG \ baud clock -- brgval`
Generate the combined baud rate divisor

`: brg>rlh \ brgval -- brfa brl brh`
Generate the three portions of the baud rate divisor.

`: genBRG \ baud clock -- brgval`
Generate the combined baud rate divisor

`: brg>rlh \ brgval -- brfa brl brh`
Generate the three portions of the baud rate divisor.

`: (init-ser) \ brgval ^uart --`
Initialise the given UART to 8,N,1 format with no flow control and with given baud rate divisor.

18.2 Serial primitives

`: >RxQ \ char queue --`
Put character from UART into queue. INTERNAL.

`: FIFO>RxQ \ base queue --`
Given a UART base address and a character queue, empty the UART Rx FIFO into the queue. INTERNAL.

`: TxQ>FIFO \ base queue --`
Given a UART base address and a character queue, put a transmit character into the UART Tx FIFO.

`0 value FaultConsole? \ -- flag`
When set non-zero, transmit is in polled mode. This is required when fault exceptions provide debug information.

`: +FaultConsole \ --`

For use in fault exception handlers, the multi-tasker must be turned off and *\fo{EMIT) and friends must run in polled mode.

`: (qseremit)    \ char base queue --`
Send a character on the UART whose base address and queue are given.

`: (qsertype)    \ caddr len base queue --`
Send a string on the UART whose base address and queue are given.

`: (qsercr)      \ base queue --`
Send a CR/LF on the UART whose base address and queue are given.

18.3 UART0

Make sure that `init-ser0` initialises the pins to the configuration that you are using.

`console0-speed system-speed GenBRG equ u0-brgval        \ -- n`
UART0 baud rate divisor.

`#128 cqueue: SerInpQ0   \ -- addr`
Receiver circular byte queue.

`#64 cqueue: SerOutQ0    \ -- addr`
Transmitter circular byte queue.

`: ser0-isrh     \ --`
UART0 high level ISR. INTERNAL.

`' ser0-isrh INT_UART0_RX_TX EXC: ser3-isr`
UART0 interrupt service routine entry point.

`: init-ser0     \ --`
Initialise serial port 0.

`: seremit0      \ char --`
Transmit a character on UART0.

`: sertype0      \ c-addr len --`
Transmit a string on UART0.

`: sercr0        \ --`
Issue a CR/LF pair to UART0.

`: serkey?0      \ -- flag`
Return true if UART0 has a character available.

`: serkey0       \ -- char`
Wait for a character on UART0.

`create Console0 \ -- addr ; OUT managed by upper driver`
Generic I/O device for UART0.

```
  console0 constant console
```

`CONSOLE` is the device used by the Forth system for interaction. It may be changed by one of the phrases of the form:

```
  <device>  dup opvec !  ipvec !
  <device>  SetConsole
```

18.4 UART1

Make sure that `init-ser1` initialises the pins to the configuration that you are using.

`console1-speed GenBRG equ u1-brgval \ -- n`
UART1 baud rate divisor.

`#128 cqueue: SerInpQ1 \ -- addr`
Receiver circular byte queue.

`#64 cqueue: SerOutQ1 \ -- addr`
Transmitter circular byte queue.

`: ser1-isrh \ --`
UART1 high level ISR. INTERNAL.

`' ser1-isrh INT_UART1_RX_TX EXC: ser1-isr`
UART1 interrupt service routine entry point.

`: init-ser1 \ --`
Initialise serial port 1.

`: seremit1 \ char --`
Transmit a character on UART1.

`: sertype1 \ c-addr len --`
Transmit a string on UART1.

`: sercr1 \ --`
Issue a CR/LF pair to UART1.

`: serkey?1 \ -- flag`
Return true if UART1 has a character available.

`: serkey1 \ -- char`
Wait for a character on UART1.

`create Console1 \ -- addr ; OUT managed by upper driver`
Generic I/O device for UART1.

18.5 UART2

Make sure that `init-ser2` initialises the pins to the configuration that you are using.

`console2-speed GenBRG equ u2-brgval \ -- n`
UART2 baud rate divisor.

`#128 cqueue: SerInpQ2 \ -- addr`
Receiver circular byte queue.

`#64 cqueue: SerOutQ2 \ -- addr`
Transmitter circular byte queue.

`: ser2-isrh \ --`
UART2 high level ISR. INTERNAL.

`' ser2-isrh INT_UART2_RX_TX EXC: ser2-isr`
UART2 interrupt service routine entry point.

`: init-ser2 \ --`

Initialise serial port 2.

`: seremit2 \ char --`
Transmit a character on UART2.

`: sertype2 \ c-addr len --`
Transmit a string on UART2.

`: sercr2 \ --`
Issue a CR/LF pair to UART2.

`: serkey?2 \ -- flag`
Return true if UART2 has a character available.

`: serkey2 \ -- char`
Wait for a character on UART2.

`create Console2 \ -- addr ; OUT managed by upper driver`
Generic I/O device for UART2.

18.6 UART3

On the K60 Tower system, the console is implemented on UART3 connected through the DB9 connector on the TWR-SER board.

`console3-speed bus-speed GenBRG equ u3-brgval \ -- n`
UART3 baud rate divisor.

`#128 cqueue: SerInpQ3 \ -- addr`
Receiver circular byte queue.

`#64 cqueue: SerOutQ3 \ -- addr`
Transmitter circular byte queue.

`: ser3-isrh \ --`
UART3 high level ISR. INTERNAL.

`' ser3-isrh INT_UART3_RX_TX EXC: ser3-isr`
UART3 interrupt service routine entry point.

`: init-ser3 \ --`
Initialise serial port 3.

`: seremit3 \ char --`
Transmit a character on UART3.

`: sertype3 \ c-addr len --`
Transmit a string on UART3.

`: sercr3 \ --`
Issue a CR/LF pair to UART3.

`: serkey?3 \ -- flag`
Return true if UART3 has a character available.

`: serkey3 \ -- char`
Wait for a character on UART3.

`create Console3 \ -- addr ; OUT managed by upper driver`
Generic I/O device for UART3.

18.7 UART4

Make sure that `init-ser4` initialises the pins to the configuration that you are using.

`console4-speed bus-speed GenBRG equ u4-brgval  \ -- n`
UART4 baud rate divisor.

`#128 cqueue: SerInpQ4  \ -- addr`
Receiver circular byte queue.

`#64 cqueue: SerOutQ4  \ -- addr`
Transmitter circular byte queue.

`: ser4-isrh  \ --`
UART4 high level ISR. INTERNAL.

`' ser4-isrh INT_UART4_RX_TX EXC: ser4-isr`
UART4 interrupt service routine entry point.

`: init-ser4  \ --`
Initialise serial port 4.

`: seremit4  \ char --`
Transmit a character on UART4.

`: sertype4  \ c-addr len --`
Transmit a string on UART4.

`: sercr4  \ --`
Issue a CR/LF pair to UART4.

`: serkey?4  \ -- flag`
Return true if UART4 has a character available.

`: serkey4  \ -- char`
Wait for a character on UART4.

`create Console4 \ -- addr ; OUT managed by upper driver`
Generic I/O device for UART4.

18.8 UART5

Make sure that `init-ser0` initialises the pins to the configuration that you are using.

`console5-speed bus-speed GenBRG equ u5-brgval  \ -- n`
UART5 baud rate divisor.

`#128 cqueue: SerInpQ5  \ -- addr`
Receiver circular byte queue.

`#64 cqueue: SerOutQ5  \ -- addr`
Transmitter circular byte queue.

`: ser5-isrh  \ --`
UART5 high level ISR. INTERNAL.

