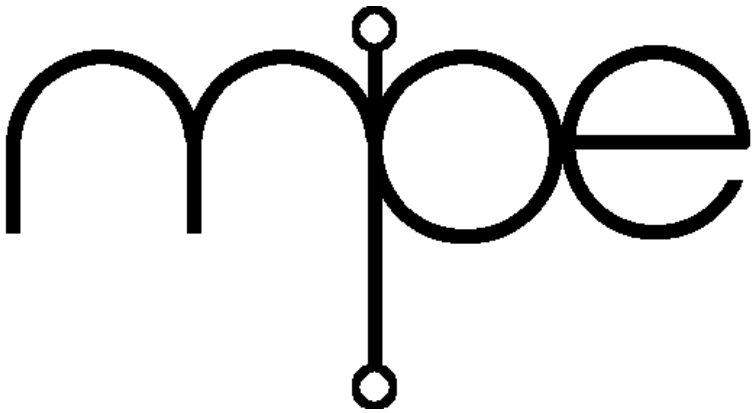
ARM Cortex-M Target Code

v7.5

Microprocessor Engineering Limited



Published by Microprocessor Engineering

ARM Cortex-M Target Code v7.5
User manual
Manual revision 7.5
4 May 2016

Software
Software version 7.5

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 How Forth is documented

The Forth words in this manual are documented using a methodology based on that used for the ANS standard document. As this is not a standards document but a user manual, we have taken some liberties to make the text easier to read. We are not always as strict with our own in-house rules as we should be. If you find an error, have a complaint about the documentation or suggestions for improvement, please send us an email or contact us in some other way.

When you browse the words in the Forth dictinary using `WORDS` or when reading source code you may come across some words which are not documented. These words are undocumented because they are words which are only used in passing as part of other words (factors), or because these words may change or may not exist in later versions.

"*Documentation is like sex: when it is good, it is very, very good; and when it is bad, it is better than nothing.*" - *Dick Brandon*

1.1 Forth words

Word names in the text are capitalised or shown in a bold fixed-point font, e.g. SWAP or `SWAP`. Forth program examples are shown in a Courier font thus:

```
: NEW-WORD   \ a b -- a b
  OVER DROP
;
```

If you see a word of the form `<name>` it usually means that `name` is a placeholder for a name you will provide.

The notation for the glossary entries in this manual have two major parts:

- The definition line.
- The description.

The definition line varies depending on the definition type. For instance - a normal Forth word will look like:

```
: and            \ n1 n2 -- n3                      6.1.0720
```

The left most column describes the word `NAME` and type (colon) the center column describes the stack effect of the word and the far right column (if it exists) will specify either the ANS language reference number or an MPE reference to distinguish between ANS standard and MPE extension words.

The stack effect may be followed by an informal comment separated from the stack effect by a ';' character.

```
: and            \ x1 x2 -- x3 ; bitwise and
```

This is a "quick reference" comment.

When you read MPE source code, you will see that most words are written in the style:

```
: foo       \ n1 n2 -- n3
\ *G This is the first glossary description line.
\ ** These are following glossary description lines.
  ...
;
```

Most MPE manuals are now written using the DocGen literate programming tool available and documented with all VFX Forths for Windows, Linux and DOS. DocGen extracts documentation lines (ones that start "\ *X ") from the source code and produces HTML or PDF manuals.

1.2 Stack notation

```
before -- after
```

where *before* means the stack parameters before execution and *after* means stack parameters after execution. In this notation, the top of the stack is to the right. Words may also be shown in context when appropriate. Unless otherwise noted, all stack notations describe the action of the word at execution time. If it applies at compile time, the stack action is preceded by C: or followed by (compiling)

An action on the return stack whill be shown

```
R: before -- after
```

Similarly, actions on the separate float stack are marked by F: and on an exception stack by E:. The definition of >R would have the stack notation

```
x -- ; R: -- x
```

Defining words such as VARIABLE usually indicate the stack action of the defining word (VARIABLE) itself and the stack action of the child word. This is indicated by two stack actions separated by a ';' character, where the second action is that of the child word.

```
: VARIABLE    \ -- ; -- addr
```

In cases where confusion may occur, you may also see the following notation:

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

Unless otherwise stated all references to numbers apply to native signed integers. These will be 32 bits on 32 bit CPUs and 16 bits on embedded Forths for 8 and 16 bit CPUs. The implied range of values is shown as {from..to}. Braces show the content of an address, particularly for the contents of variables, e.g., BASE {2..72}.

The native size of an item on the Forth stack is referred to as a CELL. This is a 32 bit item on a 32 bit Forth, and on a byte-addressed CPU (the vast majority, most DSP chips excluded) this is a four-byte item. On many CPUs, these must be stored in memory on a four-byte address

boundary for hardware or performance reasons. On 16 bit systems this is a two-byte item, and may also be aligned.

The following are the stack parameter abbreviations and types of numbers used in the documentation for 32 bit systems. On 16 bit systems the generic types will have a 16 bit range. These abbreviations may be suffixed with a digit to differentiate multiple parameters of the same type.

```
Stack         Number     Range              Field
Abbreviation  Type       (Decimal)          (Bits)
flag          boolean    0=false, nz=true   32
true          boolean    -1 (as a result)   32
false         boolean    0                  32
char          character  {0..255}           8
b             byte       {0..255}           8
w             word       {0..65535}         16
  here word means a 16 bit item, not a Forth word
n             number     {-2,147,483,648    32
                         ..2,147,483,647
x             32 bits    N/A                32
+n            +ve int    {0..2,147,483,647} 32
u             unsigned   {0..4,294,967,295} 32
addr          address    {0..4,294,967,295} 32
a-addr        address    {0..4,294,967,295} 32
  the address is aligned to a CELL boundary
c-addr        address    {0..4,294,967,295} 32
  the address is aligned to a character boundary
32b           32 bits    not applicable     32
d             signed     {-9.2e18..9.2e18}  64
              double
+d            positive   {0..9.2e18}        64
              double
ud            unsigned   {0..1.8e19}        64
              double
sys    0, 1, or more system dependent entries
char          character  {0..255}          8
"text"  text read from the input stream
```

Any other symbol refers to an arbitrary signed 32-bit integer unless otherwise noted. Because of the use of two's complement arithmetic, the signed 32-bit number (n) -1 has the same bit representation as the unsigned number (u) 4,294,967,295. Both of these numbers are within the set of unspecified weighted numbers. On many occasions where the context is obvious, informal names are used to make the documentation easier to understand.

1.3 Input text

Some Forth words read text from the input stream (e.g the keyboard or a file). That text is read from the input stream is indicated by the identifiers "<*name*>" or "*text*". This notation refers to text from the input stream, not to values on the data stack.

Likewise, *ccc* indicates a sequence of arbitrary characters accepted from the input stream until

the first occurrence of the specified delimiter character. The delimiter is accepted from the input stream, but it is not one of the characters *ccc* and is therefore not otherwise processed. This notation refers to text from the input stream, not to values on the data stack.

Unless noted otherwise, the number of characters accepted may be from 0 to 255.

1.4 Other markers

The following markers may appear after a word's stack comment. These markers indicate certain features and peculiarities of the word.

`C` The word may only be used during compilation of a colon definition.

`I` The word is immediate. It will be executed even during compilation, unless special action is taken, e.g. by preceding it word with the word `POSTPONE`.

`M` Affected by multi-tasking

`U` A user variable.

2 Cortex start up for LPC17xx

The file *Cortex/Hardware/LPC17xx/startLPC17xx.fth* contains the start up code for LPC17xx devices. Start up code for other Cortex-M3 implementations, e.g. STM32, can be found in parallel directories. The absolute minimum code to start a Cortex-M3 is in *Cortex/StartCortex.fth.*

`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 Cortex-M3 Technical Reference Manual (ARM DDI0337, rev E page 8-21). 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. **Do not** use this to set the ISP checksum!

`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. This 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.
Find the clock divider that generates the CCO clock at 288 Mhz. The value in the CCLKCFG register is one less than this.

`#288000000 system-speed / 1- equ CCLKCFGval \ -- x`
The CCLKCFG register initial value.

`CCLKCFGval 1+ system-speed * equ cco-speed \ -- hz`
The actual CCO frequency that will be generated by the calculated configuration values.

`: genPLL0 \ -- mval nval`
INTERPRETER: Generate the data for the PLL0CFG register. For the moment, we set N=1 (nval=0), and calculate the Mval only.

`: gen-FlTim \ -- flcfg`
Generate the Flash configuration register value.

`L: ^PCLKSEL0val`
This location contains the initial value of the PCLKSEL0 register if the equate `PCLKSELbug?` is set non-zero.

