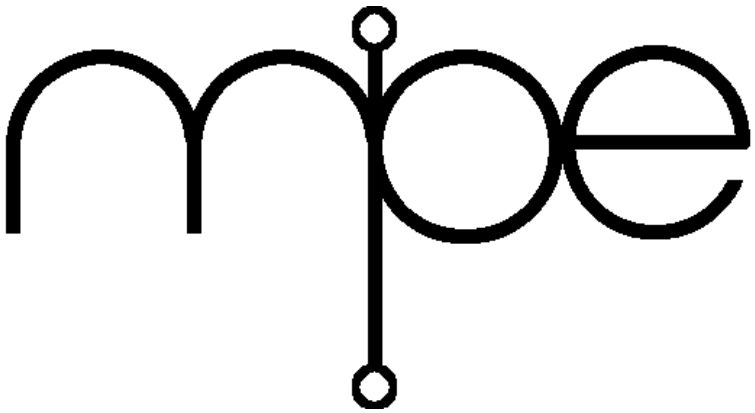
ARM Target Code

v7.4

Microprocessor Engineering Limited



Published by Microprocessor Engineering

ARM Target Code v7.4
User manual
Manual revision 7.40
25 January 2016

Software
Software version 7.4

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 Generic start up

The file ARM\INITARM.FTH contains the generic vector table, Forth start up code and Forth initialisation tables. It must be the first file compiled that generates code unless you are compiling operating specific code, such as with an AIF header.

For CPUs such as the Atmel AT91 series which have a vectored interrupt controller, this file is not used, and a variant of this code will be included in a CPU and hardware specific file INITxxxx.FTH. Such files may be found in the HARDWARE directory e.g. Hardware\EB55\InitARM55800.fth is used for the EB55 board which has an AT91x55800A CPU.

1.1 Vectors

This code branches by loading the PC with an ABSOLUTE address contained in a location at a RELATIVE offset from the LDR instruction. This avoids the use of a relative branch which may be out of range if the execution address is high in memory.

For faster response speed, you may wish to place your FIQ handler here, replacing the ldr pc, FIQV instruction with the first instruction of the FIQ handler. This saves the ARM from the extra overhead of having to branch to the handler.

The following vectors usually end up in RAM after the EPROM/Flash is remapped into high memory and RAM is remapped to low memory.

1.2 Start of Forth system

This section contains initial values of Forth registers and the code to initialise and run Forth.

2 ARM code definitions

2.1 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>
```

2.2 Register usage

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
r8          fsp     float stack pointer
r0-r7       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.

2.3 Configuration

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

```
false equ HIGH-LEVEL-ROLL       \ -- flag
```

Set this equate true to compile a high level version of ROLL.

```
false equ CLZ?  \ -- flag
```

Set this non-zero to compile CLZ.

```
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.

2.4 Logical and relational operators

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

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

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

`CODE NOT \ x -- x'`
Perform a bitwise NOT on the top stack item and retain result.

`CODE INVERT \ x -- x'`
Perform a bitwise NOT on the top stack item and retain result.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

`CODE 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.

`CODE <<  \ x1 u -- x1<<u`
Logically shift X1 by U bits left. Obsolete - replaced by LSHIFT.

`CODE >>  \ x1 u -- x1<<u`
Logically shift X1 by U bits right. Obsolete - replaced by RSHIFT.

2.5 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.

2.6 Basic arithmetic

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

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

```
CODE NOOP       \ --
```
A NOOP, null instruction.)

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

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

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

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

```
CODE 1-         \ n1|u1 -- n2|u2
```
Subtract one from top-of stack.

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

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

```
CODE 2*         \ x1 -- x2
```
Signed multiply top of stack by 2.

```
CODE 4*         \ x1 -- x2
```
Signed multiply top of stack by 4.

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

```
CODE U2/        \ x1 -- x2
```

Unsigned divide top of stack by 2.

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

```
CODE 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 single precision integer numbers. N3|u3=n1|u1-n2|u2.

```
CODE NEGATE      \ n1 -- n2
```
Negate a single precision integer number.

```
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.

2.7 Multiplication

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

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

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

2.8 Division

Division on an ARM is really slow. Avoid it if you can where performance matters. Multiplication by the reciprocal and polynomial approximations are usually the recommended techniques. We have opted for performance over code size. The memory behaviour of many embedded ARM

devices compunds the impact of the additional test and branch overheads. Consequently, the test and subtract loops are unrolled. Unlike most other processors, signed division is faster than unsigned!

For more details of implementing division on an ARM, and for many other numerical coding techniques, see the "ARM System Developer's Guide", ISBN 1-55860-874-5

`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 920 bytes of code space.

`proc Udiv63/31 \ r0:r1/tos ; 64/32 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 division of N1 by N2 single-precision yielding remainder and quotient.

`CODE M/MOD \ d1 n2 -- rem quot`
A synonym for `FM/MOD` for Forth-83 compatibility. Obsolete - Use `FM/MOD` instead.

`: / \ 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.

2.9 Scaling - multiply then divide

These operations perform a multiply followed by a divide. The intermediate result is in an extended form. The point of this operation 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.

2.10 Stack manipulation

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

`CODE 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.

`: 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`.

`CODE 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`.

`CODE -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.

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

`CODE DUP \ x -- x x`

DUPlicate the top stack item.

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

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

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

```
CODE 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.

2.11 String and memory operators

```
CODE 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 but much larger version will be compiled. See *Arm\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 ON \ a-addr --`
Given the address of a CELL this will set its contents to TRUE (-1).

`CODE 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.

`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.