`' ser5-isrh INT_UART5_RX_TX EXC: ser5-isr`
UART5 interrupt service routine entry point.

`: init-ser5  \ --`

Initialise serial port 5.

```
: seremit5      \ char --
```
Transmit a character on UART5.

```
: sertype5      \ c-addr len --
```
Transmit a string on UART5.

```
: sercr5        \ --
```
Issue a CR/LF pair to UART5.

```
: serkey?5      \ -- flag
```
Return true if UART5 has a character available.

```
: serkey5       \ -- char
```
Wait for a character on UART5.

```
create Console5 \ -- addr ; OUT managed by upper driver
```
Generic I/O device for UART5.

18.9 System initialisation

```
: init-ser      \ --
```
Initialise all serial ports

19 Freescale K60 GPIO utilities

The code in *Cortex\Drivers\gpioK60.fth* provides utility words for accessing GPIO pins on a bit by bit basis. The code is written for ease of use rather than performance.

19.1 Defining I/O pins

FIO/GPIO acess is defined using a bit number. Bits 0..31 form P0.0 to P0.31, bits 32..63 form P1.0 to P1.31 and so on. Although Freescale name their ports from A, here we number them from 0.

```
create PortTable        \ -- addr
```
Holds data about each port.

```
: PIO:          \ port# bit# -- ; -- iobit#
```
Define a port I/O bit by name. For example

```
  2 11 PIO: P2.11
```

19.2 GPIO pin access

```
: setPin        \ struct --
```
Set the pin high.

```
: clrPin        \ struct --
```
Set the pin low.

```
: getPin        \ struct -- 0/1
```
Read the pin state.

19.3 IO pin configuration

```
: setPinCtrl    \ value mask struct --
```
Apply value and mask to the pin's control register.

```
: isPinMode     \ mode struct --
```
Sets the GPIO mode for a pin. Use with the constants below.

```
: pinBit        \ 0/nz bit# struct --
```
Set/clear the given bit in the control regster.
Pins can be in one of eight modes, 0..7. In the majority of cases, mode 0 means disabled and mode 1 means GPIO mode. Most multi-function pins are disabled after reset. Consult the Chip Configuration chapter of the chip's Reference Manual for the full details.

```
0 constant UnusedMode   \ -- mode#
```
The pin is unused.

```
1 constant GPIOmode     \ -- mode#
```
Specifies GPIO mode.

```
: isInput       \ struct --
```
Set the pin to GPIO mode and input.

```
: isOutput      \ struct --
```
Set the pin to GPIO mode and output.

```
: isPinStrength \ 0/1 struct --
```
Set the pin driver to low (0) or high (nz) drive.

```
: isPinOD       \ 0/1 struct --
```
Enable (0) or disable (nz) the open drain driver.

```
: NotPulled     \ struct --
```
No pull up or pull down.

```
: PulledUp      \ struct --
```
Enable the pin pull up resistor.

```
: PulledDown    \ struct --
```
Enable the pin pull down resistor.

19.4 Test code for Freescale TWR-K60N512

```
0 [if]
decimal

0 11 pio: led1          \ GPA.11
0 28 pio: led2          \ GPA.28

: tled1         \ --
  led1 isOutput
  begin
    led1 clrPin  200 ms
    led1 setPin  200 ms
  key? until
;

: tled2         \ --
  led2 isOutput
  begin
    led2 clrPin  200 ms
    led2 setPin  200 ms
  key? until
;
[then]
```

20 Rebooting the CPU

The word `REBOOT` permits the system to be reset by disabling all interrupts and activating the SYSRESETREQ bit in the NVIC AIRCR register.

`: reboot \ --`
Reboots the CPU by activating the watchdog.

21 M4 System Ticker

The system ticker uses the Cortex-M4 *SysTick* timer. Although the *SysTick* timer is common to all Cortex-M4 devices, there is one clocking option. This code does not use that option and so may be used by all CM4 devices.

The CM4 implementation is fed with the system clock and has an option controlled by bit 2 of the SysTick Control and Status Register (ARM DDI 0337, Rev E, page 8-8, ARM DDI 0403D, page B3-746/7).

- Bit2=0: SysTick clock is vendor defined
- Bit2=1: SysTick clock is CCLK.

This code always uses CCLK.

```
tick-ms system-speed #1000 */ 1- equ /SysTick   \ -- x
```
Initial value of the SysTick Reload register.

```
timebase? [if]  \ needs high-level ISR
```
If the timebase option is being used, we need the high-level ISR handler.

```
: SysTicker     \ --
```
High level version of ticker ISR.

```
' SysTicker SysTickvec# EXC: SysTickISR \ -- addr
```
Setting the high level ISR.

```
Proc SysTickISR \ --
```
Ticker ISR in assembler.

```
: start-clock   \ --
```
Initialise the system ticker to run with a period of `tick-ms` milliseconds.

```
: stop-clock    \ --
```
Stop the sytem ticker.

```
: ms            \ n --
```
Waits for n milliseconds.

```
: start-timers  \ -- ; Start internal time clock
```
Initialise the TIMEBASE system.

```
: stop-Timers   \ -- ; disable timer system
```
Stop the TIMEBASE ticker.

22 Kinetis K60 Real Time Clock

22.1 RTC access

```
: initRTC       \ --
```
Initialise RTC - does not affect current state or contents. Uses the 32kHz crystal.

```
: xsecs>RTC     \ frac int --
```
Write the seconds and fraction of seconds values to the RTC.

```
: RTC>xsecs     \ -- frac int
```
Read the seconds and fractional seconds from the RTC.

```
_RTC rtcTSR + constant lseconds \ -- addr
```
The address of the seconds counter.

```
0 value dow     \ -- u
```
Return the day of week number where* 0 is Sunday, 1 is Monday and so on. Set as a result of running `Time&Date` below.

22.2 LSECONDS conversions

When you have LSECONDS, you need conversion between Unix seconds and the time/data functions supported by the RTC. This code is taken from *UnixTime.fth* and is compiled if the equate `ConvertSecs?` is true.

```
#24 #60 * #60 * equ secs/day    \ -- 86400
```
Seconds per day.

```
#60 #60 * equ secs/hr           \ -- 3600
```
Seconds per hour

```
#60 equ secs/min                \ -- 60
```
Seconds per minute

```
#60 equ mins/hr                 \ -- 60
```
Minutes per hour

```
#24 equ hrs/day                 \ -- 24
```
Hours per day

```
#34698 equ uday0                \ -- 34698
```
The Julian day number for Unix day 0 at 1 Mar 1995.

```
: j>uday        \ jday -- uday
```
Convert a Julian day based at 0 for 1 March 1900 to a Unix day number derived from `LSECONDS`.

```
: u>jday        \ uday -- jday
```
Convert a Unix day number as used by `LSECONDS` to a Julian day based at 0 for 1 March 1900.

```
: >date         \ jday -- day month year
```
Convert a Julian day number to day month year format.

```
: date>         \ day month year -- jday ; see month codes
```
Given day month and year generate the day number.

```
: /umod         \ u1 u2 -- urem uquot
```
Single precision division of unsigned U1 by U2 returning both remainder and quotient.

`: weekday \ jday -- n ; 0=Sunday`
Given a Julian day return the day of week number where 0 is Sunday, 1 is Monday and so on.

`: secs>td \ lsecs -- sec min hr day mon yr`
Convert a Unix seconds value to time and date.

`: td>secs \ sec min hr day mon yr -- secs`
Use the given values for time and date, and return the Unix `LSECONDS` value.