`L: ^PCLKSEL1val`
This location contains the initial value of the PCLKSEL0 register if the equate `PCLKSELbug?` is set non-zero.

`: setClocks \ --`
Enable the clocks, set the bus dividers, and enable the PLLs as required.

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

`: 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.

`code (z") \ -- zaddr`
Return the address of the following counted and zero-terminated string. Step over the string and the trailing zero.

`: 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 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.

4.1 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.

5 Exception Fault handling

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

5.1 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.

5.2 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.

5.3 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.

5.3.1 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.

5.3.2 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.

6 ARM Cortex multitasker

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

The code in *MultiCortex.fth* only works with the Cortex-M3 register set. When using the a Cortex-M0 device with its register allocation, use *MultiCM0.fth*. If you need the interworking (ARM32) register set, please contact MPE or modify the code yourself.

6.1 Configuration

The configuration equates can be defined before this file is compiled.

`1 equ lazy-save?  \ -- flag`
If previously undefined, `LAZY-SAVE?` is set to one and registers are only saved if they have to be.

`0 equ norun-sleep?  \ -- flag`
If previously undefined, `NORUN-SLEEP?` is set to zero. If set to one, the processor is put to sleep using a WFI instruction if there are no other tasks to run. This can save a lot of power, but requires regular interrupts and care that tasks are properly enabled. For this equate to be used `LAZY-SAVE?` must be non-zero.

`0 equ test-multi?  \ -- flag ; true to compile test code`
If previously undefined, `TEST-MULTI?` is set to zero and test code is not compiled.

6.2 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.

6.3 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 - with lazy register save.

`CODE pause \ -- ; the scheduler itself`
The software scheduler itself - without lazy register save.

`: 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`
`HALT`s the current task, and executes `PAUSE`.

6.4 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.

6.5 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.

6.6 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 `HALT`ed. Use in the form:

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

to put a task on the active task list, but as if `HALT`ed. `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.

6.7 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.

6.8 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:.

6.9 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.

7 VFP Floating Point - single precision

7.1 Introduction

The Forth data stack and the floating point stack are separate. As with the data and return stacks, the floating point stack grows down. The floating point data storage format is IEEE 32 bit (single precision) format. The source code is in the file *Cortex/VFP32SX.fth*. It should also work with minimal change on ARM32 CPUs but has not been tested for these.

Note that single-precision floating point only has 23 bits in the mantissa and 8 bits in the exponent; and thus has a severely restricted dynamic range.

This code and the compiler with IEEE F.P. support are supplied with the Professional edition of the compiler, not with the Lite, Stamp or Standard editions.

7.2 Compiling the code

The source code for the ARM/Cortex version of the code is in the file *Cortex/VFP32SX.fth*. In order to provide full initialisation in other files, the following equates must be set in the control file before any code is compiled:

```
0 equ softfp?     \ true for software floating point
1 equ hardfp?     \ true for hardware floating point
1 equ fpstack?    \ true for a separate floating point stack,
                  \ in which case FP-SIZE must be non-zero
$0100 equ FP-SIZE \ size of FP stack in bytes. Must not be 0.
```

The separate floating point stack is used when `fpstack?` is true and `FP-SIZE` is defined and is non-zero. Add the line below to compile the code.

```
include %CpuDir%/VFP32SX.fth
```

The code must be compiled with *xArmCortexHfpDev*, which has support for compiling IEEE F.P. numbers.

7.3 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). Inside a colon definition, a floating point literal number must be preceded by `F#`.

```
: foo  ... f# 1.234e0 ...  ;
```

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.

7.4 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 `!`.

7.5 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:

```
F# 1.234e0 FCONSTANT FCON1
```

When `FCON1` is executed, it returns 1.234 on the float stack.

7.6 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.

7.6.1 Calculating sines, cosines and tangents

To calculate sine, cosine and tangent, use `FSIN`, `FCOS` and `FTAN` respectively. Angles are expressed in radians.

7.6.2 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.

7.6.3 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.

7.6.4 Calculating powers

Three power functions are supplied:

```
FEXP F10^X X^Y
```

7.7 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.

7.8 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.

7.9 Number formats, ANS and Forth-2012

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'. In order to distinguish between double and floating point numbers, a floating point number must contain 'e' and an exponent.

7.10 FP Stack primitives

```
4 equ FPCELL    \ -- u
```
Size of a floating point number in bytes.

```
FPCELL constant FPCELL  \ -- u
```
Size of a floating point number in bytes.

```
FPCELL setFloatSize     \ --
```
Tell the cross compiler the size in memory of a floating point number.

```
: finit         \ F: i*f -- ; resets FPU and FP stack
```
Reset the floating point stack. Do not forget to use this in a task before using floating point.

```
: fdepth        \ -- #f
```
Floating point equivalent of `DEPTH`. The result is returned on the Forth data stack.

```
code CLZ        \ x -- u
```
Return the number of leading zeros in x.

```
: DCLZ          \ dx -- u
```
Return the number of leading zeros in the double dx.

```
code >fs        \ f -- ; F: -- f
```
Copy a float from the data stack to the floating point stack.

```
code fs>        \ F: f -- ; -- f
```
Copy a float from the float stack to the data stack.

```
code fps@       \ -- fps
```
Read the floating point status/control register.

```
code fps!       \ fps --
```
Set the floating point status/control register.

```
code exp@       \ F: f -- f ; -- exp(2)
```
Copy the exponent of the top float to the data stack. The IEEE exponent offset is removed.

```
code exp!               \ exp(2) -- ; F: f -- f'
```
Change/Set the exponent of the top float. The IEEE exponent offset is added.

```
code F!         \ F: r -- ; addr --
```
Stores *r* at *addr*.

```
code F@         \ addr -- ; F: -- r
```
Fetches *r* from *addr*.

```
synonym SF! F!          \ F: r -- ; addr --
```
Stores *r* at *addr* in IEEE 32 bit format.

```
synonym SF@ F@          \ addr -- ; F: -- r
```
Fetches *r* from *addr*, which contains a float in IEEE 32 bit format..

```
: F,            \ F: r --
```
Lays a real number into the dictionary, reserving FPCELL bytes.

```
Synonym SF, F,          \ F: r --
```
Lays a real number into the dictionary as an IEEE 32 bit number.

```
code FDUP       \ F: r -- r r
```
Floating point equivalent of `DUP`.

```
code FOVER      \ F: r1 r2 -- r1 r2 r1
```
Floating point equivalent of `OVER`.

```
code FSWAP      \ F: r1 r2 -- r2 r1
```
Floating point equivalent of `SWAP`.

```
code FPICK      \ u -- ; F: fu..f0 -- fu..f0 fu
```
Floating point equivalent of `PICK`.

```
code FROT       \ F: r1 r2 r3 -- r2 r3 r1
```
Floating point equivalent of `ROT`.

```
code F-ROT              \ F: r1 r2 r3 -- r3 r1 r2
```
Floating point equivalent of `-ROT`.

```
code FDROP      \ F: r --
```
Floating point equivalent of `DROP`.

```
: FNIP          \ F: r1 r2 -- r2
```
Floating point equivalent of `NIP`.

```
code f>r        \ F: f -- ; R: -- f
```
Put float onto return stack.

```
code fr>        \ R: f -- ; F: -- f
```
Pull float from the return stack.

```
code flit       \ F: -- f ; inline literal
```
Run-time routine for a floating point literal. Cortex version.

```
code flit       \ F: -- f ; inline literal
```
Run-time routine for a floating point literal. ARM32 version.

7.11 Floating point defining words

`: FVARIABLE \ "<spaces>name" -- ; Run: -- addr`

Use in the form: `FVARIABLE <name>` to create a variable that will hold a floating point number.

`: FCONSTANT \ F: 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 -- ; F: -- 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.

`: FBUFF \ u "name" -- ; i -- addr`

Creates a buffer for 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.

```
3 foo
```

Returns the address of element 3 in the buffer.

7.12 Type conversions

`code FSIGN \ F: fn -- |fn| ; -- flag ; true if negative`

Return the absolute value of fn and a flag which is true if fn is negative. F.P. stack operation.

`code S>F \ n -- ; F: -- fn`

Converts a single signed integer to a float.

`code F>S \ F: 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 -- ; F: -- fn`

Converts a double integer to a float.

`: f>d \ f -- ; -- dint(f)`

Converts a float to a double integer. Note that `F>D` truncates the number towards zero according to the ANS specification.

`: FINT \ F: f1 -- f2`

Chop the number towards zero to produce a floating point representation of an integer.

