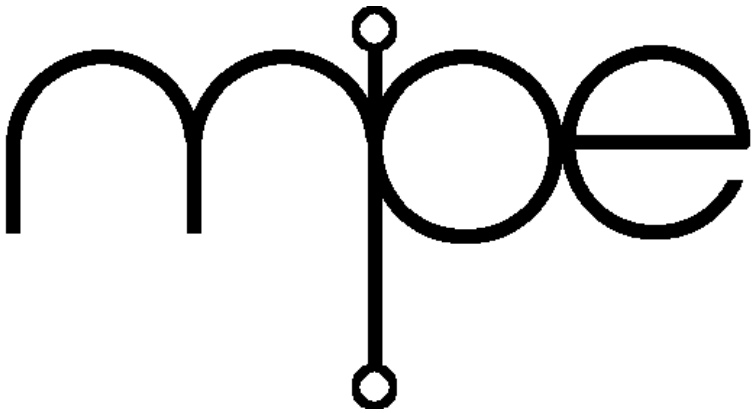
Cortex Lite Target Code

v2.0

Microprocessor Engineering Limited



Published by Microprocessor Engineering

Cortex Lite Target Code v2.0
User manual
Manual revision 2.0
28 December 2015

Software
Software version 2.0

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 Lite version licence terms

1.1 Compiler

You may not redistribute any portion of the compiler or Lite system distribution without written permission from MicroProcessor Engineering Ltd (MPE). The distribution should be downloaded as a whole from the MPE web site.

The compiler is licensed for non-commercial uses only. For example, you may not sell a product that contains code generated with the Lite compiler. If your job or payment depends on use of the Lite compiler, that is a commercial use.

If you think that you are a special case, e.g. you want to use the compiler in a school, college or university class, just ask us.

If you are not a special case, use the Lite compiler for evaluation and then buy a Stamp, Standard or Professional version of the compiler to acquire many more facilities and a commercial-use licence.

1.2 Distribution of application programs

Applications compiled with the MPE Lite compiler may be distributed free-of-charge. The MPE sign-on message must be preserved and a link to the MPE website must be provided. No part of the cross-compiler or the target source code may be further distributed except as detailed above.

1.3 Warranties and support

We try to make our products as reliable and bug free as we possibly can. We support our products. If you find a bug in this product and its associated programs we will do our best to fix it. Please check first by email to `tech-support@mpeforth.com` to see if the problem has already been fixed. Please send us enough information including source code on disc or by email to us, so that we can replicate the problem and then fix it. Please let us know the version/build number of your system.

Technical support will only be available on the current version of the product.

2 Introduction

This manual documents the MPE Forth Lite kernel for ARM Cortex processors. The Lite kernel is not the same as the PowerForth kernel supplied with the Professional, Standard and Stamp compilers. The Lite kernel is smaller and compiles directly to Flash, which has required changes to the kernel. These changes are discussed later.

2.1 Supported boards

Supported boards are the ST STM32F072B Discovery, ST STM32F4 Discovery (with the STM32F407 chip), the Freescale Freedom KL25Z, the NXP LPC1114FN28, and the Infineon XMC2Go. Preliminary releases for other boards may also be included.

The following discuss getting a cross-compiled binary image into the boards.

2.1.1 STM32F072B Discovery board

The MPE Forth kernel uses the hardware USART1 for serial communications to talk to the Forth kernel on the board. We use an FTDI USB to 3v3 TTL serial cable for this:

`http://www.ftdichip.com/Products/Cables/RPi.htm`

Connect the cable to the Discovery board:

- Black - Gnd,
- Yellow - PA9 - PC Rx, STM32 USART1 Tx,
- Orange - PA10 - PC Tx, STM32 USART1 Rx.

Set your terminal emulator, e.g. PowerTerm in AIDE, to 115200 baud, 8 data bits, no parity.

Power the board using the central (ST Link v2) USB connector.

After cross compiling the control file *Cortex/Hardware/STM32F072Bdisco/liteSTM32F072sa.ctl* the image file will be *Cortex/Hardware/STM32F072Bdisco/liteSTM32F072sa.hex*

To program the Flash on the Discovery board for the first time, use the STM32 ST-LINK utility. See:

`http://www.st.com/web/en/catalog/tools/PF258168`

2.1.2 STM32F4 Discovery board

`http://www.ftdichip.com/Products/Cables/RPi.htm`

Connect the cable to the Discovery board:

- Black - Gnd,
- Yellow - PD8 - PC Rx, STM32F4 USART3 Tx,
- Orange - PD9 - PC Tx, STM32F4 USART3 Rx.

Set your terminal emulator, e.g. PowerTerm in AIDE, to 115200 baud, 8 data bits, no parity.

Power the board using the ST Link v2 USB connector.

After cross compiling the control file *Cortex/Hardware/STM32F407disco/liteSTM32F407discoSA.ctl* the image file will be *Cortex/Hardware/STM32F407disco/liteSTM32F072.hex*.

To program the Flash on the Discovery board for the first time, use the STM32 ST-LINK utility. See:

```
http://www.st.com/web/en/catalog/tools/PF258168
```

2.1.3 Freescale Freedom FRDM-KL25Z board

The Forth uses UART0 on the FRDM-KL25Z board connected to the on-board serial to USB adapter. The board presents itself as a USB memory stick and a comms port. To use these, download the latest OpenSDA firmware drivers from

```
http://www.pemicro.com/opensda/
```

and install them. After installation on Windows, you will see the memory stick as a drive, and a new comms port will be available to AIDE.

When you put a new binary file into the new drive, the board's second processor checks the file and Flashes it into the target. The compiler uses this facility. It is automated by adding the following two lines before `FINIS` in the control file:

```
afterwards  fdel F:\FRDMKL25Zsa.bin
afterwards  fcopy FRDMKL25Zsa.bin F:\FRDMKL25Zsa.bin
```

Change the drive letter as required.

After cross compiling the control file *Cortex/Hardware/KinetisKL20/liteFRDMKL25Zsa.ctl* copy the binary file *Cortex/Hardware/KinetisKL20/FRDMKL25Zsa.bin* to the Flash drive. The board will reboot and the new Forth file will be running. By default, the Forth Lite kernel continuously changes the colour of the tri-colour LED.

The terminal emulator should be set to 115200 baud. After the file has been copied to the Flash drive, it is programmed into the target. When the target reboots, the Forth prompt appears.