2.12 Miscellaneous words

`CODE NAME> \ nfa -- cfa`
Move a pointer from an NFA to the CFA or "XT" in ANS parlance.

`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 n -- 0|n true`
If the ASCII value *CHAR* can be treated as a digit for a number within the radix *N* then return the digit and true, otherwise return false.

2.13 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.

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

2.14 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.

2.15 Defining words and runtime support

`L: DOCREATE` `\ -- addr`
The run time action of `CREATE`. INTERNAL.

`CODE LIT` `\ -- x`
Code which when CALLED at runtime will return an inline cell value. 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. See the definition of `(.")` for an example. INTERNAL.

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

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

`: 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.

`: 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
```

`: DOCOLON,` `\ --`
Compile the runtime entry code required by colon definitions. INTERNAL.

`: !call` `\ dest addr --`
Install a BL DEST opcode at addr.

`: scall,` `\ addr --`
Compile a machine code BL to addr. No range checking is performed.

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

`: call,` `\ dest --`
Compile a BL or long CALL 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.

: : \ 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.

: 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.

: 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 DEFERed word. This will simply THROW a code back to the system.

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

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

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

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

2.16 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.

2.17 Branch constructors

Used when compiling code on the target.

`: c_branch<  \ addr --`
Lay the code for BRANCH. INTERNAL.

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

2.18 Main structure compilers

`: c_lit  \ lit --`
Compile the code for a LITERAL. 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 equate `FullCase?` is non-zero.

`: c_NEXTCASE  \ C: case-sys -- ; Run: x --`

Compile the code for `NEXTCASE`. INTERNAL. Only compiled if equate `FullCase?` is non-zero.

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

2.19 Miscellaneous

`code clz \ x -- #lz`
Count the number of leading zeros in x. The equate `CLZ?` must be set true to compile this word.

`32bit? [if] 32 [else] 16 [then] equ #bits \ -- u`
Number of bits in a cell.

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

3 Interrupt handlers

The file *ARM\IntARM5.fth* contains generic ARM interrupt handlers, plus alternative handlers for CPUs with vectored interrupt controllers. It replaces *ARM\IntARM3/4.fth* for new code with no changes to word usage. The main difference in *IntARM5.fth* is that R8 is assumed to be the floating point stack pointer. If you are using MPE-supplied VFP floating point or separate stack code, you should use *IntARM5.fth*. You are recommended to use *INTARM5.fth* for new code. In most cases, the only change required is to change the file name in your control file.

IntARM5.fth requires *ARM\ARMDEF.FTH* to be compiled before the *SFRxxxx* file for your particular CPU. The file *ARM/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 by the assembler routine `InitStacks` in *ARM\InitStacks.fth*. This routine can be referenced from your initialisation code.

The *IntARM5/4/3.fth* interrupt handlers are much more paranoid, and hence safer, than previous releases. The default diagnostic code gives much more information when an abort occurs. The files *INTARM.FTH*, *INTARM2.FTH*, *ARM\IntARM3.fth*, *INTS3C4510.FTH* and *INTAT91.FTH* remain in the distribution but are no longer supported.

Separate sections are provided for CPUs with vectored interrupt controllers, e.g. AT91 and Samsung, and these may have a different word set for initialising interrupts. Be careful with these to distinguish between Forth words (denoted by XTs) and the interrupt service routines (denoted by ISRs) that despatch the Forth words.

Each exception (interrupt) type requires a predefined stack frame on which the interrupt handler builds the Forth stacks and `USER` area. The system initialisation code must preset the banked R13 registers to the top of the frames. This code requires the Forth return stack pointer to be R13.

The IRQ 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 for each CPU mode. See *ARM\CONFIGS\LPC2106U.CTL* (contains stack layout) and *ARM\HARDWARE\INITLPC210X.FTH* (contains initialisation code) for examples.

From *IntARM4.fth* onwards, FIQ handlers take more advantage of the banked registers to speed up the FIQ entry and exit code. Nesting of FIQ interrupts is **not** supported.

In order to support exception nesting the equate `#IRQs` in the control file must be set to the maximum number of nestings required for ALL modes. When `#IRQs` is greater than 1 (to indicate IRQ nesting), the exception handler entry and exit code is extended to handle all exceptions in SVC mode. This avoids corruption of R14 (the LINK register) when nesting occurs. The equate #IRQs is used to calculate the size and layout of the IRQ and SVC mode stacks along with the equates `#SWIs` and `#FIQs` in the control file.

Note that each interrupt has its own `USER` area, but that NO USER VARIABLES ARE INITIALISED except for `S0` and `R0` in SWI handlers.

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

Note that it is implicit throughout the code that the four system registers UP, PSP, FSP, RSP are always set so that:

```
UP > PSP > FSP > 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.

3.1 Configuration

The floating point stack (FSP = ARM R8) will be set up if the equate `FP-SIZE` is non-zero.

```
0 equ FP-SIZE   \ -- 0
```

Set FP-SIZE to zero if undefined.
The following equates define values used for the equate `ARM-VIC?` that controls how IRQ and FIQ exceptions are processed. Note that if `ARM-VIC?` contains any other values, no IRQ or FIQ code will be compiled, permitting you to define different versions for other CPUs in another file. If you find and code other vectored interrupt controllers for ARMs, we will be happy to include your code in this file in future releases.

```
0 equ NO-VIC    \ -- n
```

specifies that no vectored interrupt controller is present.

```
1 equ AT91-VIC  \ -- n
```

specifies that the AT91 AIC vectored interrupt controller is present.

```
2 equ S3C-VIC   \ -- n
```

specifies that the Samsung vectored interrupt controller as used in the S3C4510B is present.

```
3 equ PL190-VIC \ -- n
```