7.13 Arithmetic

```
code FNEGATE    \ F: r1 -- r2
```
Floating point negate.

```
: ?FNEGATE      \ n -- ; F: fn -- fn|-fn
```
If n is negative, negate fn.

```
code FABS       \ F: fn -- |fn|
```
Floating point absolute.

```
code F+         \ F: r1 r2 -- r3
```
Floating point addition.

```
code F-         \ F: r1 r2 -- r3
```
Floating point subtraction, r3=r1-r2

```
code F*         \ F: r1 r2 -- r3
```
Floating point multiply.

```
code F/         \ F: r1 r2 -- r3
```
Floating point divide.

```
code fsqrt      \ F: f1 -- f2
```
F2=sqrt(f1).

```
: FSEPARATE     \ F: 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         \ F: f1 f2 -- f3
```
Leave the fractional remainder from the division f1/f2. The remainder takes the sign of the dividend.

7.14 Relational operators

```
code F0<        \ F: f1 -- ; -- flag
```
Floating point 0<.

```
code F0>        \ F: f1 -- ; -- flag
```
Floating point 0>.

```
code F0=                \ F: f1 -- ; -- flag
```
Floating point 0=.

```
code F0<>       \ F: f1 -- ; -- flag
```
Floating point 0<>.

```
: F=            \ F: f1 f2 -- ; -- flag
```
Floating point =.

```
: F<            \ F: r1 r2 -- ; -- flag
```
Floating point <.

```
: F>            \ F: f1 f2 -- ; -- flag
```
Floating point >.

```
: FMAX          \ F: r1 r2 -- r1|r2
```
Floating point MAX.

```
: FMIN          \ F: r1 r2 -- r1|r2
```
Floating point MIN.

7.15 Miscellaneous

```
: FALIGNED      \ addr -- f-addr
```
Aligns the address to accept a 4-byte float.

```
: FALIGN        \ --
```
Aligns the dictionary to accept a 4-byte float.

```
Synonym SFALIGNED ALIGNED       \ addr -- f-addr
```
Aligns the address to accept a 4-byte float.

```
Synonym SFALIGN ALIGN           \ --
```
Aligns the dictionary to accept a 4-byte float.

```
: FLOAT+        \ f-addr1 -- f-addr2
```
Increments addr by 4, the size of a float.

```
: FLOATS        \ n1 -- n2
```
Returns *n2*, the size of *n1* floats.

```
Synonym SFLOAT+ 4+      \ f-addr1 -- f-addr2
```
Increments *addr* by 4, the size of an S-float.

```
Synonym SFLOATS CELLS   \ n1 -- n2
```
Returns *n2*, the size of *n1* S-floats.

7.16 Powers of ten operations

Floating point IEEE numbers have the following approximate ranges:

- Single precision - 10^-44 to 10^+38
- Double precision - 10^-323 to 10^308

As a result, the input code is different for 32 bit and 64 bit floats.

```
f# 1.0e-32 fconstant %10^-32
```
Floating point 1.0e-32.

```
f# 1.0e-16 fconstant %10^-16
```
Floating point 1.0e-16.

```
f# 0.1e0 fconstant %.1
```
Floating point 0.1.

```
f# 1.0e0 fconstant %1
```
Floating point 1.0.

```
f# 10.0e0 fconstant %10
```
Floating point 10.0.

```
f# 1.0e16 fconstant %10^16
```
Floating point 1.0e16.

```
f# 1.0e32 fconstant %10^32
```
Floating point 1.0e32.

```
16 FARRAY POWERS-OF-10E1
```
An array of 16 powers of ten starting at 10^0 in steps of 1.

```
16 FARRAY POWERS-OF-10E-1
```

An array of 16 powers of ten starting at 10^0 in steps of -1.

`: RAISE_POWER  \ exp(10) -- ; F: f -- f'`
Raise the power in preparation for number formatting.

`: SINK_FRACTION \ exp(10) -- ; F: f -- f'`
Reduce the power in preparation for number formatting.

`: *10^X  \  exp(10) -- ; F: f -- f'`
Generate float' = float *10^dec_exp.

7.17 Floating point input

Note that number conversion takes place in `PAD`.

`: CONVERT-EXP  \ c-addr --`
If the character at c-addr is 'D' convert it to 'E'.

`: CONVERT-FPCHAR  \ c-addr --`
Convert the f.p. char '.' to the double char ',' for conversion.

`: ALL-BLANKS?  \ c-addr len -- flag`
Return true if string is all blanks (spaces).

`: 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.

`: doMNUM  \ c-addr u -- d 2 | 0`
Convert the mantissa string to a double number and 2. If conversion fails, just return 0.

`: doENUM  \ 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.

`: FIXEXP  \ dmant exp(10) -- ; F: -- f`
Convert a double integer mantissa and a single integer exponent into a floating point number.

`: FNUMBER?  \ addr -- 0 | n 1 | d 2 | -1 ; F: -- [f]`
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 text with an 'E' as a floating point indicator, e.g, 1.2345e0. If *\fo{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 -- true|false ; F: -- [f]`
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`.

`: FLITERAL  \ Comp: F: r -- ; Run: F: -- r`
Compiles a float as a literal into the current definition. At execution time, a float is returned. For example, `[ %PI 2e0 F* ] FLITERAL` will compile 2PI as a floating point literal. Note that `FLITERAL` is immediate.

`: (F#)  \ addr -- -1|0 ; F: -- [f]`

The primitive for F# below.

```
: F#            \ 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.

7.18 Floating point output

Note that, because global conversion buffers are used, the floating point output routines are not thread-safe.

```
8 value precision       \ -- u
```
Number of significant digits output.

```
: set-precision         \ u --
```
Set the number of significant digits used for output.

```
: exp(10)       \ F: f -- f ; -- exp[10]
```
Generate the power of ten corresponding to the float's power of two.

```
64 buffer: fopbuff      \ -- addr
```
Buffer in which output string is built.

```
32 buffer: frepbuff     \ -- addr
```
Buffer for use as the output of REPRESENT.

```
: roundfp       \ F: +f -- +f'
```
Add 0.5e(exp-precision-1).

```
: REPRESENT     \ F: 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.

```
: append        \ c-addr u $dest --
```
Add the string described by C-ADDR U to the counted string at $DEST. The strings must not overlap.

```
: (.sign)       \ flag $out --
```
Add '-' or nothing to the output string.

```
: (.mant)       \ binp $out n --
```
Add the mantissa string at *binp), produced by *\fo{REPRESENT*, to a counted string at *$out) with *\i{n* digits before the decimal point.

```
: (.exp)        \ exp(10) $out --
```
Add the exponent to the output string.

```
: (.initfop)    \ f -- ; -- exp(10)
```
initialise output conversion.

```
: (fs.)         \ F: f -- ; -- caddr len
```
Produce a string containing the number in scientific notation.

```
: (fe.)         \ F: f -- ; -- caddr len
```
Produce a string containing the number in engineering notation.

`: ff?  \ f: f -- f ; -- flag`
Return true if the number can be represented in free format.

`: (ff.)  \ F: f -- ; -- caddr len`
Produce a string containing the number in free notation. If the number cannot be displayed in free notation, scientific notation is uesed.

`: fs.  \ F: f --`
Display *f* in scientific notation:

```
x.xxxxxE[-]yy
```

`: fe.  \ F: f --`
Display *f* in engineering notation:

```
x.xxxxxE[-]yy
```

where the mantissa is 1 <= mantissa < 1000 and the exponent is a multiple of three.

`: ff.  \ F: f --`
Display *f* in free notation:

```
xxx.xxxxx
```

`: F.  \ F: f --`
Print the f.p. number in free format, xxxx.yyyy, if possible. Otherwise display using the x.xxxxEyy format.

7.19 Rounding

`f# 1.0e0 fconstant %ONE`
Floating point 1.0.

`f# 2.0e0 fconstant %two`
Floating point 2.0.

`: 1/f  \ F: f1 -- f2`
Reciprocal; *f2=1/f1.*

`: f2/  \ F: f1 -- f2`
Divide by 2.0; *f2=f1/2.0.*

`: FLOOR  \ F: r1 -- r2`
Floored round towards -infinity.

`: FROUND  \ F: r1 -- r2`
Round the number to nearest or even.

7.20 Trigonmetric functions

N.B. All angles are in radians.

`: DEG>RAD  \ F: n1 -- n2`
Convert degrees to radians.

`: RAD>DEG  \ F: n1 -- n2`
convert radians to degrees.

`: FSIN  \ F: f1 -- f2`
f2=sin(f1).

```
: FCOS          \ F: f1 -- f2
```
f2=cos(f1).

```
: FTAN          \ F: f1 -- f2
```
f2=tan(f1).

```
: FASIN         \ F: f1 -- f2
```
f2=arcsin(f1).

```
: FACOS         \ F: f1 -- f2
```
f2=arccos(f1).

```
: FATAN         \ F: f1 -- f2
```
f2=arctan(f1).