`: time&date \ -- sec min hr day mon yr ; 10.6.2.2292`
The ANS Forth word to return time and date. Sets `DOW` as a side effect.

`: settime&date \ sec min hr day mon yr --`
Use the given values for time and date, and set `LSECONDS` and any underlying RTC hardware.

22.3 Time and date output

`: .2d \ u --`
Display u as two digits with leading zero.

`: (.time) \ secs mins hours --`
Display given time.

`create months$ \ -- addr`
String containing 3 character text for the months.

`: (.date) \ day month year --`
Display given date.

`create days$ \ -- addr`
String containing 3 character text for the days of the week.

`: .dow# \ dow --`
Display day of week by name.

`: .time&date \ --`
Display the day of week, date and time.

`: .time \ --`
Read and display time.

`: .date \ --`
Read and display date.

`: .AnsiDate \ zone --`
Display the day of week, date and time. If zone is 0 GMT is displayed. The format is:

```
dow, hh:mm:ss dd Mmm yyyy [GMT]
```

23 Ethernet driver for Kinetis K60 devices

The driver in *Cortex\Drivers\EtherK60.fth* was tested on a Freescale Tower System with TWR-K60N512 and TWR-SER modules. Only RMII mode has been tested. The 1588 timestamp modes are not implemented.

23.1 Configuration

The following must be defined before the file *Drivers\EtherK60.fth* is compiled.

```
: const equ ;   \ n -- ; -- n
```

If you are using a standalone target with heads and you want interactive access to all the registers and bit masks, define `CONST` as `CONSTANT`, otherwise by default `CONST` is defined as `EQU`.

```
0 equ EtherCom? \ -- flag
```

Only needed if the EtherCom generic I/O device is being used. This only happens for systems with multiple IP ports and PowerNet v3.x upwards.

```
1 equ EtherDiags?       \ -- flag
```

Set this equate true to compile diagnostic code for register dumping and so on. False by default.

```
1 equ MultiPhy? \ -- flag
```

Set this flag true if you are using multiple PHYs or do not know the address of the PHY.

```
0 value Phy#    \ -- n
```

Multi PHY: The number of the PHY in use: 0..31

```
0 equ Phy#      \ -- n
```

Single PHY: The number of the PHY in use: 0..31. For the TWR-K60N512 and TWR-SER use 0.

```
1 equ FindPhy?  \ -- flag
```

Set this flag true if you want to scan for the PHY address, because it varies or you don't know it. Use only if you have a single PHY connected and `MultiPhy?` must also be set to non-zero.

```
1 equ RMII?     \ -- n
```

Set non-zero if using RMII rather than the MII interface.

```
1 equ initPHY?  \ -- x
```

If it is difficult to get the PHY to respond during testing, set `initPHY?` to zero. For normal use it should be set to 1.

```
0 equ sniff?    \ -- flag
```

Set this equate true to compile the packet sniffer code, which can be used to test Ethernet reception.

```
0 equ pingit?   \ -- flag
```

Set this equate true to compile the PING code, which tests Ethernet transmission and reception.

```
create EtherAddress     \ -- addr
```

Holds the Ethernet MAC address (six bytes). Note that you must obtain these from the IEEE (www.ieee.org) or from other sources.

```
create IpAddress        \ -- addr
```

Holds the Ethernet IP address (four bytes).

23.2 Buffers

```
#1518 equ /EPacketMax   \ -- len
```

Max length of a normal Ethernet packet.

```
#1536 equ /ERxBuff      \ -- u
```

Size of an Ethernet receive buffer rounded up to 16 byte unit. If the buffers are in cached memory, they should also be rounded to a cache line (64 bytes). Although 1520 (1518 rounded up) is 16 byte aligned, it permits normal frames but not VLAN frames.

```
#4 equ #ERxBuffs                \ -- u
```

Number of receive buffers. Must be at least two.

```
#1536 equ /ETxBuff      \ -- u
```

Size of an Ethernet transmit buffer rounded up to 16 byte unit.

```
#2 equ #ETxBuffs                \ -- u
```

Number of transmit buffers. Must be at least one. Transmit buffers are managed in software.

```
struct /EDesc   \ -- len
```

Structure defining an Ethernet transmit/receive descriptor. Note that the contents are in big-endian format.

```
  2 field ed.status             \ 2 byte status
  2 field ed.len                \ 2 byte length
  4 field ed.addr               \ address of buffer - 16 byte aligned
end-struct


/Edesc #ErxBuffs * buffer: ERxDescs     \ -- addr
```

Ethernet receive descriptors.

```
/Edesc #ETxBuffs * buffer: ETxDescs     \ -- addr
```

Ethernet transmit descriptors.

```
/ERxBuff #ERxBuffs * buffer: ERxBuffs   \ -- addr
```

Ethernet receive buffers.

```
/ETxBuff #ETxBuffs * buffer: ETxBuffs   \ -- addr
```

Ethernet transmit buffers.

```
variable FirstRxDesc    \ -- addr
```

Holds the address of the first read descriptor to check.

```
variable NextRxDesc     \ -- addr
```

Holds the address of the next read descriptor containing unhandled data. Set to zero to indicate none or unknown.

```
variable NextTxDesc     \ -- addr
```

Holds the address of the next transmit descriptor to check.

```
: initRxDescs   \ --
```

Initialise the receive descriptors.

```
: initTxDescs   \ --
```

Initialise the transmit descriptors.

23.3 Tools

23.4 PHY access

`$60020000 equ ~RdPhy \ -- mask`
Mask for PHY read operations. Assumes use of PHY0.

`$50020000 equ ~WrPhy \ -- mask`
Mask for PHY write operations. Assumes use of PHY0.

`: ~Phy# \ -- mask`
Multi PHY: Mask used for PHY command generation.

`Phy# #23 lshift equ ~Phy# \ -- mask`
Single PHY: Mask used for PHY command generation.

`: WaitPhyDone \ --`
Wait until the current PHY operation has completed.

`: ReadPhy \ reg -- w`
Read a PHY register. Note that these are 16 bit values.

`: WritePhy \ w reg --`
Write a PHY register. Note that these are 16 bit values.

`: OrPhy \ mask reg --`
Or mask into reg in the PHY. Set bits in mask are set in the register.

`: InitPhy \ -- ; autonegotiation version`
Initialise the PHY with autonegotiation

`: EtherLink? \ -- flag`
Return true if the Ethernet link is established. For most PHYs, this is performed by testing bit 2 of register 1.

`: FindPhy \ -- phy|-1`
Scan for PHYs. Return the number of the first PHY found, or -1 if no PHY is found. `Phy#` is set for each PHY tested.

23.5 Initialisation

`equ initMSCR \ -- x`
Initial value of MSCR.

`: setMACAddress \ addr --`
Set the MAC address in the EMAC unit.

`$0400 equ Alt4 \ -- x`
Port control register value for MUX=4.

`: setEtherPins \ -- ; RMII`
Set the K60 pins required for RMII mode. These are:

`RMII_MDC RMII_MDIO RMII_CRS_DV RMII_RXER RMII_TXEN`

`RMII_RXD[1:0] RMII_TXD[1:0]`

`RMII_REF_CLK 1588_TMRn ENET_1588_CLKIN`

`: setEtherPins \ -- ; MII`
Set the K60 pins required for MII mode. These are:

`MII_COL MII_CRS MII_RXCLK MII_TXCLK MII_TXER`

`MII_MDC MII_MDIO MII_RXDV MII_RXER RMII_TXEN`

```
  MII_RXD[1:0]  RMII_TXD[1:0]
  1588_TMRn ENET_1588_CLKIN
```

`: setPHYdata    \ --`
Collect the results of initialising the PHY and apply them to the MAC.