2.1.4 LPC1114FN28

This chip is a Cortex-M0 in a DIP28 package with 32 kb of Flash and 4 kb of RAM. The chip runs at 48 MHz from the internal oscllator. Comms are handled by UART0 at 115200 baud, connected to RXD (PIO1.6) and TXD (PIO1.7).

Providing that your hardware has buttons for both the reset pin and the boot pin, you can use FlashMagic to program the Flash. See:

```
http://www.flashmagictool.com/
```

2.1.5 Infineon XMC2Go

This tiny board includes an XMC1100-0064 CPU with 64k of Flash and 16 kb RAM, plus an

on-board J-Link device with a virtual COM port. The MPE cross compiler supports the J-Link directly and programs the Flash at the end of compilation.

You must install the Segger J-Link tools from

```
https://www.segger.com/jlink-software.html
```

and then copy the file *JLinkARM.dll* into the directory containing the cross compiler. You must use the Segger tools of version 5.02h or later.

2.1.6 STM32F031K6 Nucleo-32 board

The MPE Forth kernel uses the hardware USART1 for serial communications to talk to the Forth kernel on the board through the on-board ST-Link v2.1 which provides a USB virtual serial port.

Set your terminal emulator, e.g. PowerTerm in AIDE, to 115200 baud, 8 data bits, no parity.

Power the board using the central (ST Link v2.1) USB connector.

After cross compiling the control file *Cortex/Hardware/Nucleo-F031K6/liteSTM32F031NucleoSA.ctl* the image file will be *Cortex/Hardware/Nucleo-F031K6/LITESTM32F031SA.hex*

To program the Flash on the Discovery board for the first time, use the STM32 ST-LINK utility. See:

```
http://www.st.com/web/en/catalog/tools/PF258168
```

2.1.7 STP LPC812 board

This is a minimalist board with 16 kb of Flash. It is used with the FlashMagic utility in a conventional cross-compile, download to Flash, and then test cycle. It **is** possible to squeeze the Flash programming code onto the board, but that does not leave enough space for a real application.

The board's UART0 is connected to an FTDI serial to USB converter. The kernel runs the UART at 115200 baud. Most versions of Windows include a driver for the FTDI adapters.

After cross compiling the control file *Cortex/Hardware/LPC81x/liteSTP812sa.ctl* the image file will be *Cortex/Hardware/LPC81x/liteSTP812sa.hex*

To program the Flash on board, move the jumper by the reset button so that both pins are covered. Press the reset button and then use use the FlashMagic utility. See:

```
http://www.flashmagictool.com/
```

After the Flash has been programmed, move the jumper so that one pin is uncovered; it is convenient to leave the jumper on the other pin. The press the reset button to start the Forth application.

2.2 Producing the kernel

The Forth kernel is built using a cross compiler running in the AIDE environment. AIDE contains three primary windows: compiler, terminal emulator and text editor. The compilation results can be seen in the compiler window. During compilation, the compilation results can be seen in the compiler/tool window. The process below is very similar for all boards except for the use of vendor-specific tools.

To compile the standalone Forth kernel, find the button for your board, e.g. "STM32F072 Lite SA", in AIDE's main toolbar and click it. After successful compilation, program the Flash with the file *LITESTM32F072SA.hex* using the STM32 ST-LINK utility. See:

```
http://www.st.com/web/en/catalog/tools/PF258168
```

Other boards will have other utilities to Flash the board.

After programming the board, the new kernel will be running. To talk to the kernel, use AIDE's PowerTerm terminal emulator. The kernel talks to PowerTerm through the FTDI cable. Finding this UART can require a little botheration. There are two ways to do this.

1. Go to Windows Control Panel -> System -> Device Manager -> Ports. One of the entries will be something like: **USB Serial Port (COM14)**. COM14 will be the UART you will use.
2. Go to AIDE's PowerTerm configuration dialog. Select the drop down list in the COM Port# group. It will list all the available COM ports by COM number and name. The one you want includes: **COMxx USB Serial Port**.

Now select the required port in the PowerTerm configuration dialog and ensure that it is set to 115200 baud, 8 data bits, 1 stop bit, no parity and file server is enabled. Save the configuration and press OK. Start the PowerTerm connection and you should be talking to the kernel.

Once you have the Forth kernel running on the target, you can use Forth's internal flash utility to flash a newly compiled Forth image. Type:

```
REFLASH
```

The Flash will be erased, and AIDE will present you with a file dialog box. Find the file *LITESTM32F072SA.img* (**not** the .hex file)), select it and the file will be downloaded and flashed. If everything goes wrong, just start again with the Flash utility.

2.3 About the kernel

The Forth kernel you have made and installed is a cut-down version of the full PowerForth kernel. It has extensions so that code is compiled directly into Flash memory. When you want to add code, you can compile it directly on the Forth kernel by using the phrase

```
include <filename>
```

on the target Forth command line. AIDE will then try to deliver the file to the kernel. If the file cannot be found, AIDE will let you change the default directory (the commonest problem) or correct the spelling.

To start over just type `EMPTY`, the application region of the Flash will be cleaned, and the CPU

will be rebooted. To reuse your compiled code at the next reboot or power up, use the word `COMMIT`. If you just want to preserve the code, use:

```
0 COMMIT
```

If you want the Forth to run a word, say `APP`, use:

```
' APP COMMIT
```

To support compilation to Flash, the kernel uses a compilation buffer (usually 512 bytes) in RAM. The use of this buffer is mostly transparent to the user.

2.4 Gotchas

2.4.1 Flash problems

If you forget to use `COMMIT` and `EMPTY` appropriately, the Flash may not be correctly erased and application compilation may fail. Try to use `EMPTY` and if that fails, reinstall a new kernel.

2.4.2 Flash kernel is non-standard

The Lite kernel is not completely to the ANS and Forth-2012 standards.

A particular change is that the Forth word `IMMEDIATE` is not present. Instead, precede the the word with `IMM` and will be immediate. This change is a result of the particulars of Flash implementations and their controllers.

2.4.3 Building tables with CREATE

The word `CREATE` switches on compilation to the RAM buffer. When you just need to build a table, you must remember to turn the buffer off to flush the contents to Flash.

```
create MyTable   \ -- addr
  5 ,  6 ,  7 ,  8 ,          \ lay down data
-BuffComp                     \ flush buffer to Flash
```

2.5 Technical support

Technical support for the Lite compilers is done in our spare time. If you have problems, send us an email to

```
tech-support@mpeforth.com
```

3 Control file for STM32F072 Discovery board

Every Forth project has a control file, which is similar to the project file in other languages. The control file tells the cross compiler all about the target. Not all of it is documented here, but it is all commented in the source code.