7.21 Logarithms and Powers

```
: FLN           \ F: f1 -- f2
```
Take the logarithm of f1 to base e and return the result.

```
: FLOG          \ F: f1 -- f2
```
Take the logarithm of f1 to base 10 and return the result.

```
: FEXP          \ F: f1 -- f2
```
f2=e^f1.

```
Synonym FE^X FEXP       \ F: f1 -- f2
```
Compatibility word.

```
: fexpm1        \ r1 -- r2
```
Raise e to the power *r1* and subtract one, giving *r2*.

```
: F10^X         \ F: f1 -- f2
```
f2=10^f1

```
: FX^N          \ n -- ; F: fx -- fx^n
```
fx^n=x^n where x is a float and n is an integer.

```
: F**   \ F: fx fy -- fx^fy
```
fn=X^Y where Y and Y are both floats.

```
Synonym FX^Y F**        \ --
```
Compatibility word for old code.

7.22 COSEC SEC COTAN and hyberbolics

```
: fcosec        \ F: f -- cosec(f)
```
Floating point cosecant.

```
: fsec          \ F: f -- sec(f)
```
Floating point secant.

```
: fcotan        \ f: f -- cot(f)
```
Floating point cotangent.

```
: fsinh         \ F: f -- sinh(f) ; (e^x - 1/e^x)/2
```
Floating point hyberbolic sine.

```
: fcosh         \ F: f -- cosh(f) ; (e^x + 1/e^x)/2
```
Floating point hyberbolic cosine.

```
: ftanh         \ F: f -- tanh(f) ; (e^x - 1/e^x)/(e^x + 1/e^x)
```
Floating point hyberbolic tangent.

```
: fasinh        \ F: f -- asinh(f) ; ln( f+sqrt(1+f*f) )
```
Floating point hyberbolic arcsine.

```
: facosh        \ F: f -- acosh(f) ; ln( f+sqrt(f*f-1) )
```
Floating point hyberbolic arccosine.

```
: fatanh        \ F: f -- atanh(f) ; ln( (1+f)/(1-f) )/2
```
Floating point hyberbolic arctangent.

7.23 Plugging floats into the system

```
: reals         \ -- ; turn FP system on
```
Switch the system and `NUMBER?` to permit floating point input using `FNUMBER?`. This action can be reversed by `INTEGERS`.

```
: integers      \ -- ; turn FP system off
```
Switch the system and `NUMBER?` to restore integer-only input.

7.24 Gotchas

7.24.1 Number formats

The ANS and Forth-2012 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`.

7.24.2 Floating point literals

Because we still have to support a variety of legacy floating point packages, plus new ones for as yet undefined CPUs, the handling of F.P. literals is far from perfect.

Our recommendation for use of floating point literals in both cross-compiled and target code, is to use `F#` and/or `FLITERAL`, e.g to compile 2.0e0, use one of the following

```
: foo  ... f# 2.0e0 ... ;
: foo  ... [ 2 s>f ] fliteral ... ;
```

8 Mixed Language Programming using SockPuppet

8.1 Introduction

With the ever increasing complexity of microcomputers such as the Cortex cores and systems, manufacturers are providing development hardware and software systems based on C libraries to make using their chips easier. Such libraries reduce the requirement for chip documentation. The conventional approach to providing support for development boards in Forth has been to manually port the C library sources to Forth. The **SockPuppet** system takes a different approach by providing an interface solution between Forth and C; the Forth system calls the underlying C libraries. In turn, this allows the details of the hardware to be abstracted away by the C libraries, whilst allowing the Forth system to provide a powerful, uniform and interactive user interface.

The MPE ARM/Cortex Forth cross-compiler supports calling functions in C or any language that can provide functions that use the AAPCS calling convention. This is an ARM convention documented in *IHI0042F_aapcs.pdf*. Calls with a variable number of parameters (**varargs**) are not supported.

The example code in both Forth and C is available for the Professional versions of the ARM Cortex cross compiler. The example code provides a simple GUI for an STM32F429I Discovery board using sample C code provided by ST and others. The interface is defined for Cortex-M CPUs only. If you need an ARM32 interface, contact MPE.

This work is directly inspired by Robert Sexton's Sockpuppet interface:

`https://github.com/rbsexton/sockpuppet`

His contribution and permission are gratefully acknowledged.

8.2 How the Forth to C interface works

Each function that is exported from the C world to the Forth world appears as one of a number of types of call. These words are called `externs`. You can handcraft these words in assembler, but the compiler includes code generators for several techniques. The call format and return values match the AAPCS standard used by ARM C compilers.

There are several ways to call `externs`. They have their pros and cons and are discussed in following sections.

- **SVC calls**. You just need to know the SVC numbers. SVC calls provide the greatest isolation between sections of code written in other languages. The functions foreign to Forth are accessed by SVC calls and/or jump tables. The example solution uses SVC calls for most foreign functions. Regardless of the primary call technique used, all techniques rely on a small number of SVC calls.
- **Jump table**. The base address of the table can be set at run time, e.g. by making a specific SVC call. The calling words fetch the run-time address from the table, given an index.
- **Double indirect call**. A primary jump table is at a fixed address and contains the addresses of secondary tables, which hold the actual routine addresses. The fixed address and both indices must be known at compile time. This technique is used by TI's Stellaris parts and some NXP parts to accesss driver code in ROM.

- **Direct calls** to the address of the routine. You need to know the address at compile time.

There is a practical limit of four arguments if you use SVC calls for the insulation between Forth and C. If you want this limit changed, call MPE! The other interface methods do not suffer from this limit.

It is a matter of convention between the Forth and C code as to parameter passing order. It can be changed by either side. MPE convention is for the left-most Forth parameter to be passed in R0. This matches the AAPCS code used by the hosted Forth compilers such as VFX Forth for ARM Linux. We strongly suggest that you use the MPE convention in order to take advantage of future Sockpuppet developments.

8.2.1 SVC calls

The examples use the MPE calling convention and are illustrated in assembler as well as by using the code generator.

```
SVC( 67 ) void BSP_LCD_DrawCircle( int x, int y, int r );
\ SVC 67 draw a circle of radius r at position (x,y).
```

```
CODE BSP_LCD_DrawCircle \ x y r --
\ SVC 67 draw a circle of radius r at position (x,y).
  mov r2, tos                    \ r
  ldr r1, [ psp ], # 4           \ y
  ldr r0, [ psp ], # 4           \ x
  svc # __SAPI_BSP_LCD_DrawCircle
  ldr     tos, [ psp ], # 4      \ restore TOS
  next,
END-CODE
```

When the SVC call occurs, the Cortex CPU stacks registers R0-R3, R12, LR, PC, xPSR on the calling R13 stack with R0 at the lowest address. The SVC handler places the address of this frame in R0/R4, extracts the SVC call number, reloads the AAPCS parameters from the frame and jumps to the appropriate C function. In this case

```
void BSP_LCD_DrawCircle(
  uint16_t Xpos, uint16_t Ypos, uint16_t Radius
);
```

SVC calls provide the highest insulation between Forth and C, but suffer from several issues.

- The SVC call mechanism is part of the Cortex interrupt and exception system. The assembler and/or C side of this uses code written in assembler to allow the C routines called from a jump table to to be AAPCS compliant.
- The SVC mechanism is inefficient compared to a direct AAPCS handler.
- Because SVC calls are part of the CPU interrupt mechanism, you have to care how long a call takes. Playing games with Cortex interrupt mechanism can fix this, but is complex.

8.2.2 Jump table

In order to avoid the penalties of the SVC call mechnism, you can make an array of function pointers in C or assembler and call functions using an index into the table.

```
jumptable:
  dd func0          ; address of function 0
  dd func1          ; address of function 1
  ..
```

We still need to know the address of the jump table. This is found using an SVC call (15) and stored in a variable. The jump table address could be hard-coded, but given the horrors of perverting link map files and the like, the overhead of a single SVC call is preferable.

```
SVC( 15 ) void * GetDirFnTable( void );
\ Define SVC call that returns the address of the
\ jump table.

variable JT     \ -- addr
\ Holds the address of the jump table.
JT holdsJumpTable
\ Tell the cross compiler where the jump table address
\ is held.

: initJTI       \ -- ; initialise jump table calls
  GetDirFnTable JT !  ;

JTI( n ) int open(
  const char * pathname, int flags, mode_t mode
);
```

If constructed in assembler, the SVC despatch table and the main jump table can be the same table; it's just a question of what you put in the table.