`: initEther     \ --`
Initialise the FEC.

23.6 Packet read/write

`: IsRx?         \ -- flag ; nz = has data`
Check if incoming data is present, returning true if a packet is available for `get_ether_pkt` below.

`: get_ether_pkt \ *dest maxlen -- len`
Get a pending receive packet to a buffer of size *\b{maxlen). Return the number of bytes received, or zero for a bad packet. Note that data is valid only when `ISRX?` returns true, so that you must poll with `ISRX?` before using `GET_ETHER_PKT`. The received size of the packet (not clipped to the buffer size) is returned. In this implementation, a destination address of 0 indicates that the packet should be discarded, and no memory transfer occurs.

`: DiscardRX     \ --`
Throw away pending input packet due to lack of host buffer memory.

`: IsTx?         \ n -- flag`
Return true if a packet of length n can be sent.

`: send_ether_pkt  \ caddr len --`
Send packet from supplied buffer.

23.7 Generic I/O for PowerNet v3 and above

`: GetAddrs      \ -- ipaddr macaddr 0`
For Generic I/O IPDGetAddr function

`: SetAddrs      \ ipaddr|0 macaddr|0 mode --`
Set up the Ethernet MAC and the IP addresses. At present mode is always 0, but will be used in future releases to indicate data formats. If ipaddr or macaddr are zero, the corresponding stored data will not be changed.
The layout and usage of the Ethernet common I/O structure is defined in the file *COMMON\ETHERCOM.FTH*.

```
create IPdevice
  ' MyInit ,       \ 0: initialisation
  ' MyTerm ,    \ 1: shutdown
  ' MyRx? ,     \ 2: receive test
  ' MyRx                \ 3: receive packet
  ' MyTx? ,     \ 4: transmit test
  ' MyTx ,              \ 5: transmit packet
  ' MyGetAddr ,    \ 6: Get device addresses
  ' MySetAddr ,    \ 7: Set device addresses
  ' MySave ,    \ 8: Save IP device state
```

`create E52xxVector      \ -- addr`
The device vector needed by PowerNet v3+ and *COMMON\ETHERCOM.FTH*.

```
E52xxVector constant IPDevice   \ -- addr
```
`IPDevice` is the default device name required by *\{ETHERCOM.FTH}. If you have multiple ports, only one should be called `IPDevice`.

23.8 Diagnostics

This code is compiled if `EtherDiags?` is non-zero.

```
: .phy          \ --
```
Display contents of all Ethernet Controller's PHY registers.

```
: .phys         \ --
```
Compiled if the equate `MultiPhy?` is non-zero. Displays the register contents of all PHYs.

23.9 System test

The packet sniffer code is only compiled if the equate `SNIFF?` is non-zero.

```
1536 buffer: pbuff      \ -- addr
```
Temporary buffer for `SNIFF`.

```
: NextRxPkt     \ --
```
Collect and display packet

```
: sniff         \ --
```
Listens to the network and displays all the traffic that has a broadcast destination or is for this device. `SNIFF` can be used to test reception.
The PING example is used to test Ethernet transmission and reception. Before testing transmission, test reception by using the maintenance system to download a user application and then pinging the target board instead of TFTPing to it.

```
Example ARP request:  #42/$2A bytes
C020.0600  FF FF FF FF FF FF 00 C0  F0 0D 9D 7E 08 06 00 01
C020.0610  08 00 06 04 00 01 00 C0  F0 0D 9D 7E C0 A8 01 7F
C020.0620  00 00 00 00 00 00 C0 A8  FF FE 11 0A 75 9C C0 A8
```

Note that the equates `SNIFF?` and `PINGIT?` must both be set non-zero to compile the PING code.

```
create requested-ip     \ -- addr
```
Holds IP address to be pinged.

```
create zero-mac         \ -- addr
```
Holds MAC address 00-00-00-00-00-00

```
create EtherBC          \ -- addr
```
Holds MAC address FF-FF-FF-FF-FF-FF

```
: SendArpReq    \ -- ; send ARP request
```
Send an ARP request. All initialisation must have been performed before using this word.

```
: pingonce      \ -- ; an ARP request
```
Send an ARP request and process the response once. All initialisation must have been performed before using this word.

```
: pingit        \ --
```

Ping the IP address at `REQUESTED-IP` until a key is pressed.

24 Kinetis K60 Flash tools

These tools are provided for applications which reserve part of the Flash for data. This code uses the Flash primitives in *FlashFTFL.fth*.

N.B. An erase causes the whole of the sector containing that address to be erased.

24.1 Primitives

`: (doFlashCmd)  \ cob03 cob47 cob8B -- fstat`
Perform a Flash Command and return the status byte. If bit 7 of the status is clear, a timeout occurred. This routine is run from RAM.

`/doFlashCmd buffer: rdoFlashCmd \ -- addr`
The RAM buffer used for the Flash programming code.

`: doFlashCmd    \ cob03 cob47 cob8B -- fstat`
Perform a Flash Command and return the status byte. If bit 7 of the status is clear, a timeout occurred.

`: eraseSector   \ addr -- stat`
Erase the sector containing the address.

`: writeFlash    \ dest len -- stat`
Write *len* bytes from the start of the programming RAM to the Flash at *dest*, returning the raw status byte.

`: writeSector   \ src dest len -- stat`
Write the given data to the sector. *dest* should be the start of the sector.

24.2 User level

`cell buffer: #PrgErrs   \ -- addr`
Holds error count.

`: .src/dest     \ src dest -- src dest`
Display the addresses and contents of the source and destination.

`: EraseUnit     \ dest --`
Erase the flash sector starting at *dest*. On error, the variable `PrgErrs` is incremented.

`: ProgUnit      \ src dest --`
Program `/FlashSector` bytes at *src* to *dest*. On error, the variable `PrgErrs` is incremented.

`: (EraseFlash)  \ dest dlen --`
Erase the flash for the given range. Note that complete sectors in the range will be erased.

`: (ProgFlash)   \ src dest len --`
Write len bytes from memory at src to Flash at *dest*. Note that the Flash is assumed to be erased, and that no verification is performed except when each byte is programmed. *Dest* is forced to a 2k byte boundary and *len* is rounded up to the next 2k byte unit. *Src* must be on a word boundary.

`: (VerifyFlash) \ src dest len --`
Verify len bytes from memory at src to Flash at dest.

`: ProgramFlash  \ src dest len --`

Using the RAM memory buffer at *src*, program the Flash at *dest* with *len* bytes. The relevant Flash sectors are erased, the Flash is programmed, and the result verified.

25 Reprogramming the Kinetis K60 serially

25.1 Introduction

ReprogK60.ctl is the control file for the serial loader which reprograms the on-chip Flash memory. Because the on-chip Flash cannot be accessed during programming, and because the initial loader code may itself be replaced, the Flash programming code is copied into RAM for execution. The only exit from the code is a reset of the CPU to execute the newly Flashed program. All interrupts are disabled during reprogramming, and so polled serial drivers are used.

Access to the reprogramming software is defined in *ReFlash.fth* for the main system code and in this control file. The two files must match.

Return is by rebooting the system. The contents of the internal RAM and Flash are destroyed, therefore any data that must be preserved should be saved externally to the chip.