The control file *CortexLite/Hardware/STM32F072Bdisco/liteSTM32F072sa.ctl* produces a standalone Forth for the STM32F072B Discovery board. The control files for other boards are very similar in structure, and different in the details of the CPU and board specific options. Read the specific code for your CPU and board.

The serial port is connected to USART1 on pins PA9 (Tx -> PC Rx) and PA10 (Rx -> PC Tx). You cannot use the easy GPIO code for UART initialisation unless the GPIO ports are clocked and taken out of reset in the start up code.

When you modify this file for your own hardware do not forget to update the GPIO pin assignments and alternate function selections.

To Flash the board, you can use the low-cost ST-LINK/V2 JTAG unit, or use the one integrated into the ST Discovery boards. The ST-LINK Utility software is a free download from the ST website `www.st.com`.

3.1 Define directory macros

The MPE cross compilers contain a useful text macro system. Macros allow easy porting of control files when projects are moved from their default locations. Each directory macro defines where a particular set of files are located.

```
c" ."                  setmacro AppDir         \ application files
c" ."                  setmacro HwDir          \ Board hardware files
c" ..\.."              setmacro CpuDir         \ CPU specific files
c" ..\..\Examples"     setmacro ExampleDir     \ example files
```

3.2 Turn on the cross compiler

```
include %CpuDir%/Macros         \ macros needed by the cross-compiler

CROSS-COMPILE                   \ Turn host Forth into a cross-compiler

only forth definitions          \ default search order

  no-log                        \ uncomment to suppress output log
  rommed                        \ split ROM/RAM target
  interactive                   \ enter interactive mode at end
  +xrefs                        \ enable cross references
  align-long                    \ code is 32bit aligned
  Cortex-M0                     \ Thumb2 processor type and register usage
  -LongCalls                    \ no calls outside 25 bit range
  +FlashCompile                 \ target compiles to Flash
```

```
  Hex-I32                       \ also produce Intel Hex-32 obj format
  +SaveCdataOnly                \ no data area image files

0 equ false
-1 equ true
```

3.3 Configure target

3.3.1 STM32F0 variant definitions

`$0800:0000 equ FlashBase        \ -- addr`
Start address of Flash. The bottom 2kb (the vector area) is mirrored at $0000:0000 for booting.

`#128 kb equ /Flash      \ -- len`
Size of Flash.

`2 kb equ /FlashPage     \ -- len`
Size of a Flash Page.

`1 equ KeepPages         \ -- u`
Set this non-zero for the number of pages at the end of Flash that are reserved for configuration data. Often set to 1 or 2 by systems that use PowerNet.

`FlashBase /Flash + /FlashPage KeepPages * - equ CfgFlash  \ -- len`
Base address of the configuration Flash area.

`$1FFF:C800 equ /InfoBase        \ -- addr`
Base address of system memory information block.

`#12 kb equ /SysMem              \ -- len`
Size of system memory block.

`$1FFF:F800 equ OptionBytes      \ -- addr`
Base address of option bytes

`#16 equ /OptionBytes            \ -- len`
Number of option bytes

`#64 cells equ /ExcVecs  \ -- len`
Size of the exception/interrupt vector table. There are 16 slots reserved by ARM.
The system clocks are generated from PLLs. How to set them up is non-obvious and is mostly documented by the ST demonstration code in the file *system_stm32f0x2.c*. If you are not going to use an existing setup, copy and rename one of the existing *startSTM32F0xx.fth* files.

`8 MHz equ xtal-speed            \ -- hz`
Master oscillator crystal clock rate in HZ. This is the HSI internal oscillator which has better than +/-1% accuracy and is more accurate than the internal 48 MHz oscillator.

`48 MHz equ system-speed         \ -- hz`
Requested CPU clock speed in HZ. Note that you must calculate the PLL values.

`1 equ AHBdiv    \ -- u`
Division ratio of the AHB clock from the system clock. This may not be more than 48 MHz.

`1 equ APBdiv    \ -- u`
Division ratio of the APB clock from the system clock. This may not be more than 48 MHz.

```
system-speed AHBdiv / equ AHB-speed     \ -- hz
```
AHB bus speed.

```
system-speed APBdiv / equ APB-speed     \ -- hz
```
APB bus speed.

3.3.2 Memory map

If you are using the `Reflash` code in the *ReProg* folder, note that the Flash reprogramming code uses RAM from $2000:0000..$2000:0FFF and its mirrors. Ensure that your stacks are outside this region.

The Flash memory starts at $0800:0000. The bottom 2 kb (the vector area) is mirrored at $0000:0000. The top 4 kb (two pages) is used to save autostart and application linkage information.

```
  $0800:0000 $0800:BFFF cdata section liteSTM32F072sa   \ code
  $2000:0000 $2000:0FFF udata section PROGu     \ 4k UDATA RAM
  $2000:1000 $2000:3FFF idata section PROGd     \ 12k IDATA RAM

interpreter
: prog liteSTM32F072sa  ;               \ synonym
target

PROG PROGd PROGu  CDATA         \ use Code for HERE , and so on

$0801:F000 equ INFOSTART        \ kernel status is saved here.
$0801:F800 equ APPSTART         \ application data is saved here.
$0801:FFFF equ INFOEND          \ end of kernel/application status data.
$2000:0000 equ RAMSTART         \ start of RAM
$2000:4000 equ RAMEND           \ end of RAM
$0800:0000 equ FLASHSTART       \ start of Main Flash
$0810:0000 equ FLASHEND         \ end of possible main Flash
$0800:C000 equ APPFLASHSTART    \ start of application flash
$0801:F000 equ APPFLASHEND      \ end of application flash
APPFLASHEND APPFLASHSTART - equ /APPFLASH       \ size of application flash

APPFLASHSTART TargetFlashStart  \ sets HERE at kernel start up
```

3.3.3 Stack and user area sizes

```
$0F0 equ UP-SIZE                \ size of each task's user area
$0F0 equ SP-SIZE                \ size of each task's data stack
$0100 equ RP-SIZE               \ size of each task's return stack
up-size rp-size + sp-size +
  equ task-size                 \ size of TASK data area
\ define the number of cells of guard space at the top of the data stack
#2 equ sp-guard                 \ can underflow the data stack by this amount

$0100 equ TIB-LEN               \ terminal i/p buffer length

\ define nesting levels for interrupts and SWIs.
```

```
2 equ #IRQs                     \ number of IRQ stacks,
                                \ shared by all IRQs (1 min)
0 equ #SVCs                     \ number of SVC nestings permitted
                                \ 0 is ok if SVCs are unused
```

3.3.4 Serial and ticker rates

`1 equ useUSART1?` `\ -- n`
Set non-zero to compile code for USART1, device `Console1`.