8.2.3 Double indirect call tables

Before use, you have to declare the base address of the primary ROM table used for calling ROM functions. For Luminary/TI CPUs, this will probably be:

```
$0100:0010 setPriTable
```

Now you can define a set of ROM calls, for example, again for a TI CPU.

```
DIC( 4, 0 ) void ROM_GPIOPinWrite(
   uint32 ui32Port, uint8 ui8Pins, uint8 ui8Val
);
```

where:

- `ROM_APITABLE` is an array of pointers located at `0x0100.0010`.

- `ROM_GPIOTABLE` is an array of pointers located at `ROM_APITABLE[4]`.
- `ROM_GPIOPinWrite` is a function pointer located at `ROM_GPIOTABLE[0]`.

Parameters:

- `ui32Port` is the base address of the GPIO port.
- `ui8Pins` is the bit-packed representation of the pin(s).
- `ui8Val` is the value to write to the pin(s).

Description:

Writes the corresponding bit values to the output pin(s) specified by `ui8Pins`. Writing to a pin configured as an input pin has no effect. The pin(s) are specified using a bit-packed byte, where each bit that is set identifies the pin to be accessed, and where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

Returns:

None.

To call this function, use the Forth form:

```
port pins val ROM_GPIOPinWrite
```

8.2.4 Direct calls

Where the address of the routine is known at the Forth compile time, you can use a direct call.

```
DIR( addr ) int foo( int a, char *b, char c );
```

The Forth word marshalls the parameters and calls the subroutine at target address *addr*.

8.3 SVC call number list

The demonstration code provided here mostly uses SVC calls. To call the SVC, a small Forth word puts the arguments into registers, calls the SVC, and then returns a value if required. SVCs are identified by a number, which corresponds here to an index into a a table.

The list must match the equivalents in the C code `SVC_syscall_table[]`. To ease sharing of data between the Forth and C systems, you can use the `enum` parser described in the "Interpreter directives" chapter of the generic cross compiler manual.

The first 16 entries (0..15) are defined by the Sockpuppet system. Each application is free to use entries 16 onwards as suits the application.

This list will be different for every application.

```
const=equ       \ constants invisible on target
\ const=constant  \ constants visible on target
#00 const __SAPI_00_ABIVersion          \ SVC 00  - Required
#01 const __SAPI_01_GetLinkList         \ SVC 01  - Required
#02 const __SAPI_02_PutChar             \ SVC 02
```

```
#03 const __SAPI_03_GetChar             \ SVC 03
#04 const __SAPI_04_GetCharAvail        \ SVC 04
#05 const __SAPI_05_PutCharHasRoom      \ SVC 05
#06 const __SAPI_06_SetIOCallback       \ SVC 06
#07 const __SAPI_07_GetTimeMS           \ SVC 07 - Required
#08 const __SAPI_08_NVICReset           \ SVC 08

#10 const __SAPI_10_LauchUserApp        \ SVC 10 - Reserved
#11 const __SAPI_11_MPULoad             \ SVC 11 - Reserved
#12 const __SAPI_12_GetPrivs            \ SVC 12 - Reserved
#13 const __SAPI_13_GetUsageCPU         \ SVC 13 - Reserved
#14 const __SAPI_14_PetWatchdog         \ SVC 14 - Reserved
#15 const __SAPI_15_GetFnTable          \ SVC 15 - Required

#20 const __SAPI_20_GetIP               \ SVC 20
#21 const __SAPI_21_PBuf_Ram_Alloc      \ SVC 21
#22 const __SAPI_22_PBuf_Free           \ SVC 22
#23 const __SAPI_23_DNS_GetHostByName   \ SVC 23
#24 const __SAPI_24_UDP_GetPCB          \ SVC 24
#25 const __SAPI_25_UDP_Connect         \ SVC 25
#26 const __SAPI_26_UDP_Recv            \ SVC 26
#27 const __SAPI_27_UDP_Send            \ SVC 27
#28 const __SAPI_28_GetTimeMS           \ SVC 28
#29 const __SAPI_29_GetKeyButton        \ SVC 29
#30 const __SAPI_30_SetLeds             \ SVC 30
#31 const __SAPI_31_SetLedsHW           \ SVC 31
\ Graphics and Touchscreen
#32 const __SAPI_32_WaitForPressedState \ SVC 32
#33 const __SAPI_33_BSP_TS_GetState     \ SVC 33
#34 const __SAPI_34_Touchscreen_Cal     \ SVC 34
#35 const __SAPI_35_Touchscreen_demo    \ SVC 35

#37 const __SAPI_BSP_LCD_Init           \ SVC 37
#38 const __SAPI_BSP_LCD_GetXSize       \ SVC 38
#39 const __SAPI_BSP_LCD_GetYSize       \ SVC 39
#40 const __SAPI_BSP_LCD_LayerDefaultInit           \ SVC 40
#41 const __SAPI_BSP_LCD_LayerDefaultInitPixelForm  \ SVC 41
#42 const __SAPI_BSP_LCD_SetTransparency            \ SVC 42
#43 const __SAPI_BSP_LCD_SetLayerAddress            \ SVC 43
#44 const __SAPI_BSP_LCD_SetColorKeying             \ SVC 44
#45 const __SAPI_BSP_LCD_ResetColorKeying           \ SVC 45
#46 const __SAPI_BSP_LCD_SetLayerWindow             \ SVC 46
#47 const __SAPI_BSP_LCD_SelectLayer                \ SVC 47
#48 const __SAPI_BSP_LCD_SetLayerVisible            \ SVC 48
#49 const __SAPI_BSP_LCD_SetTextColor   \ SVC 49
#50 const __SAPI_BSP_LCD_SetBackColor   \ SVC 50
#51 const __SAPI_BSP_LCD_GetTextColor   \ SVC 51
#52 const __SAPI_BSP_LCD_GetBackColor   \ SVC 52
#53 const __SAPI_BSP_LCD_SetFont        \ SVC 53
#54 const __SAPI_BSP_LCD_GetFont        \ SVC 54
#55 const __SAPI_BSP_LCD_ReadPixel      \ SVC 55
#56 const __SAPI_BSP_LCD_DrawPixel      \ SVC 56
```

```
#57 const __SAPI_BSP_LCD_Clear          \ SVC 57
#58 const __SAPI_BSP_LCD_ClearStringLine      \ SVC 58
#59 const __SAPI_BSP_LCD_DisplayStringAtLine  \ SVC 59
#60 const __SAPI_BSP_LCD_StrAtLineMode        \ SVC 60
#61 const __SAPI_BSP_LCD_DisplayStringAt      \ SVC 61
#62 const __SAPI_BSP_LCD_DisplayChar    \ SVC 62
#63 const __SAPI_BSP_LCD_DrawHLine      \ SVC 63
#64 const __SAPI_BSP_LCD_DrawVLine      \ SVC 64
#65 const __SAPI_BSP_LCD_DrawLine       \ SVC 65
#66 const __SAPI_BSP_LCD_DrawRect       \ SVC 66
#67 const __SAPI_BSP_LCD_DrawCircle     \ SVC 67
#68 const __SAPI_BSP_LCD_DrawPolygon    \ SVC 68
#69 const __SAPI_BSP_LCD_DrawEllipse    \ SVC 69
#70 const __SAPI_BSP_LCD_DrawBitmap     \ SVC 70
#71 const __SAPI_BSP_LCD_FillRect       \ SVC 71
#72 const __SAPI_BSP_LCD_FillCircle     \ SVC 72
#73 const __SAPI_BSP_LCD_FillTriangle   \ SVC 73
#74 const __SAPI_BSP_LCD_FillPolygon    \ SVC 74
#75 const __SAPI_BSP_LCD_FillEllipse    \ SVC 75
#76 const __SAPI_BSP_LCD_DisplayOff     \ SVC 76
#77 const __SAPI_BSP_LCD_DisplayOn      \ SVC 77
#78 const __SAPI_BSP_LCD_GetFontTable   \ SVC 78

#81 const __SAPI_tsWaitUp               \ SVC 81
#82 const __SAPI_tsWaitDown             \ SVC 82
```

8.4 Words containing an SVC call

`svc( 0 ) int SAPI-Version( void );`
SVC 00: Return the version of the API in use.

`svc( 1 ) int GetSharedVars( void );`
SVC 01: Get the address of the shared variable list.

`svc( 15 ) int GetSvcFnTable( void );`
SVC 15: Get the address of the SVC function table.

`: FN  \ n --`
Call a function in the SVC function table directly. SVC calls that take a long time may/will block interrupts such as the system ticker, and thus will fail. To avoid this, such calls should be called directly so that they do not affect the interrupt priority level. The functions **must** be declared as

```
void function( void );
```

as the normal SVC parameter passing mechanism is completely bypassed. Note also that these functions inhibit the Forth multitasker for the duration of the call. It is usually better to refactor the C library if you wish to use the Forth multitasker.

`svc( 28 ) int ticks( void );`
SVC 28: Get the ms counter. Returns a value in milliseconds that eventually wraps around.