25.2 Configuration

```
#512 kb equ /MainFlash  \ -- u
```
Define the size of the internal Flash.

25.3 Memory map

```
$1FFF:F000 $1FFF:FEFF cdata section ReprogK60   \ 4k - 256 bytes
$1FFF:FF00 $1FFF:FF7F udata section PROGd       \ 128 bytes
$1FFF:FF80 $1FFF:FFFF udata section PROGu       \ 128 bytes
$1400:0000 $1400:07FF udata section Xbuff       \ 2k byte Xmodem buffer
  sec-base equ Xbase                    \ base address of buffer
  sec-len equ Xlen                      \ length of buffer
  Xbase Xlen + equ Xtop                 \ top of buffer
interpreter
: prog  ReprogK60  ;
target

PROG PROGd PROGu  CDATA                 \ use Code for HERE , and so on
```

25.4 Items of interest

The stacks have already been set up by the calling program. The stacks should not be in the last 4kb of the lower half of the Kinetis SRAM, otherwise they will overlap the program copied into RAM.

```
include %CpuDir%\CortexDef              \ Cortex CPU equates
include %CpuDir%\sfrK60         \ Special function registers


l: CLD1                 \ -- addr
```
Filled in later with the xt of `ReprogFlash`. This label marks the start of the RAM code area.

```
l: CLD_CopyFlash        \ -- addr
```
Filled in later with the xt of `CopyFlash`

```
l: CLD_UART             \ -- addr
```

Filled in later with the UART base address to use. The default is UART3.

```
l: CLD_clockspeed       \ -- addr
```
Contains the clock speed to use. Filled in by the calling application. Default is 96 MHz.

```
l: CLD_/FlUsed          \ -- addr
```
Holds the length of Flash for the reprogram operation. Flash is assumed to start at $0000:0000. Default is 510 kb.

```
l: CLD_/FlSector        \ -- addr
```
Holds the size of a Flash sector. Default is 2 kb.

```
: prog-speed    \ -- hz
```
A macro to fetch the system clock speed from the entry table.

```
include %AppDir%\primitives             \ Forth primitives from CODEARM.FTH
include %AppDir%\MinSerK60p             \ polled serial line driver
include %AppDir%\delays                 \ software delays
include %CpuDir%\Drivers\rebootCM4      \ reboot somehow


synonym ser-emit emit
```
A synonym required by the Xmodem code.

```
synonym ser-key? key?
```
A synonym required by the Xmodem code.

```
synonym ser-key  key
```
A synonym required by the Xmodem code.

```
include %AppDir%\MinXmodemRx            \ Xmodem receiver
include %AppDir%\FlashFTFL              \ Flash primitives
include %AppDir%\ReprogApp              \ Application


make-turnkey ReprogFlash                \ start up action
```
Define the action of the reprogramming code.

```
' CopyFlash CLD_CopyFlash !
```
Define the action of the Flash copy code.

26 Freescale Kinetis Reflashing

26.1 Introduction

If you destroy the application in the internal Flash, you must use third party tools to reflash the unit.

Once the Forth system is running again, you can use the word `REFLASH ( -- )` to download a new binary image. Note that `REFLASH` only handles binary memory image files. These should be transferred using the XModem 128 (checksum) protocol. If you are using AIDE with the file server enabled, a file selection dialog will appear automatically after you have executed `REFLASH`.

The Kinetis parts can only be reflashed from an application by a program running from RAM. A separate application built by a control file *Reprog\Reprog*.CTL* is used to do this. A binary image `REPROG*.IMG` is inserted into the application. When required, the code is copied into the internal RAM and executed from RAM.

Return is by rebooting the system. The contents of the internal RAM and Flash are destroyed, therefore any data that must be preserved should be saved externally to the chip. Alternatively, modify the reprogramming code to use the last sector to hold data to be preserved.

26.2 Code in main application

The layout of the RAM code is defined in the control file for the RAM application. This defines the first cell as the Forth word to execute. The RAM code is included in the Flash image and copied to RAM when needed.

```
create reprog.img       \ -- addr
```
The start address of the reprogramming code

```
  data-file %HwDir%/ReProg/REPROGK60.img equ /reprog    \ -- len
```
Generic K60: The length of the reprogramming code loaded into the dictionary.

```
$1FFF:F000 equ reprogrun        \ -- addr
```
The run time address of the reprogramming code. This **must** match the start address of the `CDATA` section defined in *ReprogK60.ctl*, and that section **must not** overlap the run-time stacks and user area.

```
_UART3 value ReflashDev \ -- addr
```
Holds THE UART base address used for XModem transfer. The default is UART3.
The start of the RAM code block holds data about the running Forth.

- Cell 0 - xt of `ReprogFlash` to run,
- Cell 1 - xt of `CopyFlash` to run,
- Cell 2 - 0 for default UART or base address of another UART,
- Cell 3 - clock speed in Hz of the calling Forth,
- Cell 4 - Length of Flash to reprogram. Often the last sector is used to hold configuration information.
- Cell 5 - Size of a Flash sector.

```
: callit        \ xt --
```
Load the reprogramming code and execute the xt.

`: reflash \ -- ; no exit`
Copies the reprogramming code to the run-time address and executes it.

`: CopyFlash \ src dest len --`
Copy the flash. The parameters are as for `CMOVE`. The process is repeated until there are no errors, and the system is then rebooted.

27 ROM PowerForth utilities

27.1 Introduction

Supplied as source in the ROMFORTH directory are utilities to:

- compile source code on your target board from the cross-compiler IDE
- upload a binary image from your target to your PC
- download a binary image to your target from your PC.

Note that the target source code supplied with cross compiler versions 6.02 onwards is incompatible with code supplied for previous versions of the cross compiler.

These utilities can be used to generate an image that has all the tools required to develop an application, or can be used during development to transfer modules to and from your PC. All the code is designed to be used with the MPE development environment, AIDE. The code will also work with other compatible terminal emulators.

Users who wish to distribute images containing the ROM PowerForth utilities should contact MPE for details of the OEM licence, which includes documentation on disc of the Forth kernel and the ROM PowerForth utilities.

27.2 Compiling text files

Source text files can be compiled from the host PC onto the target system. This saves time in not having to cross-compile the entire source if a small modification is made. The utilities permit text file to be split into pages for better layout when printed. An ASCII Form Feed character (decimal 12) separates one page from another.

27.2.1 The required files

To compile text files from your target board, cross-compile the files *IODEF.FTH* and *TEXTFILE.FTH*.

27.2.2 Compiling a specified text file

To compile all or part of a specified text file onto your target, use `INCLUDE` in the form:

```
INCLUDE <filename>
```

This compiles the file <filename> into the target's dictionary. AIDE's internal file server must be enabled (in the console window configuration), and will be triggered automatically.

27.3 Downloading a binary image

A binary image can be downloaded from the target to your host PC. Two utilities are provided:

- Intel hex download
- XMODEM download

For both utilities the cross-compiler IDE or a suitable communications package will be required.

27.3.1 XMODEM binary image download

Binary images can be downloaded to your PC using the XMODEM protocol.

Required files

To use this utility you must cross-compile the file *COMMON\XMODEMTXRX.FTH.*

Using the XMODEM binary download utility

To download a binary image from the target system to your PC, use `BIN-DOWN` in the form:

```
addr #bytes BIN-DOWN
```

where *addr] is the start address and *\i{#bytes* is the number of bytes to down-load starting from addr. For example,

```
1200 400 BIN-DOWN
```

sends the area of memory from 1200 to 1599 to your host PC. AIDE's internal file server must be enabled (in the console window configuration), and will be triggered.

27.3.2 Intel hex download

Binary images can be downloaded to your PC using the Intel hex format. Because this format only uses the standard printable character and CR/LF, the data can be captured by very simple tools.