`#115200 equ console1-speed`
`Console1` speed in BPS.

`0 equ useUSART2?` `\ -- n`
Set non-zero to compile code for USART2, device `Console2`.

`#115200 equ console2-speed`
`Console2` speed in BPS.

`0 equ useUSART3?` `\ -- n`
Set non-zero to compile code for USART3, device `Console3`.

`115200 equ console3-speed`
`Console3` speed in BPS.

`0 equ useUSART4?` `\ -- n`
Set non-zero to compile code for USART4, device `Console4`.

`115200 equ console4-speed`
`Console4` speed in BPS.

`1 equ console-port` `\ -- n ; Designate serial port for terminal (0..n).`
Ports 1..4 are the on-chip UARTs. The internal USB device is port 10, and bit-banged ports are defined from 20 onwards.

`#1 equ tick-ms` `\ -- ms`
Timebase tick in ms.

3.3.5 Software selection

With 128 kb of Flash we can select a comfortable set of software and still have plenty of space for application code.

```
 0 equ Tiny?                    \ nz to make a minimal kernel.
 1 equ ColdChain?               \ nz to use cold chain mechanism
 1 equ tasking?                 \ true if multitasker needed
   6 cells equ tcb-size         \   for internal consistency check
 0 equ timebase?                \ true for TIMEBASE code
 0 equ softfp?                  \ true for software floating point
 0 equ FullCase?                \ true to include ?OF END-CASE NEXTCASE extensions
 0 equ target-locals?           \ true if target local variable sources needed
 0 equ romforth?                \ true for ROMForth handler
 0 equ blocks?                  \ true if BLOCK needed
 $0000 equ sizeofheap           \ 0=no heap, nz=size of heap
   1 equ heap-diags?            \   true to include diagnostic code
 0 equ paged?                   \ true if ROM or RAM is paged/banked
```

3.4 Kernel files

```
  include %CpuDir%/CM0def               \ Cortex generic equates and SFRs
  include %CpuDir%/sfrSTM32F072         \ STM32F072 special function registers
  include %CpuDir%/StackDef             \ Reserve default task and stacks
PROGd  sec-top 1+ equ UNUSED-TOP  PROG  \ top of memory for UNUSED
  include %HwDir%/startSTM32F072        \ start up code
l: crcslot
  0 ,                                   \ the kernel CRC
l: crcstart
  include %CpuDir%/CodeM0lite           \ low level kernel definitions
  include %CpuDir%/Drivers/FlashSTM32   \ Flash programming code
  include %CpuDir%/kernel72lite         \ high level kernel definitions
  include %CpuDir%/Drivers/rebootSTM32  \ reboot using watchdog
  include %CpuDir%/IntCortex            \ interrupt handlers for NVIC
  include %CpuDir%/FaultCortex          \ fault exception handlers for NVIC


: selio-ser1    \ --
```

Example to perform clock and pin selection for USART1 on PA9/10. You cannot use the easy GPIO code for this unless the GPIO ports are clocked and taken out of reset in the start up code.

```
  include %CpuDir%/Drivers/serSTM32F0xxp \ polled serial driver
  include %CpuDir%/Dump                 \ DUMP .S etc development tools

  include %CpuDir%/Drivers/gpioSTM32F0xx \ easy pin access
  include %CpuDir%/Drivers/SysTickDisco072
  ' start-clock AtCold

tasking? [if]
  include %CpuDir%/MultiCM0lite         \ multitasker
[then]
```

3.5 End of kernel

After the main kernel has been built, some version data is laid down for use by the sign-on code.

```
buildfile liteSTM32F072sa.no
l: version$
  build$,
l: BuildDate$
  DateTime$,

internal
: .banner       \ --
  cr ." ****************************"
;

: .CPU          \ -- ; display CPU type
  .banner
```

```
  cr ." MPE Forth Lite for STM32F072"
  cr version$ $. space BuildDate$ $.
  cr ." Copyright (C) 2014 MicroProcessor Engineering Ltd."
  .banner
;
external
```

3.6 Application code

This code is not essential, but makes life very much easier.

```
  include %HwDir%/ReProg/ReFlash        \ ReFlash utility
  include %CpuDir%/include              \ include from AIDE
  include %CpuDir%/Drivers/spiSTM32F0hard       \ SPI2 driver
  include %CpuDir%/Examples/l3gd20      \ L3GD20 MEMS driver

RAMEND constant RP-END  \ end of available RAM
```

3.7 Finishing up

```
libraries       \ to resolve common forward references
  include %CpuDir%/LibMOM1
  include %CpuDir%/LIBRARY
end-libs

decimal

\ Add a kernel checksum
crcstart here crcslot crc32 checksum
/DefStart 128 > [if]
  .( DEFSTART area too big ) abort
[then]

update-build                            \ update build number file

FINIS                                   \ all done
```

4 Cortex start up for STM32F072

```
l: ExcVecs      \ -- addr ; start of vector table
```
The exception vector table is /ExcVecs bytes long. The equate is defined in the control file. Note that this table table must be correctly aligned as described in the Cortex-M3 Technical Reference Manual (ARM DDI0337, rev E page 8-21). If placed at 0 or the start of Flash, you'll usually be fine.

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

```
: prediv4       \ u -- x
```
Convert a divider value to a 4 bit pattern as used for HPRE.

```
: prediv3       \ u -- x
```
Convert a divider value to a 3 bit pattern as used for PPRE.

```
: pllmul1       \ u -- x
```
Convert a multiplier value to the 4 bit pattern as used for PLLMUL.
We set up for 48 MHz CPU speed from the HSI oscillator. The HSI oscillator is more accurate than the HSI48, so the HSI is better for UART use.

```
  equ initCFGRval               \ -- x
```
Bit settings for the RCC_CFGR register.

```
  equ initCFGR2val              \ -- x
```
Bit settings for the RCC_CFGR2 register.
The number of wait states required by the Flash depends on the operating frequency.

```
system-speed 1- 24 MHz / equ FLWS       \ -- u
```
The number of Flash wait states required.

```
bit4 ( PRFTBE ) FLWS or  equ initACRval \ -- x
```
Initial value written to the FLASH_ACR register

```
: setClocks     \ --
```
Enable the clocks, set the bus dividers, and enable the PLLs as required. the PLL IS used to generate the main 48 MHz clock. By default, the AHB and APB busses are run at SYSCLK

```
: StartCortex   \ -- ; never exits
```
Set up the Forth registers and start Forth. Other primary hardware initialisation can also be performed here. All the GPIO blocks are enabled.