`svc( 29 ) int SAPI_GetKeyButton( void );`
SVC 29: Get the Key Button value

```
svc( 30 ) void SAPI_SetLeds( int mask );
```
SVC 30: Set LED3 (bit0) and LED4 (bit1) via BSP.

```
svc( 31 ) void SAPI_SetLedsHW( int mask );
```
SVC 31: Set LED3 (bit0) and LED4 (bit1) via direct hardware access

```
svc( 33 ) void SAPI_33_BSP_TS_GetState( void * TSstruct );
```
SVC 33: Get the touchscreen coordinates into a structure.

8.5 Reserved calls

These calls are commented out by default and are not present in all systems. However, the SVC numbers are reserved. Note that some stack manipulation is required in some words - they may not be direct translations.

```
CODE SetIOCallback  \ addr iovec read/write -- n
```
Set the IO Callback address for a stream. *Iovec*, read/write (r=1), address to set as zero. Note the minor parameter swizzle here to keep the old value on TOS.

```
CODE RestartForthApp ( addr ) \ --
```
Do a stack switch and startup the user App. Its a one-way trip, so don't worry about stack cleanup.

```
CODE MPULoad  \ --
```
Ask for MPU entry updates.

```
CODE privmode  \ --
```
Request Privileged Mode. In some systems, this is a huge security hole.

```
CODE PetWatchDog  \ --
```
Refresh the watchdog

```
CODE GetUsage  \ -- n
```
The number of CPU cycles consumed in the last second.

8.6 GUI SVC calls

These calls are defined for the demonstration code. Your application code may reuse the numbers for other functions.

```
SVC( 49 ) void BSP_LCD_SetTextColor( int color );
```
SVC 49: Set text and line drawing colour.

```
SVC( 50 ) void BSP_LCD_SetBackColor( int color );
```
SVC 50: Set background colour for text

```
svc( 56 ) void BSP_LCD_DrawPixel( int x, int y, int colour );
```
SVC 56: Draw a pixel at a at screen position (x,y)

```
svc( 57 ) void SAPI_BSP_LCD_Clear( int colour );
```
SVC 57: Set the LCD to the given colour.

```
svc( 60 ) void BSP_LCD_DisplayStringAtLineMode( int line, char * string, int mode);
```
SVC 60 Display the text at a on line a using the given mode

```
svc( 61 ) void BSP_LCD_DisplayStringAt( int x, int y, char * string, int mode );
```
Draw a string at position (x,y) in the given alignment mode.

```
$01 constant CENTER_MODE   \ center mode
$02 constant RIGHT_MODE    \ right mode
$03 constant LEFT_MODE     \ left mode
```

```
svc( 63 ) void BSP_LCD_DrawHLine( int x, int y, int length );
```

SVC 63: Draw a horizontal line at position (x,y) of length.

```
svc( 64 ) void BSP_LCD_DrawVLine( int x, int y, int length );
```

SVC 64: Draw a vertical line at position (x,y) of length.

```
svc( 65 ) void BSP_LCD_DrawLine( int x1, int y1, int x2, int y2 );
```

SVC 65: Draw a line between two points.

```
svc( 66 ) void BSP_LCD_DrawRect( int x, int y, int w, int h );
```

SVC 66: Draw a rectangle.

```
svc( 67 ) void BSP_LCD_DrawCircle( int x, int y, int radius );
```

SVC 67: Draw a circle of the given radius at screen position (x,y)

```
SVC( 70 ) void BSP_LCD_DrawBitmap( int x, int y, void * image );
```

SVC 70: draw the bitmap image at addr at the screen position (x,y).

```
svc( 71 ) void BSP_LCD_FillRect( int x, int y, int w, int h );
```

SVC 71: Draw a filled rectangle at (x,y) of size (w,h).

```
svc( 72 ) void BSP_LCD_FillCircle( int x, int y, int radius );
```

SVC 72 draw a filled circle of the given radius at screen position (x,y)

8.7 Debug tools

```
: mainSP@       \ -- addr
```

Return the main stack pointer.

```
: procSP@       \ -- addr
```

Return the process stack pointer.

8.8 Sharing data between Forth and C

In the MPE environment, `VALUE`s work very well for this. `VALUE`s are defined at compile time and can be updated at runtime. So the basic logic is - walk the dynamic list, and if you find something that is already defined, assume it's a `VALUE` otherwise, generate a `CONSTANT`.

The code is based around 32 byte records. They are accessed from both Forth and C.

8.8.1 C Linkage structure

```
#define DYNLINKNAMEMLEN 22

typedef struct {
  // This union is a bit crazy, but it's the simplest way of
  // getting the compiler to shut up.
  union {
     void (*fp) (void);
     int*  ip;
     unsigned int    ui;
     unsigned int*  uip;
     unsigned long* ulp;
  } p;            ///< Pointer to the object of interest (4)
  int16_t size;   ///< Size in bytes (6)
  int16_t count;  ///< How many (8)
  int8_t kind;    ///< Is this a variable or a constant? (9)
  uint8_t strlen; ///< Length of the string (10)
  const char name[DYNLINKNAMEMLEN]; ///< Null-Terminated C string.
} runtimelink_t;
```

When the Forth system powers up it runs the Forth word `dy-populate` which uses SVC call 01 to get the address of the `dynamiclinks[]` table, and walks through the table creating Forth named variables whose addresses match those in the C system.

A Forth word `dy-show` is provided to list the entries in the table.

8.8.2 Forth Linkage structure

```
interpreter
: hword  2 field  ;
: byte   1 field  ;
target

struct /runtimelink     \ -- len
\ Forth equivalent of the C structure above.
  int   fdy.val         \ usually a pointer     0, 4
  hword fdy.size        \ size in bytes         4, 2
  hword fdy.count       \ how many              6, 2
  byte  fdy.type        \ variable or constant  8, 1
  byte  fdy.nlen        \ name length           9, 1
  22 field fdy.zname    \ zero terminated name  10, 22
end-struct
```

The accessor words just read the fields defined above. They are defined as compiler macros. For interaction on the target, use the field names above.

```
compiler
: dy.val        fdy.val @ ;     \ addr -- n
: dy.size       fdy.size w@ ;   \ addr -- w
: dy.count      fdy.count w@ ;  \ addr -- w
```

```
: dy.type       fdy.type c@ ;   \ addr -- c
: dy.name       fdy.nlen  ;     \ addr -- addr'
target
```

8.8.3 Accessing shared data from Forth

`: dy-recordlen  \ -- n`
Return the recordlength. Its the first thing.

`: dy-first  \ -- addr`
return the first entry in the dynamic variable list

`: dy-next  \ addr -- addr`
return the next entry in the dynamic variable list

`: dy-stuff  \ n xt --`
Store n in the VALUE defined by xt

`: make-const  \ n addr len --`
Create a `CONSTANT` of name *addr/len* and value *n* by laying down a header and some machine code. A key trick is that we have to lay down a pointer to the first character of the definition after we finish This is what a constant looks like after emerging from the compiler.

```
( 0002:0270 4CF8047D Lx.} )  str r7, [ r12, # $-04 ]!
( 0002:0274 004F .O )        ldr r7, [ PC, # $00 ] ( @$20278=$7D04F84C )
( 0002:0276 7047 pG )        bx LR
```

`: dy-create  \ addr --`
Look for an entry and if its a value, update it.

`: dy-populate  \ --`
Walk the table and make the constants.

`: dy-print  \ addr --`
Dump out an entry as a set of constants

`: dy-show  \ --`
Walk the table and show the entries

`: dy-compare  \ addr n addr -- n`
Given a record address and a string, figure out if it is the one we're looking for.

`: dy-find  \ addr n -- addr|0`
Walk the dynamic variable list and find a string, return its address if found, else 0.

8.9 The Sockpuppet C code

The demonstration system runs on an STM32F429I Discovery board with a QVGA colour display. The Forth demonstration code uses a mixture of SVC calls and direct calls into the C jump table. This section documents the C code. The examples are taken from the file *SockPuppetInterface.c*.