Required files

To use this utility you must cross-compile the file *INTELHEX.FTH.*

Using the Hex download utility

To download a binary image from the target system to your PC, use `HEX-DOWN` in the form:

```
addr #bytes HEX-DOWN
```

where *addr* is the start address and *#bytes* is the number of bytes to down-load starting from addr. For example,

```
1200 400 HEX-DOWN
```

sends the area of memory from 1200 to 1599 to your host PC. In AIDE, turn on console logging to receive the file. In other packages this may be referred to as file capture.

27.4 ROM PowerForth

ROM PowerForth can be used to generate a stand-alone Forth system. With these utilities, you can generate an image that contains an interactive Forth with the ability to develop an application. Note: A licence is required to distribute open Forth systems. Contact MPE for more details.

27.4.1 Hardware requirements

To develop an application using ROM PowerForth, your board requires an area which:

- is always Flash
- is always RAM
- is RAM for development and Flash for application

27.4.2 Flash area

The area that is always Flash contains the development kernel.

27.4.3 RAM area

The area that is always RAM is used for variables and all changeable data.

27.4.4 RAM/Flash area

This area is used to develop your application. Therefore, it must be RAM while developing. Once your application is developed, the application's image must be saved into battery-backed RAM or Flash. Therefore, this area must have the ability to be alterable but also be non-volatile.

27.4.5 Types of board

The type of board that can be used to develop using ROM PowerForth is restricted to:

- three site boards
- two site boards with battery backed RAM
- two site boards with socket converter.

Three site boards

The three areas are provided by three memory sockets:

- Flash holding development kernel
- RAM which holds the variables and changeable data
- Flash or RAM which is selectable by a link on the board

Two site boards with battery backed RAM

The three areas are provided by two sockets:

- Flash holding the development kernel
- battery-backed RAM which is split into two areas.

Two site boards with socket converter

On many boards, there is unused space in the Flash as ROM PowerForth occupies less than

32k bytes of memory. A RAM image can be saved to Flash or external serial EEPROM and then restored at power up. Such code is device-specific and is *not* part of the generic ROM PowerForth.

27.4.6 Making your application turnkey

Once your application has been developed, it needs to be made turnkey so that it is always available. The application can be made semi-permanent by compiling into battery-backed RAM in the RAM/Flash area. Alternatively, it can be copied into Flash if the board allows.

Configuring a turnkey application

The word `SETUP` takes the address of the word passed to it and marks this in the RAM/Flash header as the address of the word to be run at power-up. If a value of zero is passed to *SETUP*, the interactive Forth kernel will be run at power up.

For example, the word `JOB` is to be run at power-up. Therefore you type,

```
' JOB SETUP
```

27.4.7 Discarding the application RAM area

The application can be discarded by typing:

```
0 ROM !
```

27.4.8 Changing the application RAM start address

The constant ROM returns the start address of the application RAM area. If the address of this area is to be changed, the Flash must be modified. To do this, the cell value in ROM must be changed.

27.4.9 AIDE file server protocols

AIDE's file server must be enabled for automatic file handling. Details of the protocols used should be obtained from this manual and the source code in the *COMMON\ROMFORTH* directory.

27.5 IODEF.FTH

*\i{Common/ROMFORTH/IODEF.FTH] provides equates and protocol primitives for AIDE.

Before compiling this file, synonyms may need to be defined for `SER-EMIT`, `SER-KEY?` and `SER-KEY`. Add these in the control file before compiling *IODEF.FTH*.

27.5.1 AIDE support

```
variable disk-error     \ -- addr ; set non-zero on error
```
This variable is set true when a transfer error occurs.

```
: wait-ack      \ -- ; wait for ACK character
```
This word waits for the host to send an ACK at the end of part of a transfer. INTERNAL.

```
: wait-ack/nack \ -- t/f ; true for NACK
```
This waits for either a NACK or an ACK from the host and leaves true or false on stack. INTERNAL.

`: send-block#  \ n -- ; send block number to server`
Sends a single length number as two bytes 00-FF, low byte first. INTERNAL.

`: synch-to-host \ -- ; sync host to us`
Waits for a START (0x01) character, flushes the input, and sends an ACK. INTERNAL. INTERNAL.

27.6 Miscellaneous

`: cls  \ --`
Clear PowerTerm screen

27.7 Application Extensions

ROM PowerForth can be used to generate a stand-alone Forth system. With these utilities, you can generate an EPROM that contains an interactive Forth with the ability to develop an application. Note: A licence is required to distribute open Forth systems. Contact MPE for more details.

The code in *COMMON\ROMFORTH\LINK.FTH* provides the facilities to link the extension area into the ROM PowerForth kernel.

`appl-rom equ ROM  \ start of ROM/RAM area`
Define the address of a Flash/RAM area in which applications are to be developed. This area is RAM during development. When development is complete, the word `SETUP` is executed to initialise the memory area. The compiled image is then downloaded to the PC using the tools in the ROMFORTH directory. The image is then programmed into Flash, which can then replace the RAM. If the word `RELINK` is added to `COLD`, the application Flash will be automatically linked in when the target powers up.

`variable rp  EM rp !  \ top of RAM area`
If your processor has separate CODE and DATA address spaces, it has a HARVARD architecture. This word is only compiled for Harvard targets such as 8051s. The working RAM area has the interactive RAM dictionary at the bottom and the application buffer space at the top. Define the initial value of RP to be the highest available free location in the RAM area. Application variable and buffer space will then be allocated downwards from this address. When creating an application to be ROMmed, download the code for `BUFFER` and the redefined `VARIABLE` first of all.

`: setup  \ xt|0 -- ; sets up for ROM generation`
Use XT as the xt of the word to be run when the extension is found.

`: use-setup-data  \ --`
Apply the data in the setup area.

`: link-rom  \ --`
Link in the application ROM area.

`: link-ram  \ -- ; link in application RAM area`
Link in the application RAM area.

`: link-nv-ram  \ -- ; link in for NV-RAMs`
Link in the application NV-RAM area.

`: relink  \ -- ; re-link application if there`
Link in the application ROM/RAM/NV-RAM area.

27.8 INCLUDE source code from AIDE

The file *COMMON\ROMFORTH\TEXTFILE.FTH* provides support for compiling a source file from the AIDE server. The code has been updated for AIDE version 2.500 onwards.

```
: end-load      \ -- ; switch back to keyboard input
```

This word is automatically performed at the end of a download to tidy up the comms.

```
: file-error    \ n --
```

Handle an error when a file is being INCLUDEd.

```
: $include      \ $addr -- ; compile host file, counted string
```

Given a counted string representing a file name, compile the file from AIDE.

```
: include       \ "<filename>" -- ; load file from host
```

Compile a file across the serial line from the AIDE file server. Use in the form:

```
include <filename>
```

The filename extension must be supplied.

27.9 Simple source file loader

The code in *COMMON\ROMFORTH\FILETRAN.FTH* provides a simple source file loader which can be used with most terminal emulators. The download is controlled by XON/XOFF flow control. When using the PowerTerm terminal emulator in AIDE, use the `INCLUDE <filename>` system which supports nested files and needs no special termination.

Each file compiled must include a single line

```
END-UP-LOAD
```

at the end to reset the interpreter.

For slow 8 bit CPUs without queued serial input, the terminal server may need to include pacing delays after each character and an additional after CR/LF pairs.

```
: Up-Load       \ -- ; Load ASCII text
```

Compile a file delivered by the terminal emulator. This word is intolerant of compilation errors.

```
: End-Up-Load   \ -- ; Finish Up-Loading
```

Used on the target to restore the Forth interpreter after a file has been compiled.