5 Cortex code definitions

The file *Cortex/CodeM0lite.fth* contains primitives for the standalone Lite Forth kernel.

5.1 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>
```

5.2 Register usage

For Cortex-M0/M1 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         --
r11         up      user area pointer
r10         --
r9          lp      locals pointer
r8          --
r7          tos     cached top of stack
r6          psp     data stack pointer
r0-r5       scratch
```

The VFX optimiser reserves R0 and R1 for internal operations. `CODE` definitions must use R7 as TOS with NOS pointed to by R6 as a full descending stack in ARM terminology. R0..R5, R12 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.

```
0 equ Tiny?     \ -- flag
```
If this equate is not already set, a non-minimal kernel is built.

5.3 Logical and relational operators

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

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

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

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

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

`: 0<>  \ x -- flag`
Compare the top stack item with 0 and return TRUE if not-equal. Not compiled if the flag `TINY?` is set.

`: 0>  \ x -- flag`
Return TRUE if the top of stack is greater-than-zero. Not compiled if the flag `TINY?` is set.

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

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

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

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

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

`: <=  \ n1 n2 -- flag`
Return TRUE if n1 is less than or equal to n2. Not compiled if the flag `TINY?` is set.

`: >=  \ x1 x2 -- flag`
Return TRUE if n1 is greater than or equal to n2. Not compiled if the flag `TINY?` is set.

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

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

`: D0<  \ d -- flag`
Returns true if signed double d is less than zero. Not compiled if the flag `TINY?` is set.

`: D0=  \ xd -- flag`
Returns true if xd is 0. Not compiled if the flag `TINY?` is set.

`CODE D=  \ xd1 xd2 -- flag`
Return TRUE if the two double numbers are equal. Not compiled if the flag `TINY?` is set.

`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.

`: within  \ n1|u1 n2|u2 n3|u3 -- flag`
Return true for *n2* `<=` *n1* `<` *n3*. 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 arshift \ x1 u -- x2`
Arithmetic shift right X1 by U bits.

5.4 Control flow

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

`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`. Not compiled if the flag `TINY?` is set.

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

5.5 Basic arithmetic

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

`CODE M+ \ d1|ud1 n -- d2|ud2`
Add double d1 to sign extended single n to form double d2. Not compiled if the flag `TINY?` is set.

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

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

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

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

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

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

`CODE D- \ d1 d2 -- d3`
Subtract two double precision integers. D3=D1-D2. Not compiled if the flag `TINY?` is set.

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

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

`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. Not compiled if the flag `TINY?` is set.

`CODE D2/ \ xd1 -- xd2`

Divide the given double number by two. Not compiled if the flag `TINY?` is set.

5.6 Multiplication

`: * \ n1 n2 -- n3`

Standard signed multiply. N3 = n1 * n2.

`code UM* \ u1 u2 -- ud`

Perform unsigned-multiply between two numbers and return double result.

`: m* \ n1 n2 -- d`

Signed multiply yielding double result. Not compiled if the flag `TINY?` is set.

5.7 Division

`code um/mod \ ud1 u2 -- urem uquot`

Full 64 by 32 unsigned division subroutine. ARM Cortex-M0/M1 provides no division instructions, so this is a software loop performing repeated trial subtraction.

`: sm/rem \ d n -- rem quot ; symmetric division`

Perform a signed division of double number d by single number n and return remainder and quotient using symmetric (normal) division.

`: /mod \ n1 n2 -- rem quot`

Signed symmetric division of N1 by N2 single-precision returning remainder and quotient. Symmetric.

`: / \ n1 n2 -- n3`

Standard signed division operator. n3 = n1/n2. Symmetric.

`: MOD \ n1 n2 -- n3`

Return remainder of division of N1 by N2. n3 = n1 mod n2.

`: 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.

`: fm/mod \ d n -- rem quot`

Perform a signed division of double number d by single number n and return remainder and quotient using floored division. See the ANS Forth specification for more details of floored division. Not compiled if the flag `TINY?` is set.

`: u/ \ u1 u2 -- u3`

Unsigned division operator. u3 = u1/u2. Not compiled if the flag `TINY?` is set.

`: M/ \ d n1 -- n2`

Signed divide of a double by a single integer. Not compiled if the flag `TINY?` is set.

5.8 Scaling - multiply then divide

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

`: */MOD \ n1 n2 n3 -- n4 n4`
Multiply n1 by n2 to give a double precision result, and then divide it by n3 returning the remainder and quotient. Not compiled if the flag `TINY?` is set.

`: */ \ n1 n2 n3 -- n4`
Multiply n1 by n2 to give a double precision result, and then divide it by n3 returning the quotient. Not compiled if the flag `TINY?` is set.

5.9 Stack manipulation

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

`: TUCK \ x1 x2 -- x2 x1 x2`
Insert a copy of the top data stack item underneath the current second item. Not compiled if the flag `TINY?` is set.

`: 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 \fo{DUP}.

`CODE ROLL \ xu xu-1 .. x0 u -- xu-1 .. x0 xu`
Rotate the order of the top N stack items by one place such that the current top of stack becomes the second item and the Nth item becomes TOS. See also `ROT`. Not compiled if the flag `TINY?` is set.

`: 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.

`: -ROT \ x1 x2 x3 -- x3 x1 x2`
The inverse of `ROT`. Not compiled if the flag `TINY?` is set.

`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. Not compiled if the flag `TINY?` is set.

`CODE 2R> \ -- x1 x2 ; R: x1 x2 --`
Transfer the top two return stack items to the data stack. Not compiled if the flag `TINY?` is set.

`CODE 2R@ \ -- x1 x2 ; R: x1 x2 -- x1 x2`
Copy the top two return stack items to the data stack. Not compiled if the flag `TINY?` is set.

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

`: DUP \ x -- x x`

DUPlicate the top stack item.

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

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

`: 2DUP \ x1 x2 -- x1 x2 x1 x2`
DUPlicate the top cell-pair on the data stack. Not compiled if the flag `TINY?` is set.

`: 2OVER \ x1 x2 x3 x4 -- x1 x2 x3 x4 x1 x2`
As `OVER` but works with cell-pairs rather than single-cell items. Not compiled if the flag `TINY?` is set.