Specifies that the ARM PL190 vectored interrupt controller as used by the Philips LPC210x is present.

```
4 equ AT91GIC-VIC       \ -- n
```

specifies that the AT91 GIC vectored interrupt controller as used in AT91SAMxxx devices is present.

```
5 equ PL192-VIC \ -- n
```

Specifies that the ARM PL192 vectored interrupt controller as used by the Philips LPC23xx/24xx is present.

```
NO-VIC equ ARM-VIC?     \ -- n
```

If the equate ARM-VIC? is undefined before this file is compiled, the generic (no vectored controller) will be compiled for FIQ and IRQ handling.
The following equates define what code is compiled. If they are defined before *IntARM4.fth* the default values below will be overridden.

```
1 equ test-isr? \ -- n ; non-zero to compile test code
```

Non-zero to compile test and diagnostic code.

```
0 equ SimpleAborts?     \ -- n
```

If this equate is non-zero, simple DAbort, PAbort, and Undef handlers are installed. The simple high level action receives only a pointer to the instruction that caused the abort and issues a simple message. If this equate is zero, a more complex DAbort handler is installed. The complex high level action receives a pointer to a status frame and a pointer to the instruction that caused the abort. The complex handlers are compiled if `TEST-ISR?` is nonzero. They perform a register dump and a return stack trace. The additional code space required for complex handlers is about 2300 bytes.

```
0 equ Reserved_ISR?     \ -- n
```
When non-zero, a handler will be installed for the RESERVED exception (vector at $0000:0014). This vector was used by the 26 bit architecture, but is now reserved for future expansion. It is used for other purposes by some CPUs, e.g. Philips LPC2xxx.

```
: !call         \ xt addr --
```
An `INTERPRETER` word to create a call opcode to xt at addr.

3.2 Interrupt management

```
proc initFIQ    \ --
```
Initialise FIQ banked registers. R0 contains the top of the FIQ stack frame. R0 and R5 must not be changed. Modifies RSP, PSP, UP and R4. This routine is called by `InitStacks`.

```
code EFI        \ -- ; enable FIQ interrupt
```
Global enable FIQ interrupt.

```
code DFI        \ -- ; disable interrupts
```
Global disable FIQ interrupt.

```
code EI         \ -- ; enable interrupts
```
Global enable IRQ.

```
code DI         \ -- ; disable interrupts
```
Global disable IRQ.

```
code [I         \ R: -- x ; preserve I/F status on return stack, disable ints
```
Preserve I/F status on return stack, disable IRQ. The state is restored by I].

```
code I]         \ R: x -- ; restore int. status from r. stack
```
Restore interrupt status saved by [I from the return stack.

3.3 Default Fault handlers

The interrupt structure of the ARM 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 *ARM/Drivers/serLPC210xqi.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.

In the (rare) case that SWIs are used, perform I/O and return, you must be careful with interrupt re-enabling if you want to use the interrupt driven I/O drivers inside the SWI handler. Your serial driver will also have to provide `-FaultConsole`.

```
: intoFault     \ --
```

Set up I/O for a fault handler.

3.4 SWI handler

The SWI handler is only compiled if the equate #SWIs is non-zero.

The SWI handler assumes that we may already be in supervisor mode, and that the RSP must be preserved. A new RSP stack is allocated, and the original RSP, R4..R12 and LINK are saved on the new RSP stack. New data stack and USER area are then allocated, parameters R0..R2 and the SWI# are put on the new data stack, and the handler is executed. On return, registers RSP and R3..12 are restored. R2 will be destroyed. Return data (if any) may be placed in R0 and R1.

By default, on entry to `SWI_HANDLER`, TOS contains the SWI#, and the next three items contain R0..2. This is done so that the standard ARM SWI calls can be emulated. Although it is not strictly necessary to remove this data, a canonical handler will have the stack effect:

```
  SWI_ACTION    \ r2 r1 r0 swi# -- r1' r0'
defer SWI_handler       \ r2 r1 r0 swi# -- r1' r0' ; default action
```

The default action is `2DROP`. Assign your own action to this word.

```
  assign mySwi to-do SWI_handler
PROC SWI_exception      \ --
```

The actual SWI handler which calls SWI_handler above. It assumes RSP=R13, R0-2 are parameters, R2 will be destroyed, Return parameters are in R0-R1, R3..R13 will be preserved

```
: testswi       \ r2 r1 r0 -- r1' r0' ; executes swi0
```

An example SWI handler.

```
code run-swi0   \ r2 r1 r0 -- r1 r0
```

Test execution of SWI 0.

3.5 Support for complex abort handlers

The complex exception handlers store the ARM CPU state in a data frame. The frame is 72 bytes long (18 4 byte cells), whose format is:

```
--- high ---
ABORT mode R14/link = faulting PC+8
faulting R12
faulting R11
faulting R10
faulting R9
faulting R8
faulting R7
faulting R6
faulting R5
faulting R4
faulting R3
faulting R2
faulting R1
faulting R0
faulting R14
faulting R13
faulting CPSR
ABORT mode CPSR on entry
--- low --
```

`struct /exframe \ -- n`
structure defining the exception frame

`: .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.

`: .frame \ ^frame --`
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.

`: .rsframe \ ^frame --`
Display the return stack indicated by the frame, assuming the Forth RSP=R13 and RUP=R11.

`: .psframe \ ^frame --`
Display the data stack indicated by the frame, assuming the Forth PSP=R12 and RUP=R11.