27.10 Intel Hex transfers

The code in *COMMON\ROMFORTH\INTELHEX.FTH* provides a simple way to transfer binary images from ROM PowerForth to a host. The Intel Hex format is printable and can be captured (logged) by most terminal emulators including AIDE.

```
: HEX-DOWN      \ addr #bytes --
```

Send *#bytes* at *addr* from ROM PowerForth to the host in Intel Hex format.

27.11 Block support

BLOCKS.FTH provides support for the transfer of 1k blocks between a host PC and the embedded target. Although the use of blocks as source screens is now obsolete, blocks are still useful for the transfer of binary data, especially for data loggers which use paged RAM or Flash for data storage. AIDE version 3 or above is required.

Despite the comment about the obsolescence of screens, the code in *BLOCKS.FTH* provides screen file support for legacy systems.

27.11.1 Primitives

```
: BLK-READ      \ addr blk# --
```
Reads one block from host to addr.

```
: BLK-WRITE     \ addr blk# --
```
Sends given block number to host.

27.11.2 Application words

```
$400 equ /block                 \ -- size ; size of a block
```
The size of a block.

```
/first buffer: FIRST            \ -- addr ; first element is block no.
```
The data buffer. The first cell contains the block number and is followed by `/BLOCK` data bytes.

```
: SAVE-BUFFERS  \ --
```
Save the data buffer if it has been modified.

```
: EMPTY-BUFFERS \ --
```
Mark the data buffer as empty.

```
: UPDATE        \ --
```
Mark the data buffer as modified.

```
: FLUSH         \ --
```
Force a changed buffer to be uploaded to the host.

```
: BLOCK         \ u -- addr
```
Addr is the address of the first character of the block buffer assigned to mass-storage block u. An ambiguous condition exists if u is not an available block number. If block u is already in a block buffer, a-addr is the address of that block buffer. If block u is not already in memory and there is an unassigned block buffer, transfer block u from mass storage to an unassigned block buffer. Addr is the address of that block buffer. If block u is not already in memory and there are no unassigned block buffers, unassign a block buffer. If the block in that buffer has been UPDATEd, transfer the block to mass storage and transfer block u from mass storage into that buffer. Addr is the address of that block buffer. At the conclusion of the operation, the block buffer pointed to by addr is the current block buffer and is assigned to u.

27.11.3 Block file management

```
: -trailing     \ addr len -- addr len' ; strip trailing spaces
```
This word strips the trailing spaces from a string. It is most useful when displaying lines from a screen to strip the trailing spaces from the 64 byte fixed length lines.

```
: .LINE         \ line# blk# --
```
Display line# from blk#.

```
: ?LOADING      \ --
```

THROW #-501 if input from console.

```
: LIST          \ blk# -- ; display screen given
```

Display screen blk# as 16 lines of 64 characters.

```
: L             \ --
```

LIST the current screen in SCR.

```
: N             \ --
```

LIST the Next screen.

```
: P             \ --
```

LIST the Previous screen.

```
: Index         \ n1 n2 --
```

Display the top lines in the (inclusive) range of screens.

```
: Qx            \ --
```

INDEX all available screens.

```
: LOAD          \ blk# -- ; compile given screen
```

Save the current input-source specification. Store u in BLK (thus making block u the input source and setting the input buffer to encompass its contents), set >IN to zero, and interpret. When the parse area is exhausted, restore the prior input source specification. Other stack effects are due to the words LOADed.

```
: THRU          \ first last --
```

LOAD screens from first to last in that order.

```
: -->           \ --
```

LOAD the next screen.

```
: $using        \ $addr --
```

Select the file specified by the counted string at $addr as the current block/screen file.

```
: using         \ "<file>" --
```

Use in the form "USING <filename>" to select the current block/screen file. The AIDE file server is required. AIDE takes care of creating non-existent files.

27.12 Target BUFFER: and VARIABLE

The definitions below reorganise the use of RAM for ROM PowerForth applications which are themselves to be put in Flash.

The variable RP points to the top of available RAM, and is decremented by the amount of RAM required. The word `BUFFER:` allocates memory from this space, returning its base (low) address. Words such as `VARIABLE` can then be defined in terms of `BUFFER:`.

```
: Buffer:       \ n -- ; -- addr
```

Return the address of an n byte buffer. Use in the form:

```
  <size> BUFFER: <name>
```

```
: variable      \ -- ; -- addr ; replacement
```

Create a new variable. Use in the form:

```
  VARIABLE <name>
```

27.13 Some simple tools

The file *COMMON\ROMFORTH\TOOLS.FTH* provides some simple application tools.

The words `(#IN)` and `#IN` are provided to collect numbers from the keyboard in decimal. `(#IN)` is useful because it returns the number of digits converted so that 0 digits indicates that the user just pressed <Enter>, rather than entering a valid number of value 0.

```
: (#in)          \ -- n #chars
```
Read a decimal number from the keyboard returning the number and the number of digits converted.

```
: #in            \ -- n
```
Collect a decimal number from the keyboard.

28 Minimal Umbilical code definitions

The file *Cortex/MinCortex.fth* contains the minimum code definitions required to support Umbilical Forth. If additional words are required, they may be copied to a new file from *Cortex/CodeCortex.fth* or *Common/Kernel62.fth*.

28.1 Register usage

For Cortex-M3+ the following register usage is the default:

```
r15         pc      program counter
r14         link    link register; bit0=1=Thumb, usually set
r13         rsp     return stack pointer
r12         psp     data stack pointer
r11         up      user area pointer
r10         --
r9          lp      locals pointer
r8          --
r7          tos     cached top of stack
r0-r6       scratch
```

The VFX optimiser reserves R0 and R1 for internal operations. `CODE` definitions must use R10 as TOS with NOS pointed to by R12 as a full descending stack in ARM terminology. R0..R8 are free for use by `CODE` definitions and need not be preserved or restored. You should assume that any register can be affected by other words.

28.2 Configuration

These equates are set false (zero) if they have not already been defined.

```
false equ DSQRT?        \ -- flag
```

Set this non-zero to compile `DSQRT`.

```
false equ FastCmove?    \ -- flag
```

Set this flag true to use a fast but vast (~1kb) version of `CMOVE`. If your application uses either `CMOVE` or `MOVE` in time-critical code, the fast version offers an overall speed up of about four times. The code for the fast but vast version was written by Rowley Associates, whose permission to adapt and publish the code with the MPE cross compiler is much appreciated.

28.3 Flow of control

```
CODE (DO)       \ limit index --
```

The run time action of `DO` compiled on the target. INTERNAL.

```
CODE (?DO)      \ limit index --
```

The run time action of `?DO` compiled on the target. INTERNAL.

```
CODE EXECUTE    \ xt --
```

Execute the code described by the XT. This is a Forth equivalent to an assembler JSR/CALL instruction.

28.4 Stack operations and maths

`CODE NOOP \ --`
A NOOP, null instruction.)

`: DROP \ x --`
Lose the top data stack item and promote NOS to TOS.

`CODE WITHIN? \ n1 n2 n3 -- flag`
Return TRUE if N1 is within the range N2..N3. This word uses signed arithmetic.

`CODE WITHIN \ n1|u1 n2|u2 n3|u3 -- flag`
The ANS version of WITHIN?. This word uses unsigned arithmetic, so that signed compares are treated as existing on a number circle.

28.5 Multiplication

`: UM* \ u1 u2 -- ud`
Perform unsigned-multiply between two numbers and return double result.

`: * \ n1 n2 -- n3`
Standard signed multiply. N3 = n1 * n2.