`: 2DROP \ x1 x2 -- )`
Discard the top two data stack items. Not compiled if the flag `TINY?` is set.

`: 2SWAP \ x1 x2 x3 x4 -- x3 x4 x1 x2`
Exchange the top two cell-pairs on the data stack. Not compiled if the flag `TINY?` is set.

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

`CODE SP@ \ -- x`
Get the current address value of the data-stack pointer. Not compiled if the flag `TINY?` is set.

`CODE SP! \ x --`
Set the current address value of the data-stack pointer. Not compiled if the flag `TINY?` is set.

`CODE RP@ \ -- x`
Get the current address value of the return-stack pointer. Not compiled if the flag `TINY?` is set.

`CODE RP! \ x --`
Set the current address value of the return-stack pointer. Not compiled if the flag `TINY?` is set.

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

5.10 String and memory operators

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

`: /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. Not compiled if the flag `TINY?` is set.

`: 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. Not compiled if the flag `TINY?` is set.

`code cmove \ asrc adest len --`
Copy *len* bytes of memory forwards from *asrc* to *adest*.

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

`: MOVE \ addr1 addr2 u -- ; intelligent move`
An intelligent memory move, chooses between `CMOVE` and `CMOVE>` at runtime to avoid memory overlap problems. Note that as Forth Lite characters are 8 bit, there is an implicit connection between a byte and a character.

`: upc \ char -- char' ; force upper case`
Convert char to upper case.

`: upper \ c-addr len --`
Convert the ASCII string described to upper-case. This operation happens in place.

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

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

`: Erase \ addr n --`
Fill U bytes of memory from A-ADDR with 0. Not compiled if the flag `TINY?` is set.

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

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

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

`: ! \ x a-addr --`
Store the CELL quantity X at memory A-ADDR.

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

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

`: ON \ a-addr --`
Given the address of a CELL this will set its contents to TRUE (-1). Not compiled if the flag `TINY?` is set.

`: OFF \ a-addr --`
Given the address of a CELL this will set its contents to FALSE (0). Not compiled if the flag `TINY?` is set.

`: INCR \ a-addr --`
Increment the data cell at a-addr by one. Not compiled if the flag `TINY?` is set.

`: DECR \ a-addr --`
Decrement the data cell at a-addr by one. Not compiled if the flag `TINY?` is set.

`: +! \ n|u a-addr --`
Add N to the CELL 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.

5.11 Miscellaneous words

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

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

`: >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 a TRUE flag, otherwise return FALSE.

5.12 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. Not compiled if the flag `TINY?` is set.

`CODE CELLS \ n1 -- n2`
Return the size in address units of N1 cells in memory. Not compiled if the flag `TINY?` is set.

`CODE CELL- \ a-addr1 -- a-addr2`
Decrement an address by the size of a cell. Not compiled if the flag `TINY?` is set.

`CODE CELL \ -- n`
Return the size in address units of one CELL. Not compiled if the flag `TINY?` is set.

5.13 Code buffer in RAM

Many ARM chips cannot program their own Flash on a byte by byte basis. Some chips will only program a single 32 bit word set to $FFFF:FFFF. Others will only program a minimum of 64 bytes. To avoid unpleasantness and compromises, code is compiled into RAM and then copied to Flash. In most cases, this is all hidden from the programmer, but in some cases you do need to be aware of what is happening. The actual implementation is chip specific and is contained in the Flash driver file for the chip.

A few words are common to all implementations. For further details see the relevant Flash driver source code.

`: +BuffComp \ --`
Start compiling into the RAM buffer. `HERE` still points into Flash. This word is already used inside words such as `:` and `VARIABLE`.

`: -BuffComp \ --`
Stop compiling into the RAM buffer and program the RAM buffer to Flash. This word is already inside `;` and `VARIABLE`.

`: c@c \ addr -- b`
8 bit fetch from Flash or the RAM buffer if active

`: w@c \ addr -- w`
16 bit fetch from Flash or the RAM buffer if active

`: @c \ addr -- x`
32 bit fetch from Flash or the RAM buffer if active

`: c!c \ b addr --`
8 bit store to Flash or the RAM buffer if active

`: w!c \ w addr --`
16 bit store to Flash or the RAM buffer if active

`: !c \ b addr --`
32 bit store to Flash or the RAM buffer if active

5.14 Supporting compilation on the target

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

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

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

5.15 Defining words and runtime support

`: aligned \ addr -- addr'`

Given an address pointer this word will return the next ALIGNED address subject to system wide alignment restrictions.

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

`: >BODY  \ xt -- a-addr`
Move a pointer from a CFA or "XT" to the definition's data area. `>BODY` should only be used with children of `CREATE`. If `FOOBAR` is defined with `CREATE foobar`, then the phrase `' FOOBAR >BODY` would give the same result as executing `FOOBAR`.

`: (;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.

`: Imm  \ --`
Used before a definition to indicate that it is `IMMEDIATE`, which means that it will execute whenever encountered regardless of whether the Forth system is compiling or interpreting.

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

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

`: DOES>  \ C: colon-sys1 -- colon-sys2 ; Run: -- ; R:  nest-sys --`
Start 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>`.

`: CREATE  \ --`
Create a new definition in the dictionary, turning on compiling into the code buffer in RAM. When the new definition is executed it will return the address of the definition's data area. If you use CREATE to create a data table, be sure to turn off compilation into the code buffer when the table is complete, e.g.

```
create foo  5 ,  6,  7 ,  8 ,  -BuffComp
```

`: <BUILDS  \ --`
A synonym for `CREATE`. May be removed in a future release. Only kept for short-term compatibility with the MSP430 Lite editions. Not compiled if the flag `TINY?` is set.

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

`: EQU  \ x "<spaces>name" -- ; Exec: -- x`
A synonym for `CONSTANT` above to ease interactive debugging of target drivers that are normally cross-compiled. Create a new `CONSTANT` called `name` which has the value "x". When `NAME` executes the value *\i{x) is returned.

`: 2CONSTANT  \ Comp: x1 x2 "<spaces>name" -- ; Run: -- x1 x2`
A two-cell equivalent of `CONSTANT`. Not compiled if the flag `TINY?` is set.

`: 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. The RAM is not initialised. Note that the state of a user-defined `VARIABLE` is not preserved across power-down and restart.

`: 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
```

`: CRASH \ -- ; used as action of DEFER`
The default action of a `DEFER`ed word, which is `NOOP`,

`: DEFER \ Comp: "<spaces>name" -- ; Run: i*x -- j*x`
Creates a new `DEFER`ed word. No default action is assigned. User-defined `DEFER`ed words **must** be initialised by the application before use.