3.6 Undefined instruction handler

3.6.1 Simple UNDEF handler

```
defer Undef_handler     \ ^ins --
```

The place holder for the Undefined Instruction handler has a default action of DROP.

```
: testundef     \ ^ins -- ; handle undefined instruction
```

Test handler for Undefined Instructions.

```
: run-undef     \ --
```

Causes an Undefined Instruction exception.

3.6.2 Complex UNDEF handler

```
defer Undef_handler     \ ^frame ^ins -- fixed? ; true if instruction fixed up
```

Given a pointer to a data frame and a pointer to the instruction, returns true if a fixup has been performed. If the return value is non-zero, the instruction is rerun otherwise the instruction is skipped.

```
: testundef     \ ^frame ^ins -- fixed? ; handle undefined instruction
```

Test handler for Undefined Instructions.

```
: run-undef     \ --
```

Causes an Undefined Instruction exception.

3.7 Prefetch Abort handler

3.7.1 Simple PABORT handler

```
defer PAbort_handler    \ ^ins -- fixed?
```

Given a pointer to an instruction, returns true if a fixup has been performed. If the return value is non-zero, the instruction is rerun otherwise the instruction is skipped.

```
: testpabort    \ ^ins -- fixed?
```

Test PAbort handler.

```
: run-pabort    \ --
```

Run the test PAbort handler.

```
: run-pabort    \ --
```

Run the test PAbort handler.

3.7.2 Complex PAbort handler

```
defer PAbort_handler    \ ^frame ^ins -- fixed? ; true if instruction fixed up
```

Given a pointer to a data frame and a pointer to the instruction, returns true if a fixup has been performed. If the return value is non-zero, the instruction is rerun otherwise the instruction is skipped.

```
: testPabort    \ ^frame ^ins -- fixed?
```

Test PABORT handler.

```
: run-Pabort   $05000000 execute  ;     \ hardware specific
```

Run test PAbort handler.

3.8 Data Abort handler

3.8.1 Simple DAbort handler

```
defer DAbort_handler    \ ^ins -- fixed? ; true if instruction fixed up
```

Given a pointer to an instruction, returns true if a fixup has been performed. If the return value is non-zero, the instruction is rerun otherwise the instruction is skipped.

`: testdabort \ ^ins -- fixed?`
Test DABORT handler.

`: run-dabort $04000000 @ ; \ hardware specific`
Run test PAbort handler.

3.8.2 Complex DAbort handler

`defer DAbort_handler \ ^frame ^ins -- fixed? ; true if instruction fixed up`
Given a pointer to a data frame and a pointer to the instruction, returns true if a fixup has been performed. If the return value is non-zero, the instruction is rerun otherwise the instruction is skipped. The frame format is as for the `PAbort handler` above.

`: testdabort \ ^frame ^ins -- fixed?`
Test DABORT handler.

`: run-dabort $05000000 @ drop ; \ hardware specific`
Run test DAbort handler.

3.9 Reserved (26 bit address exception) handler

This exception is only used for systems and compilers which support the 26 bit PC mode of early ARM cores. The ARM/Cortex compilers do not support 26 bit mode, but ARM only compilers do.

`defer Unused_handler \ ^ins -- fixed?`
Given a pointer to an instruction, returns true if a fixup has been performed. If the return value is non-zero, the instruction is rerun otherwise the instruction is skipped.

`PROC Unused_exception \ --`
The UNUSED exception handler calls UNUSED_handler above.

`: testunused \ --`
Test code for the UNUSED exception.

3.10 Generic IRQ handler

The ARM architecture does not define a vectored interrupt controller, although several CPUs provide one. If the CPU has a vectored controller, it should be used. This implementation provides a linked list of routines to be called, each of which tests for its own interrupt and then handles it if required. The last item in the chain is the return from IRQ routine. The variable `NEXT-IRQ` points to the first ISR to execute.

Note that this routine must be modified to support nested interrupts. Because the IRQ mode link register (R11) is set by an interrupt response, as well as by the BL instruction, if an interrupt is accepted after a BL instruction but before R11 is saved, R11 will be corrupted. The result of this is that nestable interrupt handlers must switch to another mode (e.g. SVC) before re-enabling interrupts. This is handled by code in the `IRQ_EXCEPTION` and `IRQ_RETI` routines below.

The required modification is performed if the EQUate `#IRQs` is greater than one, whereupon IRQ nesting is supported with the penalty of greater overhead. IF IRQ nesting is required,

the IRQ stack size should be at least 64*#IRQs bytes, and TASK-SIZE*#IRQs bytes should be added to the SWI stack. If IRQ nesting is not required the IRQ stack must be at least `TASK-SIZE` bytes.

`variable next-irq \ -- addr`
Holds pointer to next ISR to run. ISRs are added to this chain.

`PROC IRQ_exception \ -- ; IRQ entry code`
The IRQ handler entry point which executes the chain. The USER variables `S0` and `R0` are initialised.

`proc IRQ_reti \ -- ; IRQ exit code`
This code cleans up after the IRQ and is the first item added to the IRQ chain.

`: br24, \ xt opcode --`
Given an opcode (B or BL) and an execution address, compiles a branch or call to it.

`: add-isr \ xt chain -- ; ’ <name> <chain_var> ADD-ISR adds name to the ISR handler li`
An action is added to the IRQ or FIQ chain by a phrase of the form:

```
’ <name> <chain_var> ADD-ISR
```

`: add-irq \ xt -- ; ’ <name> ADD-IRQ adds name to the IRQ handler list`
An action is added to the IRQ chain by a phrase of the form:

```
’ <name> ADD-IRQ
```

3.11 Generic FIQ handler

Note that FIQ interrupt nesting is NOT supported by default. If it is required, use the Generic IRQ handler as a model, and do not forget to switch back to FIQ mode rather than IRQ mode.