`: m* \ n1 n2 -- d`
Signed multiply yielding double result.

28.6 Division

ARM Cortex provides 32/32 division instructions, but no 64/32 ones. Avoid 64/32 division routines if you can where performance matters.

`macro: udiv64_step \ --`
Cross compiler macro to perform one step of the unsigned 64 bit by 32 bit division

`code um/mod \ ud1 u2 -- urem uquot`
Slow and short - Full 64 by 32 unsigned division subroutine. This routine uses a loop for code size. This version is commented out by default.

`code um/mod \ ud1 u2 -- urem uquot`
Fast and big - Full 64 by 32 unsigned division subroutine. Unrolled for speed. This routine uses 660 bytes of code space using the Thumb-2 instruction set, whereas the ARM32 version uses 920 bytes.

`macro: udiv63_step \ --`
Cross compiler macro to perform one step of the unsigned 63 bit by 31 bit division

`proc Udiv63/31 \ r0:r1/tos ; 63/31 unsigned divide -> tos=quot, r0=rem`
Unsigned division primitive - unrolled for speed. Note that this routine does not handle the top bit of the divisor and dividend correctly. `Udiv63/31` is used for signed divide operations for which the top bits are always zero.

`CODE FM/MOD \ d1 n2 -- rem quot ; floored division`
Perform a signed division of double number D1 by single number N2 and return remainder and quotient using floored division. See the ANS Forth specification for more details of floored division.

`CODE SM/REM \ d1 n2 -- rem quot ; symmetric division`
Perform a signed division of double number D1 by single number N2 and return remainder and quotient using symmetric (normal) division.

`CODE /MOD \ n1 n2 -- rem quot`
Signed symmetric division of N1 by N2 single-precision returning remainder and quotient.

`: / \ n1 n2 -- n3`
Standard signed division operator. n3 = n1/n2.

`: MOD \ n1 n2 -- n3`
Return remainder of division of N1 by N2. n3 = n1 mod n2.

`: */MOD \ n1 n2 n3 -- n4 n4`
Multiply n1 by n2 to give a double precision result, and then divide it by n3 returning the remainder and quotient. The point of this operation is to avoid loss of precision.

`: */ \ n1 n2 n3 -- n4`
Multiply n1 by n2 to give a double precision result, and then divide it by n3 returning the quotient. The point of this operation is to avoid loss of precision.

`: M/ \ d n1 -- n2`
Signed divide of a double by a single integer.

28.7 Miscellaneous math

`CODE D+ \ d1 d2 -- d3`
Add two double precision integers.

`CODE D- \ d1 d2 -- d3`
Subtract two double precision integers. D3=D1-D2.

`CODE DNEGATE \ d1 -- -d1`
Negate a double number.

`CODE ?NEGATE \ n1 flag -- n1|n2`
If flag is negative, then negate n1.

`CODE ?DNEGATE \ d1 flag -- d1|d2`
If flag is negative, then negate d1.

`CODE ABS \ n -- u`
If n is negative, return its positive equivalent (absolute value).

`CODE DABS \ d -- ud`
If d is negative, return its positive equivalent (absolute value).

`CODE ROLL \ xu xu-1 .. x0 u -- xu-1 .. x0 xu`
Rotate the order of the top N stack items by one place such that the current top of stack becomes the second item and the Nth item becomes TOS. See also `ROT`.

28.8 Strings

`code cmove \ asrc adest len --`
Copy *len* bytes of memory forwards from *asrc* to *adest*. If the performance of `CMOVE` is important in your application, set the equate `FastCmove?` non-zero and a much faster (four to five times) but much larger (~900 bytes) version will be compiled. See *Cortex\fcmove.fth* for the details.

`CODE CMOVE> \ c-addr1 c-addr2 u --`
As `CMOVE` but working in the opposite direction, copying the last character in the string first.

`CODE FILL \ c-addr u char --`
Fill LEN bytes of memory starting at ADDR with the byte information specified as CHAR.

`: erase \ c-addr u -- ; wipe memory`
Set U bytes of memory starting at C-ADDR with zeros.

`CODE S= \ c-addr1 c-addr2 u -- flag`
Compare two same-length strings/memory blocks, returning TRUE if they are identical.

`CODE (") \ -- a-addr ; return address of string, skip over it`
Return the address of a counted string that is inline after the CALLING word, and adjust the CALLING word's return address to step over the inline string. The adjusted return address will be at a four byte boundary. See the definition of `(.")` for an example.

`: (C") \ -- c-addr`
The run time action compiled by C".

`: (S") \ -- c-addr u`
The run time action compiled by S".

28.9 Umbilical versions of defining words

`here is-action-of constant`
The runtime code for a CONSTANT.

`here is-action-of variable`
The runtime action for a VARIABLE.

`here is-action-of value`
The runtime action of a VALUE.

`here is-action-of user`
The runtime action of a USER variable.

`: u# \ "<name>"-- u`
An `INTERPRETER` word that returns the index of the `USER` variable whose name follows, e.g.

```
u# S0
```

`: CRASH \ -- ; used as action of DEFER`
The default action of a DEFERed word. A NOOP.

`here is-action-of DEFER \ Comp: "<spaces>name" -- ; Run: i*x -- j*x`
The runtime action of a DEFERred word.

28.10 Display words

`: SPACE \ --`
Output a blank space (ASCII 32) character.

`: SPACES \ n --`
Output 'n' spaces, where 'n' > 0. If 'n' < 0, no action is taken.

`: .nibble \ n --`
Convert a nibble to a hex ASCII digit and display it.

`: .BYTE \ b --`
Display the byte *b* as a 2 digit hex number.

`: .WORD \ w --`
Display w as a 4 digit unsigned hexadecimal number.

`: .dword \ x --`
Display x as an 8 digit unsigned hexadecimal number.

```
: .lword        \ x --
```
A synonym for `.DWORD` above.

29 ARM Cortex specific library code

The code in *Cortex/LibCortex.fth* is conditionally compiled by the following code fragment to be found at the end of many control files.

```
libraries       \ to resolve common forward references
  include %CpuDir%/LibCortex
  include %CommonDir%/library
end-libs
```

Each definition in a library file is surrounded by a phrase of the form:

`[required] <name> [if] : <name> ... ;  [then]`

The phrase `[REQUIRED] <name>` returns true if `<name>` has been forward referenced and is still unresolved. The code between `LIBRARIES` and `END-LIBS` is repeatedly processed until no further references are resolved.

29.1 I/O initialisation

`: init-io       \ addr --`
Copy the contents of the I/O set up table to an I/O device. Each element of the table is of the form addr (cell) followed by data (cell). The table is terminated by an address of 0. A table of a single 0 address performs no action.

29.2 interrupt enable and disable

`code di         \ --`
Disable interrupts.

`code ei         \ --`
Enable interrupts.

`code dfi        \ --`
Disable fault exceptions.

`code efi                \ --`
Enable fault exceptions.

`code [I         \ R: -- x1 x2`
Preserve interrupt/exception status on the return stack, and disable interrupts/exceptions except reset, NMI and HardFault. The state is restored by `I]`.

`code I]         \ R: x1 x2 --`
Restore interrupt status saved by `[I` from the return stack.

29.3 Miscellaneous

`: @OFF          \ addr -- x`
Read cell at addr, and set it to 0.

`: @on           \ addr -- val`
Fetch contents of cell at addr and set it to -1.

Index

!

"

#

$

%

’

(

*

+

,

-

.

/

:

;

<

=

>

?

@

[

]

\

~

0

1

2

4

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z