```
' <action> IS <deferredword>
```

or (when compiled)

```
['] <action> IS <deferredword>
```

Note that the state of a user-defined `DEFER`red word is not preserved across power-down and restart.

`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.

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

`: to \ --`
store operator for use with `VALUE`s.

5.16 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.

5.17 Branch constructors

Used when compiling code on the target.

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

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

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

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

5.18 Main compilers

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

`: c_drop        \ --`
Compile the code for `DROP`. INTERNAL. Not compiled if the flag `TINY?` is set.

`: 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. Not compiled if the flag `TINY?` is set.

`: c_case        \ -- addr`
Compile the code for `CASE`. INTERNAL. Not compiled if the flag `TINY?` is set.

`: c_OF          \ C: -- of-sys ; Run: x1 x2 -- | x1`
Compile the code for `OF`. INTERNAL. Not compiled if the flag `TINY?` is set.

`: c_ENDOF       \ C: case-sys1 of-sys -- case-sys2 ; Run: --`
Compile the code for `ENDOF`. INTERNAL. Not compiled if the flag `TINY?` is set.

`: FIX-EXITS     \ n1..nn --`

Compile the code to resolve the forward branches at the end of a `CASE` structure. INTERNAL. Not compiled if the flag `TINY?` is set.

`: c_ENDCASE \ C: case-sys -- ; Run: x --`
Compile the code for `ENDCASE`. INTERNAL. Not compiled if the flag `TINY?` is set.

`: c_?OF \ C: -- of-sys ; Run: flag --`
Compile the code for `?OF`. INTERNAL. Not compiled if the flag `TINY?` is set.

5.19 More miscellaneous words

`defer pause \ -- ; multitasker hook`
Allows the sytem multitasker to get a look in. If the multitasker has not been compiled, `PAUSE` is set to `NOOP`.

`code di \ --`
Disable interrupts.

`code ei \ --`
Enable interrupts.

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

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

`: setMask \ value mask addr -- ; cell operation`
Clear the *mask* bits at *addr* and set (or) the bits defined by *value*.

5.19.1 Non-minimal systems

The following words are only compiled if the equate `Tiny?` is set to zero.

`: 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.

`: bor! \ mask addr --`
Set *mask* bits in the byte at *addr*.

`: bbic! \ mask addr --`
Clear *mask* bits in the byte at *addr*.

`: btoggle! \ mask addr --`
Toggle the *mask* bits in the byte at *addr*.

`: btst \ mask addr -- x`
Return non-zero if the *mask* bits in the byte at *addr* are non-zero.

`: or! \ mask addr --`
Set *mask* bits in the cell at *addr*.

`: bic! \ mask addr --`
Clear *mask* bits in the cell at *addr*.

`: toggle! \ mask addr --`
Toggle the *mask* bits in the cell at *addr*.

`: tst \ mask addr -- x`
Return non-zero if the *mask* bits in the cell at *addr* are non-zero.

`: cset \ mask addr --`
Set *mask* bits in the cell at *addr*.

`: cclr \ mask addr --`
Clear *mask* bits in the byte at *addr*.

`: ctoggle \ mask addr --`
Toggle the *mask* bits in the byte at *addr*.

`: cget \ mask addr -- flag`
Return non-zero if the *mask* bits in the byte at *addr* are non-zero.

6 High level kernel - kernel72lite.fth.

The Forth kernel words documented here are entirely written in high-level Forth. The kernel is reduced in size to match available code size in small devices.

6.1 User variables

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

```
: +user         \ size --
```
Used in the cross compiler to create a `USER` variable *size* bytes long at the next available offset and updates that offset.

```
tcb-size +user SELF             \ task identifier and TCB
```
When multitasking is installed, the task control block for a task occupies `TCB-SIZE` bytes at the start of the user area. Thus the user area pointer also acts as a pointer to the task control block. Not compiled if the flag `Tasking?` is not set.

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

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

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

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

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

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

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

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

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

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

6.2 System data

6.2.1 Constants

```
$20 constant BL \ -- char
```
A blank space character.

6.2.2 System variables and data

Note that DP and RP must be declared in that order.

`variable DP  \ -- addr`
Flash dictionary pointer.

`variable RP  \ -- addr`
RAM dictionary pointer.

`variable xDP  DP xDP !  \ -- addr`
Holds the address of the current dictionary pointer, DP or RP.

6.3 Vectored I/O handling

6.3.1 Introduction

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

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

6.3.2 Building a vector table

The example below is taken from an ARM implementation.

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

Console1 opvec !  Console1 ipvec !
```

6.3.3 Generic I/O words

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

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

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

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

`: CR \ -- ; display new line`

Perform the equivalent of a CR/LF pair on the current output device. `OUT` is zeroed. before executing the vector function.

`: SPACE \ --`

Output a blank space (ASCII 32) character.

`: SPACES \ n --`

Output n spaces, where $n > 0$. If $n < 0$, no action is taken.

6.4 Laying data in memory

These words are used to control and place data in memory. Note that the Forth system compiles headers and code into Flash memory.

`: HERE \ -- addr`

Return the current dictionary pointer which is the first address-unit of free space within the system.

`: ORG \ addr --`

Set the current dictionary pointer.

`: ALLOT \ n --`

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

`: RHERE \ -- addr`

Return the current RAM dictionary pointer.

`: RALLOT \ n --`

Allocate n bytes of RAM from `RHERE` and move the pointer.

`: RALIGN \ --`

Force `RHERE` to be cell aligned.

`: ROM \ --`

`HERE`, `ORG`, `ALLOT`, `,` and friends, are set to use the Flash dictionary pointer. This is the default.

`: RAM \ --`

`HERE`, `ORG` and `ALLOT` are set to use the RAM dictionary pointer. Use in the form:

```
RAM ... ROM
```

`: aligned \ addr -- addr'`

Given an address pointer this word will return the next `ALIGNED` address subject to system wide alignment restrictions.

`: ALIGN \ --`

`ALIGN` dictionary pointer using the same rules as `ALIGNED`.

`: , \ x --`

Place the CELL value X into the dictionary at `HERE` and increment the pointer.

`: w, \ x --`

Place the 16 bit value X into the dictionary at `HERE` and increment the pointer.

`: C, \ b --`

Place an 8 bit byte into the dictionary at `HERE` and increment the pointer.