`variable next-fiq \ -- addr`
Holds pointer to next ISR to run. FIQ ISRs are added to this chain

`PROC FIQ_exception \ --`
FIQ entry code.

`proc FIQ_reti \ --`
FIQ exit code.

`: add-fiq \ xt -- ; adds action to FIQ handler list`
An action is added to the FIQ chain by a phrase of the form:

```
’ <name> ADD-FIQ
```

3.12 AT91 IRQ and FIQ handlers

This section is not a treatise on the Atmel Advanced Interrupt Controller. These words provide a fairly basic set of tools for using the AIC. The notes in the Generic IRQ section about interrupt nesting apply here.

This code requires *Kernel62.fth* and the equate `COLDCHAIN?` must be set non-zero in the control file.

Basic clock enabling of the AIC should be performed in the CPU specific startup file, e.g. *ARM\HARDWARE\EB55\INITARM55800.FTH.*

If the EQUate `#IRQs` is greater than one, IRQ nesting is supported with the penalty of greater overhead. IF IRQ nesting is required, the IRQ stack size should be at least 64*#IRQs bytes, and TASK-SIZE*#IRQs bytes should be added to the SWI stack. If IRQ nesting is not required the IRQ stack must be at least `TASK-SIZE` bytes.

`PROC IRQ_entry  \ --`
This is the template code for vectored IRQ interrupt handlers. It is also used as the spurious interrupt handler.

`PROC FIQ_entry  \ --`
This is the template code for vectored FIQ interrupt handlers.

`: IRQ:          \ xt "<name>" -- ; -- isr`
Creates an IRQ ISR that runs the given Forth word. At run time the entry point of the ISR is returned. Use in the form:

```
' <action> IRQ: <actionISR>
```

`: FIQ:  \ xt "<name>" -- ; -- isr`
Creates an FIQ ISR that runs the given Forth word. At run time the address of the ISR is returned. Use in the form:

```
' <action> FIQ: <actionISR>
```

`: SetDefIRQ     \ xt --`
Set the spurious IRQ handler to call the Forth word whose xt is given. Use interpretively in the form:

```
' <word> SetDefIRQ
```

The compiler sets this action to NOOP unless you use `SetDefIRQ`.

`: InitSPU       \ --`
Initialise the spurious interrupt vector. This word is executed as part of the cold chain.

`: EnInt         \ int# --`
Enable the requested AIC interrupt.

`: DisInt        \ int# --`
Disable the requested AIC interrupt.

`: SetIRQisr     \ isr mode int# --`
Set the ISR to be the action of IRQ INT# with mode being set into the relevant AIC SMR register. Then enable the interrupt. The FIQ interrupt is set with INT#=0 for which the priority is unused. Note that Forth IRQ interrupt handlers must be created with `IRQ:` and Forth FIQ handlers with `FIQ:`. Assembler routines may be created using:

`PROC <name> ... END-CODE`

Interrupt numbers may be found in the AIC section of the CPU data sheet.

The mode value is a combination of priority 0..7 and interrupt type as follows. Only the first two should be used for internal interrupts.

`$000 equ ISR_low        \ -- n`
Low level sensitive.

`$020 equ ISR_nedge      \ -- n`

Negative edge triggered.

`$040 equ ISR_high \ -- n`
High level sensitive.

`$080 equ ISR_pedge \ -- n`
Positive edge triggered.
An example of setting up an interrupt follows.

```
: ISRaction  \ -- ; the Forth word
  ...
;
' ISRaction IRQ: MyIsr  \ -- isr
#15 equ MyIsr#  \ -- n

: SetupISR   \ -- ; initialise
  MyIsr ISR_low 7 or MyIsr#
;
```

3.13 Samsung S3C4510 IRQ and FIQ handlers

This implementation does NOT support nested FIQ interrupts. The notes in the Generic IRQ section about interrupt nesting apply here.

If the EQUate `#IRQs` is greater than one, IRQ nesting is supported with the penalty of greater overhead. IF IRQ nesting is required, the IRQ stack size should be at least 64*#IRQs bytes, and TASK-SIZE*#IRQs bytes should be added to the SWI stack. If IRQ nesting is not required the IRQ stack must be at least `TASK-SIZE` bytes.

`create despatch_table  \ -- addr ; 21 entry interrupt despatch table`
This table holds the addresses (XTs) of the service routines for each of the 20 interrupts plus a dummy for false interrupts. Equates for the interrupt numbers and their bit positions may be found in the file *SFRS3C4510.FTH*. This table is defined in the CDATA section because it is assumed that the code is run from RAM. If this is not so, change the `CDATA` above the definition as required, usually to `IDATA`.

`: SetFIQ \ xt int# --`
Set the XT to be the action of FIQ INT#.

`: SetIRQ \ xt int# --`
Set the XT to be the action of IRQ INT#.

`: EnInt \ int# --`
Enable the requested interrupt.

`: DisInt \ int# --`
Disable the requested interrupt.

3.14 ARM PL190 IRQ and FIQ handlers

This section is not a treatise on the ARM PL190 Vectored Interrupt Controller. These words provide a fairly basic set of tools for using the ARM PL190 VIC, which is fully documented in the ARM Technical Publications CD available free of charge from *www.arm.com*. The notes in the Generic IRQ section about interrupt nesting apply here.

This code requires *Kernel62.fth* and the equate `COLDCHAIN?` must be set non-zero in the control file. The VIC addresses should be defined in the *SFRxxxx.fth* file which defines the peripheral address for the silicon.