8.9.1 SVC handler

The SVC handler runs a short piece of assembler that permits the use of both the main and

process stacks. This version of the code is AAPCS compliant for up to four arguments and a 32-bit return value. The structure of the code is imposed by the requirements of the Cortex-M interrupt/exception handler for tail-chaining and late arrivals.

```
void SVC_Handler(void)
{
// on entry, R0..R3, R12, LR, PC, PSR have been saved
  __asm( "tst    lr, #0x4" );         // Figure out which stack
  __asm( "ite    eq" );
  __asm( "mrseq  r0,msp" );           // Main stack
  __asm( "mrsne  r0,psp" );           // Process/Thread Stack

  __asm( "push   { r4, lr }" );       // now we can preserve more
  __asm( "mov    r4, r0" );           // copy SP; R4 is preserved
  __asm( "ldr    r1,[r0,#24]" );      // get the stacked PC
  __asm( "ldrb   r1,[r1,#-2]" );      // extract the svc call number
                                      // range checking goes here
  __asm( "ldr    r2 ,=SVC_syscall_table" );
  __asm( "ldr    r12, [r2, r1, LSL #2]" );  // R12 = function address
  __asm( "ldm    r4, { r0-r3 }" );    // restore AAPCS args from R0..R3
  __asm( "blx    r12" );              // call function
  __asm( "str    r0, [ r4, #0 ]" );   // save return value
  __asm( "pop    { r4, pc }" );       // restore
}
```

8.10 Building the demonstration

The demonstration code is for an STM32F429I Discovery board, which includes a QVGA colour display.

The Forth sources are provided in the Forth cross-compiler distribution in the *Lib/SockPuppet* directory. The Forth toolchain is the MPE Forth cross compiler.

The demonstration C source code is provided as a ZIP file with the cross compiler download. The C toolchain is the version of GCC maintained by ARM at:

```
https://launchpad.net/gcc-arm-embedded
```

Flash programming is performed by ST's STM32 ST-Link utility, usually found at:

```
http://www2.st.com/content/st_com/en/products
/embedded-software/development-tool-software/stsw-link004.html
```

The ST-Link software and drivers must be installed before the board can be programmed.

8.10.1 Building the C layer

Windows Batch files are supplied in the *C_files* folder:

- MakeAll.bat make the C library system
- MakeAllFlash.bat make the C library system and download it to the target Flash.

These files trundle around the distribution to run the make systems, gcc and to program the Flash.

The C system is based on the STM32F4 LCD system by Pierpaolo Bagnasco, available from:

`http://www.pierpaolobagnasco.com/category/stm32f4xx/`

`http://www.pierpaolobagnasco.com/2014/07/13/stm32f429-discovery-display/`

Pierpaolo uses Eclipse to auto-generate makefiles so that the ST supplied libraries can be compiled using the GNU C ARM command line interface tools

`http://thehackerworkshop.com/?p=1056`

To avoid being tied to any particular IDE, the makefiles are now edited manually. If you want to know the gory details of editing makefiles produced by Eclipse, Google is your friend.

The following files have been modified or added:

- C_files\src\main.c (modified)
- C_files\src\ SockPuppetInterface.c/h (added)
- C_files\src\stm32f4xx_it.c (modified)

Main.c initialises hardware components that we are using, then enters an infinite loop testing for the presence of the Forth system in the high 1 Mbyte of FLASH memory. If found, the C system jumps to the StartCortex entry point in the Forth system.

SockPuppetInterface.c/h define the SockPuppet interface for the C toolchain.

stm32f4xx_it.c defines the SVC exception handler that jumps to the function in the SVC_syscall_table corresponding to the SVC number.

The C compiler used is the GNU Tools ARM Embedded\5.2 2015q4 gcc-arm-none-eabi-5_2-2015q4-20151219-win32.exe available from here:

`https://launchpad.net/gcc-arm-embedded/5.0/5-2015-q4-major`

The makefile Make.exe is from GNU ARM Eclipse\Build Tools\2.6-201507152002 gnuarmeclipse-build-tools-win32-2.6-201507152002-setup.exe available from here:

`http://sourceforge.net/projects/gnuarmeclipse/files/Build%20Tools/`

A copy of make.exe is included in the C_files\Debug\ directory.

The output file is *C_files\Debug\Display.hex* and runs from location 0x0800000. 1 Mbyte of Flash and 128K bytes of RAM are allocated for the C system. This is grotesquely over the top but works. For a production environment the memory map should be edited.

8.10.2 Building the Forth system

There is a control file for the SockPuppet demo:

```
Cortex/Hardware/STM32F4/SP429disco.ctl
```

Edit this file so that the line

```
afterwards sh "C:\MyApps\ST-Linkv3.8.1\ST-LINK Utility\ST-LINK_CLI.exe"  -P SP429DISCO.hex
```

contains the correct path to the command line version of the ST-Link utility.

Set up a new AIDE project to compile this file or compile it directly. At the end the Forth image will have been programmed and the application run.

8.10.3 Required target files

The build processes should have programmed the required files. In case you just have the hex files, you can program them into the board using the ST-Link utility. Find the files

- Display.hex - the C image. Usually in *C_files\Debug\Display.hex*.
- SP429DISCO.hex - the Forth image. Usually in *<xArmCortex>\Cortex\Hardware\STM32F4\SP429DISCO.hex*.

8.10.4 Memory map

The STM32F429 chip has 2 Mbytes of Flash and 256 Kbytes of RAM on chip. The available memory is divided equally between the C and the Forth portions:

```
C System Flash      0x08000000 0x100000
Forth System Flash  0x08100000 0x100000
C System RAM        0x20000000 0x020000
Forth System RAM    0x20020000 0x020000
```

The STM32F429 fetches the stack and PC from address 0 at reset, and 0x400 bytes from 0x08000000 are initially mapped to address 0, allowing the chip to effectively boot from the start of the C system Flash at 0x08000000.

The C system checks the location 4 bytes offset from the start of the Forth system Flash, 0x08100004, which contains the address of the word `StartCortex` in the Forth system. Note that the value at this address always has bit 0 set to 1, indicating Thumb code, since the entire chip runs in Thumb mode. The value at the start of the Forth system Flash, 0x08100000, normally contains the address to be used for the stack, but the xArmCortex Forth start-up code has been modified not to use this value, but to continue to use the C system stack. This allows C library functions to be called in their own stack environment.

If the value at 0x08100004 is not 0xFFFFFFFF or 0x00000000 the C system hands control over to the Forth system, which can then access C library functions via the SockPuppet interface.

9 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*.

9.1 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.

9.2 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.

9.3 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.

9.4 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.

9.5 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.

9.6 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.

9.7 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`.

9.8 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".

9.9 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.

9.10 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.

10 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.

10.1 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.

10.2 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.

10.3 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.

11 NXP LPC17xx IAP routines

The IAP is accessed by calling a Thumb routine at the IAPentry address with the address of a command block in R0 and the address of a status/result block (RAM) in R1. All the IAPxxx words return a 0 result on success. Note also that all interrupts are disabled for the duration of an IAP call, and therefore that ticker interrupts will not be serviced. In particular, the sector erase command may take 400ms, and a write of a 512 byte line may take 1ms.

```
0 equ FullIAP?          \ -- n
```
If this EQUate is set non-zero, additional IAP routines are compiled, e.g. to get the bootloader version number and the device part number. The definition here is only used if it has not been previously defined.

```
5 cells buffer: IAPcmd  \ -- addr ; max 5 cells
```
Command input buffer for IAP routines.

```
4 cells buffer: IAPres  \ -- addr ; max 4 cells
```
Result output buffer from IAP routines.

```
code IAP        \ *cmd *res --
```
The primitive to call the IAP routines. Interrupts are disabled for the duration of the IAP call.

```
: IAPprep       \ start end -- res
```
Prepare sectors for erase/write.

```
: IAPcopy       \ Rsrc Fdest len -- res
```
Copy/program len bytes from Rsrc in RAM to Fdest in Flash.

```
: IAPerase      \ start end -- res
```
Erase the inclusive range of sectors.

```
: IAPcheck      \ start end -- res
```
Blank check the inclusive range of sectors.

```
: IAPcompare    \ Rsrc Fdest len -- res
```
Compare len bytes from Rsrc in RAM to Fdest in Flash.

```
: IAPbootver    \ -- bootver
```
Return the 16 bit boot code version. The high byte is the major version and the low byte is the minor version.

```
: IAPpartno     \ -- part#
```
Return the NXP part number.

```
: IAPserialno   \ -- addr
```
Return a pointer to the 128 bit (4 cells) NXP LPC17xx serial number.

12 LPC17xx Flash tools

These tools are provided for applications which reserve part of the Flash for data. This code uses the IAP routines in *IAP17xx.fth*.

N.B. An erase causes the whole of the sector containing that address to be erased.

N.B. All writes to Flash must be from RAM as the Flash is unavailable at the time any part is being written.

12.1 Flash primitives

Although some of these routines are not the most efficient for the LPC1700 series, they are designed to be easily expanded for future LPC17xx parts with as yet unknown sector sizes.

Sector tables contain the number of sectors and starting offset of each sector, plus a dummy start address which enables the size of the last sector to be calculated. The boot block sector is **not** included in the table. The sector table is given by `SecTab` which is defined in *FlashTables.fth*.