If the EQUate `#IRQs` is greater than one, IRQ nesting is supported with the penalty of greater overhead. IF IRQ nesting is required, the IRQ stack size should be at least 64*#IRQs bytes, and TASK-SIZE*#IRQs bytes should be added to the SWI stack. If IRQ nesting is not required the IRQ stack must be at least `TASK-SIZE` bytes.

FIQ nesting is not supported by this code. The FIQ stack must be at least `TASK-SIZE` bytes.

`1 equ #VICs \ -- n`
If this equate has not already been set, a value of 1 is used. If more than one is defined, the base addresses must be named `_VICn`, and the primary VIC must also be named `_VIC`.

`PROC IRQ_entry \ --`
This is the template code for vectored IRQ interrupt handlers. It is also used as the default interrupt handler.

`PROC FIQ_entry \ --`
This is the wrapper code for FIQ interrupt handlers.

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

```
' <action> IRQ: <actionISR>
```

`: SetDefIRQ \ xt --`
Set the default IRQ handler to call the Forth word whose xt is given. Use interpretively in the form:

```
' <word> SetDefIRQ
```

The compiler sets this action to NOOP unless you use `SETDEFIRQ`.

`: SetFIQ \ xt --`
Set the FIQ interrupt to call the Forth word whose xt is given. Use interpretively in the form:

```
' <word> SetFIQ
```

`: InitVIC \ --`
Initialise the VIC Default interrupt vector. This word is executed as part of the cold chain.

`: EnInt \ src# --`
Enable the requested VIC interrupt source.

`: DisInt \ src# --`
Disable the requested VIC interrupt source.

`: SetIrqIsr \ isr src# slot# --`
Set the ISR to be the action of the source interrupt number src# (0..31) being set into the corresponding slot (slot# = 0..15) where slot# corresponds to priority (0 = highest priority). Then enable the interrupt. Note that Forth IRQ interrupt handlers must be created with IRQ: <name>. Assembler routines may be created using:

`PROC <name> ... END-CODE`

Interrupt source numbers may be found in the VIC section of the CPU data sheet and the relevant *SFRxxxx.FTH* file.

`: setFIQsrc  \ src# --`
Set *src#* to be an enabled FIQ interrupt. During cross compilation the ISR address must have been set with `SetFIQ` above. The FIQ interrupt can be enabled and disabled by `EnInt` and `DisInt`.
An example of setting up an interrupt follows.

```
: ISRaction  \ -- ; the Forth word
  ...
;
' ISRaction IRQ: MyIsr  \ -- isr
#15 equ MySrc#  \ -- n ; what triggers it
#3 equ MySlot#  \ -- n ; what priority (0=highest)

: SetupISR   \ -- ; initialise
  MyIsr MySrc# MySlot# SetIRQisr
;
```

`: .VIC  \ --`
Show status of the VIC.

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

3.15 ARM PL192 IRQ and FIQ handlers

This section is not a treatise on the ARM PL192 Vectored Interrupt Controller. These words provide a fairly basic set of tools for using the ARM PL192 VIC, which is fully documented in the ARM Technical Publications CD available free of charge from *www.arm.com*. The notes in the Generic IRQ section about interrupt nesting apply here.

This code requires *Kernel62.fth* and the equate `COLDCHAIN?` must be set non-zero in the control file. The VIC addresses should be defined in the *SFRxxxx.fth* file which defines the peripheral address for the silicon.

If the EQUate `#IRQs` is greater than one, IRQ nesting is supported with the penalty of greater overhead. If IRQ nesting is required, the IRQ stack size should be at least 64*#IRQs bytes, and TASK-SIZE*#IRQs bytes should be added to the SWI stack. If IRQ nesting is not required the IRQ stack must be at least `TASK-SIZE` bytes.

FIQ nesting is not supported by this code. The FIQ stack must be at least `TASK-SIZE` bytes.

`1 equ #VICs  \ -- n`
If this equate has not already been set, a value of 1 is used. If more than one is defined, the base addresses must be named `_VICn`, and the primary VIC must also be named `_VIC`.

`PROC IRQ_entry  \ --`
This is the template code for vectored IRQ interrupt handlers.

`PROC FIQ_entry  \ --`

This is the template code for FIQ interrupt handlers.

`: IRQ:  \ xt -- ; -- isr`

Creates an IRQ ISR that runs the given Forth word. At run time the entry point of the ISR is returned. Use in the form:

```
' <action> IRQ: <actionISR>
```

`: SetFIQ  \ xt --`

Set the FIQ interrupt to call the Forth word whose xt is given. Use interpretively in the form:

```
' <word> SetFIQ
```

`: EnInt  \ src# --`

Enable the requested VIC interrupt source.

`: DisInt  \ src# --`

Disable the requested VIC interrupt source.

`: SetIrqIsr  \ isr src# prio# --`

Set the ISR to be the action of the source interrupt number src# (0..31) with priority prio# (0..15, 0 = highest priority) Then enable the interrupt. Note that Forth IRQ interrupt handlers must be created with IRQ: <name>. Assembler routines may be created using:

`PROC <name> ... END-CODE`

Interrupt source numbers may be found in the VIC section of the CPU data sheet and the relevant *SFRxxxx.FTH* file.

`: setFIQsrc  \ src# --`

Set *src#* to be an enabled FIQ interrupt. During cross compilation the ISR address must have been set with `SetFIQ` above. The FIQ interrupt can be enabled and disabled by `EnInt` and `DisInt`.
An example of setting up an interrupt follows.

```
: ISRaction  \ -- ; the Forth word
  ...
;
' ISRaction IRQ: MyIsr  \ -- isr
#15 equ MySrc#  \ -- n ; what triggers it
#3 equ MySlot#  \ -- n ; what priority (0=highest)

: SetupISR   \ -- ; initialise
  MyIsr MySrc# MySlot# SetIRQisr
;
```

`: .VIC  \ --`

Show status of the VIC.

`: SWINT  \ int# --`

Generate interrupt from software.

3.16 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.

```
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.

3.16.1 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.

3.16.2 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.

4 ARM multitasker

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

The code in *MultiARM.fth* only works with the ARM32 register set usage. If you need interworking with one of the Cortex register set usages, please contact MPE or modify the code yourself.

4.1 Configuration - normally performed earlier

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

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

4.2 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.

4.3 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.

```
code status     \ -- task-status
```

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

`CODE restart  \ task -- ; mark task TCB as running`
Sets the RUN bit in the task's status cell.

`CODE 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`.

4.4 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.

4.5 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.

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

```
: terminate     \ task --
```
Stop a task, and remove it from the list.

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

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

4.7 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 SIGNALs 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> !
```

```
: 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.

```
: 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.

4.8 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:`.

4.9 Debugging tools

`: .taskBody \ addr --`
Display the name of a task, given the address of the link.

`: .taskBody \ addr --`
Display the name of a task, given the address of the link.

`: .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.

5 SA-1110 Cache

For CPUs that have a cache, the word FLUSHCACHE should be provided, which should write back all modified data and invalidate at least the instruction cache. Newly compiled code was written to data space and the data cache. Invalidating the ICACHE causes the newly compiled code to be executed rather than the previous contents of the instruction cache, which may contain data now modified by the newly compiled code.

For the SA-1110, see the Intel and the ARM Architecture Reference Manual, 2nd edition, ed. David Seal, chapters B2, B3 and B5.

The source code is in the file ARM\CACHESA1110.FTH.

The equate TBASE defines the start address of the MMU tables and must be defined before this file is compiled, and must be on a 64k boundary. Note that this set up performs no memory remapping, and that the adress and size of the DRAM area are set in this file. If you want to use this file with multiple cacheable sections or with other adress assignments, you will have to modify this file.

```
$C0000000 equ DRAM-START                \ start of DRAM
```
Defines the start address of the DRAM which is to be cached.

```
4 Mb      equ /DRAM                     \ size of DRAM
```
Defines the size of the DRAM to be cached.

```
DRAM-START /DRAM + equ DRAM-END         \ end of DRAM + 1
```
Defines the end address+1 of cacheable DRAM.

```
variable CE?    \ -- addr ; holds true if cache enabled
```
This variable holds -1 when the cache has been enabled, or 0 when it is disabled.

```
code EnableCache        \ -- ; Switch on cache & MMU for DRAM
```
Enable the data cache by constructing a default MMU table and enabling it. If variable CE? is already set, no action is taken.

```
code FlushCache \ -- ; flush DCache and ICache
```
Flush the ICache and DCache. See the the SA-1110 Developer Manual section 5.2.3. If the variable CE? is 0, no action is taken.

```
code DisableCache       \ -- ; Switch off DATA cache & MMU
```
Disable the data cache.

6 Minimal Umbilical code definitions

The file ARM\MINARM.FTH contains the minimum code definitions required to support Umbilical Forth. If additional definitions are required, they may be copied to a new file from ARM\CODEARM.FTH or COMMON\KERNEL62.FTH.

6.1 Flow of control

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

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

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

6.2 Stack operations and maths

```
CODE NOOP       \ --
```
A NOOP, null instruction.)

```
CODE DROP       \ x -- ; needed by default interrupt handler
```
Lose the top data stack item and promote NOS to TOS.

```
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?. Return TRUE if N1 is within the range N2..N3-1. 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.

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

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

```
CODE *          \ n1 n2 -- n3
```

Standard signed multiply. N3 = n1 * n2.

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

```
CODE UM/MOD      \ ud un -- urem uquot
```
Perform unsigned division of double number UD by single number U and return remainder and quotient.

```
: 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.

```
CODE FM/MOD      \  d1 n2 -- rem quot ; symmetric 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 division of N1 by N2 single-precision yielding 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.

```
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 -- d2
```
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.

6.3 Strings

```
CODE CMOVE      \ c-addr1 c-addr2 u --
```

Copy U bytes of memory forwards from C-ADDR1 to C-ADDR2.

```
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 -- ; used by multitasker
```

Fill U bytes of memory starting at C-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. 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".

6.4 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.

6.5 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.

7 ARM specific library code

The code in ARM\LIBARM.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%\libARM
  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.

7.1 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.

7.2 interrupt enable and disable

`code EI         \ -- ; enable interrupts`
Global enable interrupts, both FIQ and IRQ.

`code DI         \ -- ; disable interrupts`
Global disable interrupts, both FIQ and IRQ.

`code SAVE-INT   \ -- x ; return interrupt status and disable interrupts`
Geturn interrupt status and then disable interrupts. Use [I and I] for new code.

`code RESTORE-INT        \ x -- ; restore state returned by SAVE-INT`
Restore state returned by SAVE-INT. Use [I and I] for new code.

`code [I         \ R: -- x ; preserve I/F status on return stack, disable ints`
Preserve I/F status on return stack, disable interrupts. The state is restored by I].

`code I]         \ R: x -- ; restore int. status from r. stack`
Restore interrupt status saved by [I from the return stack.

7.3 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.

8 Philips LPC2xxx 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.

The LPC210x CPUs have 128kb Flash in 8kb sectors numbered from 0..15. These sectors may be programmed in units of 512 bytes. Sector 15 is the boot sector which cannot be erased or programmed. See the LPC2106 User Manual for more details.