`: SectorN  \ n -- addr len`
Convert sector number (zero based) to base address and length

`: FindSecN  \ addr -- n`
Find the sector number containing address addr. If addr is outside the internal Flash range, n is set to -1.

`: .src/dest  \ src dest -- src dest`
Display the addresses and contents of the source and destination.

12.2 Flash driver

`cell buffer: FlDest  \ -- addr`
Holds next destination address.

`cell buffer: #PrgErrs  \ -- addr`
Holds error count.

`: (Prog512)  \ src dest --`
Program 512 bytes at *src* to *dest*. The *src* address must be in RAM. On error, the variable `PrgErrs` is incremented.

`: Prog512  \ src dest --`
Program 512 bytes at src to dest. Increment error counter on error. `ISPram` is used as an intermediate buffer to avoid avoid problems/bugs in some LPC devices.

`: (EraseFlash)  \ dest dlen --`
Erase the flash for the given range. Note that complete sectors in the range will be erased. If you want to write partial sectors, check that they contain $FF in all bytes before programming in order avoid having to erase them first.

`: (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 512 byte boundary and *len* is rounded up to the next 512 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.

13 Reprogramming the LPC17xx serially

13.1 Introduction

Reprog17xx.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.

13.2 Configuration

```
#1700 equ LPCtype       \ -- 1700
```
Define the LPC type, using 1700 as the generic type

```
#512 kb equ /LPCflash   \ -- u
```
Max size of LPC17xx internal Flash

```
#96000000 equ system-speed      \ -- Hz
```
System operating speed in HZ.

```
0 equ FullIAP?          \ -- n
```
Set this equate non-zero to compile additional IAP routines. See *IAP17xxx.FTH* for more details.

13.3 Items of interest

The stacks have already been set up by the calling program.

```
include %CpuDir%\CortexDef              \ Cortex CPU equates
include %CpuDir%\sfrLPC17xx             \ 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 0 or UART base address to use. If set to 0, the default is UART0. This allows the calling application to select which UART to use.

```
l: CLD_clockspeed       \ -- addr
```
Contains the clock speed to use. Filled in by the calling application. Default is 96 MHz.

```
l: CLD_#FlSectors       \ -- addr
```
Number of sectors in the flash. Default is 30 for 512k devices.

```
: 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%\MinSerLPC17xxp          \ polled serial line driver
include %AppDir%\delays                  \ software delays
include %CpuDir%\Drivers\rebootLPC17xx   \ reboot using watchdog

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 %HwDir%\IAP17xx                  \ IAP access
include %HwDir%\FlashTables              \ Flash sector tables
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.

14 NXP LPC17xx Reflashing

14.1 Introduction

If you destroy the application in the internal Flash, you must use the NXP ISP loader to reload an Intel Hex file. A suitable baud rate is 38400 baud. To use this, ensure that port P2.10 is low. There is a link on the board that (when installed) enforces this. Then reset the board and use the Philips loader. Then close the Philips loader. Ensure that the P2.10 link is removed before resetting the board. Then connect using AIDE's PowerTerm or HyperTerm.

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 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 LPC17xx 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.

14.2 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.

`create reprog.img \ -- addr`
The start address of the reprogramming code

```
  data-file %HwDir%/ReProg/REPROG17XX.IMG equ /reprog   \ -- len
```
Generic 17xx:The length of the reprogramming code loaded into the dictionary.

`$1000:0200 equ reprogrun \ -- addr`
The run time address of the reprogramming code. This **must** match the start address of the `CDATA` section defined in *Reprog17xx.ctl*, and that section **must not** overlap the run-time stacks and user area.

`0 value ReflashDev \ -- addr`
Holds 0 or a UART base address. If set to 0, the reflash code will use the default device for the XModem transfer. Otherwise it will use the value here as the base address of the UART to use.
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 - number of sectors in the Flash.

`: 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.

15 LPC17xx GPIO utilities

The code in *Cortex\Drivers\gpioLPC17xx.fth* provides tools to access the GPIO/FIO pins on a bit by bit basis. The code is written for ease of use rather than performance.

There are similar files in the *Cortex\Drivers* directory for different CPU families. They all use a very similar set of words. Do not be afraid to browse the supplied source code!

15.1 Configuration

The equates in this block are defaults used if the equates have not been defined before this file is compiled.

```
_FIO0 equ PortBase      \ -- addr
```
Base address of port definitions.

```
$20 equ /PortBlock      \ -- len
```
Number of bytes per port for peripheral registers.

```
5 equ #PortShift        \ -- u
```
Number of bits to shift to offset by `/PortBlock`.

15.2 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.

```
: PIO:          \ port# bit# -- ; -- iobit#
```
Define a port I/O bit by name. For example

```
  2 11 PIO: P2.11
```

15.3 IO pin access

```
: setPin        \ iobit# --
```
Set the pin high.

```
: clrPin        \ iobit# --
```
Set the pin low.

```
: getPin        \ iobit# -- 0/1
```
Read the pin state.

15.4 IO pin configuration

```
: isInput       \ iobit# --
```
Set the pin as an input.

```
: isOutput      \ iobit# --
```
Set the pin as an input.

```
: isPinsel      \ sel# iobit# --
```
Set the pin configuration 0..3. By default, nearly all pins are in I/O pin mode after reset.

```
: isPinmode     \ mode# iobit# --
```
Set the pin mode 0..3. Not available for CPUs that do not have `PINMODE` registers.

```
0 constant PulledUp     \ -- mode#
```
Used as the mode# with `isPinMode` above to define how the pin operates as an input.

```
2 constant NotPulled    \ -- mode#
```
Used as the mode# with `isPinMode` above to define how the pin operates as an input.

```
3 constant PulledDown   \ -- mode#
```
Used as the mode# with `isPinMode` above to define how the pin operates as an input.

15.5 Test code for Olimex LPC1766-STK

```
decimal

1 25 pio: led1
0  4 pio: led2
1 18 pio: Usb_Link_Led
2  9 pio: Usb_Connect_Led
0 23 pio: But1
2 13 pio: But2

: 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
;
```

16 Further information

16.1 MPE courses

MicroProcessor Engineering runs the following standard courses, which can be held at MPE or at your own site:

- **Architectual Introduction to Forth** (AIF): A three-day course for those with little or no experience of Forth, but with some programming experience. The AIF course provides an introduction to the architecture of a Forth system. It shows, by teaching and by practical example how software can be coded, tested and debugged quickly and efficiently, using Forth's interactive abilities.
- **Embedded Software for Hardware Engineers** (ESHE): A three-day course for hardware and firmware engineers needing to construct real-time embedded applications using Forth cross-compilers. Includes multitasking and writing interrupt handlers.

Custom courses are available

- **Quick Start Course** (QSC): A very hands-on tailored course on your site using your own hardware, and includes installation of a target Forth on your hardware, approaches to writing device drivers, designing a framework for your application and whatever else you need. The course is usually three days long.
- Other custom courses we provide are for Open Boot and Open Firmware. These are derived from the AIF course above.

16.2 MPE consultancy

MPE is available for consultancy covering all aspects of Forth and real-time software and hardware development. Apart from our Forth experience, MPE staff have considerable knowledge of embedded hardware design, Windows, Linux and DOS.

Our software orbits the earth, will land on comets, runs construction companies, laundries, vending machines, payment terminals, access control systems, theatre and concert rigging, anaesthetic ventilators, art installations, trains, newspaper presses and bomb disposal machines.

We have done projects ranging from a few days to major international projects covering several years, continents and many countries. We can operate to fixed price and fixed term contracts. Projects by MPE cover topics such as:

- Custom compiler developments, including language extensions such as SNMP, and new CPU implementations,
- Custom hardware design and compiler installations,
- Portable binary system for smart card payment systems,
- Machinery controllers,
- Connecting instrumentation to web sites,
- Virtual memory systems,
- Code porting to new hardware or operating systems.

We also have a range of outside consultants covering but not limited to:

- Communications protocols

- Windows device drivers
- All aspects of Linux
- Safety critical systems
- Project management (including international)

16.3 Recommended reading

A current list of books on Forth may be found at:
`http://www.mpeforth.com/books.htm`

For an introduction to Forth, and all available in PDF or HTML:

- "Programming Forth" by Stephen Pelc. About modern Forth systems.
- "Starting Forth" by Leo Brodie. A classic, but very dated.
- "Thinking Forth" by Leo Brodie. A classic.

For more experienced Forth programmers:

- "Object Oriented Forth" by Dick Pountain
- "Scientific Forth" by Julian Noble

Other miscellaneous Forth books:

- "Forth Applications in Engineering and Industry" by John Matthews
- "Stack Machines: The New Wave" by Philip J Koopman Jr

All of these books can be supplied by MPE.

Index

D

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

V

W

X