```
1 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.

```
3 cells buffer: IAPres  \ -- addr ; max 3 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 Philips part number.

8.1 Gotchas

If you have problems with the IAP routines, check the bootloader version using the Philips ISP software or by typing

```
IAPBootVer .dword
```

which will give something of the form:

```
0000:xxyy
```

xx is the major version number and yy is the minor version number. If this number is less than 0000:0134 (hexadecimal) or 1.52 (decimal) you should update the bootloader using ISP software version 2.2.0 or greater. These are available on the MPE CDs and from

```
www.semiconductors.philips.com
  /files/products/standard/microcontrollers/utilities/
    philips_flash_utility.zip
    lpc2000_bl_update.zip
```

If you still have problems, use the Philips ISP software to erase the whole of the Flash, and then reinstall an appropriate .HEX file. Remember to convert the latest .IMG file to .HEX.

9 LPC2000 Flash tools

These tools are provided for applications which reserve part of the Flash for data. This code uses the IAP routines in *IAP210x.fth*.

N.B. An erase causes the whole of the sector at 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.

9.1 Flash primitives

Although some of these routines are not the most efficient for the LPC2000 series, they are designed to be easily expanded for future LPC2xxx 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.

9.2 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. Increment error counter on error.

`: (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.

```
: ProgFlashQuiet        \ src dest len --
```

As `ProgramFlash`, but with no console output.

10 Reprogramming the LPC2xxx serially

10.1 Introduction

REPROG210x.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 and the two 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.

```
#2138 constant LPCtype  \ -- u
```
Define default LPC type if none defined on command line

10.2 Configuration

```
LPCtype equ LPCtype     \ -- n
```
Define the LPC type being used. This is used to select CPU specific data and routines.

```
#10000000 equ xtal-speed        \ -- Hz
```
Crystal clock rate in HZ. For greatest baud rate flexibility use 14.7456 MHz

```
#60000000 equ system-speed      \ -- Hz
```
System operating speed in HZ. Must be the same as, or an even multiple of `XTAL-SPEED`. If not the same, the initialisation code will set up the PLL as required.

```
0 equ FullIAP?          \ -- n
```
Set this equate non-zero to compile additional IAP routines. See *IAP210x.FTH* for more details.

10.3 Items of interest

```
l: CLD1         \ -- addr
```
Filled in later with the xt of `ReprogFlash`

```
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 an application to select which UART to use.

```
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.

```
make-turnkey ReprogFlash                \ start up action
```

Define the action of the reprogramming code.

```
' CopyFlash CLD_CopyFlash !
```

Define the action of the Flash copy code.

11 Philips LPC2xxx Reflashing

11.1 Introduction

If you destroy the application in the internal Flash, you must use the Philips ISP loader to reload an Intel Hex file supplied on the ARM Stamp CD. A suitable baud rate is 38400 baud. To use this, ensure that port P0.14 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 P0.14 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 LPC2xxx 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.

11.2 Code in main application

The layout of the RAM code is defined in the source file *InitRp210x.fth*. This defines the first cell as the Forth word to execute.

```
create reprog2xxx.img   \ -- addr
```
The start address of the reprogramming code

```
  data-file %HwDir%/ReProg/REPROG210X.IMG equ /reprog   \ -- len
```
The length of the reprogramming code loaded into the dictionary.

```
  data-file %HwDir%/ReProg/REPROG2134.IMG equ /reprog   \ -- len
```
The length of the reprogramming code loaded into the dictionary.

```
  data-file %HwDir%/ReProg/REPROG213X.IMG equ /reprog   \ -- len
```
The length of the reprogramming code loaded into the dictionary.

```
$40000200 equ reprogrun \ -- addr
```
The run time address of the reprogramming code. This **must** match the start address of the `CDATA` section defined in *REPROG210x.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 LPC2xxx UART to use. N.B. No effect on the MPE USB Stamp.

```
: 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.

12 LPC2xxx GPIO utilities

The code in *ARM\LPC-GPIO.fth* provides utility words for accessing the GPIO/FIO pins on a bit by bit basis. The code is written for ease of use rather than performance.

12.1 Configuration

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

```
1 equ useFIOs?  \ -- flag
```

Use the FIO access reather than the legacy GPIO access. Note that on the Cortex-M3 devices, legacy access is not supported. FIO (Fast I/O) is **much** faster. Unless you are using LPC21xx/22xx devices before the /O1 releases, leave `useFIOs` set to 1.

```
: initFIOs      \ --
```

To use the FIOs on LPC21xx and LPC22xx parts, SCS bits 0 and 1 must be set. This is done at power up.

```
: initFIOs      \ --
```

To use the FIOs on LPC23xx and LPC24xx parts, SCS bit 0 must be set. This is done at power up.

```
_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`.

```
FIOSET equ xIOSET       \ -- u
```

Offset to port set register

```
FIOCLR equ xIOCLR       \ -- u
```

Offset to port clear register

```
FIOPIN equ xIOPIN       \ -- u
```

Offset to port pin register

```
FIODIR equ xIODIR       \ -- u
```

Offset to port pin register

12.2 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
```

12.3 IO pin access

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

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

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

12.4 IO pin configuration

```
: 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.

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

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

12.5 Test code for LPC-E2468

```
decimal

4 16 pio: led2
4 17 pio: led1

: 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
;
```

12.6 Test code for LPC-P2148

```
decimal

0 15 pio: but1
0 16 pio: but2

\ but1 getPin .
\ but2 getPin .
```

Index

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

V

W

X