6.5 Dictionary management

The Forth header is laid out as below. The start and end of the header are aligned at cell boundaries.

```
Link | Count | <name>
----------------------
Cell |  Byte | n Bytes
----------------------
```

`Link` Also called LFA. This field contains the address of the of the next count byte in the same thread of the wordlist.

`Count/Ctrl`
The bottom five bits contain the length (0..31) of the name in bytes. The top three bits are used as follows:

```
Bit 7    Always set
Bit 6    Immediate bit (0=immediate)
Bit 5    Reserved
```

`<name>` A string of ASCII characters which make up the name of the word..

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

`: .NAME \ nfa --`
Display a definition's name given an NFA.

6.6 String compilation

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

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

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

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

6.7 ANS words CATCH and THROW

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

`CATCH` and `THROW` provide a reliable mechanism for handling exceptions, without having to propagate exception flags through multiple levels of word nesting. It is similar in spirit to the "nonlocal return" mechanisms of many other languages, such as C's **setjmp()** and **longjmp()**, and

LISP's **CATCH** and **THROW**. In the Forth context, THROW may be described as a "multi-level EXIT", with CATCH marking a location to which a THROW may return.

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

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

6.7.1 Example use

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

Typical use:

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

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

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

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

6.7.2 Gotchas

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

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

6.7.3 User words

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

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

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

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

6.8 Formatted and unformatted i/o

6.8.1 Setting number bases

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

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

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

6.8.2 Numeric output

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

```
: #              \ ud1 -- ud2
```
Given a double number on the stack this will add the next digit to the pictured numeric output buffer and return the next double number to work with. **PLEASE NOTE** that the numeric output string is built from **right** (l.s. digit) to **left** (m.s. digit).

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

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

```
: #>             \ xd -- c-addr u
```

Terminate defnition of a numeric output string. Return the address and length of the ASCII string.

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

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

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

`: U. \ u --`
As with . but treat as unsigned. Not compiled if the flag `TINY?` is set.

`: U.R \ u n --`
As with D.R but uses a single-unsigned cell value. Not compiled if the flag `TINY?` is set.

`: .R \ n1 n2 --`
As `D.R` but uses a single-signed cell value. Not compiled if the flag `TINY?` is set.

6.8.3 Numeric input

`: +DIGIT \ d1 n -- d2 ; accumulates digit into double accumulator`
Multiply d1 by the current radix and add n to it. INTERNAL.

`: >NUMBER \ ud1 c-addr1 u1 -- ud2 c-addr2 u2 ; convert all until non-digits`
Accumulate digits from string c-addr1/u2 into double number ud1 to produce ud2 until the first non-convertible character is found. c-addr2/u2 represents the remaining string with c-addr2 pointing the non-convertible character. The number base for conversion is defined by the contents of `USER` variable `BASE`. `>NUMBER` is case insensitive.

`: (INTEGER?) \ c-addr u -- d/n/- 2/1/0`
The guts of `INTEGER?` but without the base override handling. See `INTEGER?` INTERNAL.

`: Check-Prefix \ addr len -- addr' len'`
If any `BASE` override prefices or suffices are used in the input string, set `BASE` accordingly and return the string without the override characters. INTERNAL.

`: integer? \ $addr -- n 1 | d 2 | 0`
Attempt to convert the counted string at 'addr' to an integer. The return result is either 0 for failed, 1 for a single-cell return result followed by that cell, or 2 for a double return. The ASCII number string supplied can also contain implicit radix over-rides. A leading $ enforces hexadecimal, a leading # enforces decimal and a leading % enforces binary.

`defer number? \ $addr -- n 1 | d 2 | 0`
Attempt to convert the counted string at 'addr' to an integer. The return result is either 0 for failed, 1 for a single-cell return result followed by that cell, or 2 for a double return. The ASCII number string supplied can also contain implicit radix over-rides. A leading $ enforces hexadecimal, a leading # enforces decimal and a leading % enforces binary. By default, this word is set to `integer?`.

6.9 String input and output

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

`: ?BS \ pos -- pos' step ; perform BS if pos non-zero`
If pos is non-zero and `ECHOING` is set, perform `BS` and return the size of the step, 0 or -1. INTERNAL.

`: SAVE-CH \ char addr -- ; save as required`
Save char at addr, and output the character if `ECHOING` is set. INTERNAL.

`: ." \ "ccc<quote>" --`
Output the text upto the closing double-quotes character. Use `.( <text>)` when interpreting.

`: $. \ c-addr -- ; display counted string`
Output a counted-string to the output device.

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

6.10 Source input control

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

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

`: REFILL \ -- flag ; refill input source`
Attempt to refill the terminal input buffer from the current source. This may be a file or the console. An attempt to refill when the input source is a string will fail. The return result is a flag indicating success with TRUE and failure with FALSE. A failure to refill when the input source is a text file indicates the end of file condition. Not compiled if the flag `TINY?` is set.

6.11 Text scanning

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

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

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

6.12 Miscellaneous

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

6.13 Wordlist control

`here is-action-of vocabulary \ --`
The runtime action of a `VOCABULARY`.

6.14 Control structures

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

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

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

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

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

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

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

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

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

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

```
BEGIN ... UNTIL/AGAIN
BEGIN ... WHILE ... REPEAT
```

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

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

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

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

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

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

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

`: RECURSE \ Comp: --`

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

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

to compile a reference to FOO from inside FOO.

6.14.1 CASE statement

The CASE statement words are only compiled if the flag Tiny? is not set. Tiny? is only set for minimal targets such as for LPC81x chips.

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

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

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

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

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

6.15 Target interpreter and compiler

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

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

`: POSTPONE \ Comp: "<spaces>name" --`
Compile a reference to another word. POSTPONE can handle compilation of IMMEDIATE words which would otherwise be executed during compilation.

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

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

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

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

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

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

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

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

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

`: [COMPILE] \ "<spaces>name" --`
Compile the next word in the input stream. `[COMPILE]` ignores the `IMMEDIATE` state of the word. `[COMPILE]` is mostly superceded by `POSTPONE`. Not compiled if the flag `TINY?` is set.

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

`: \ \ "ccc<eol>" --`
Begin a single-line comment. All text up to the end of the line is ignored.

`: (( \ -- ; (( ... ))`
Block comment operator. Any source following this is ignored upto and including the terminator, '))', which must be white space separated. Not compiled if the flag `TINY?` is set.

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

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

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

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

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

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

`: EVALUATE \ i*x c-addr u -- j*x ; interpret the string`
Process the supplied string as though it had been entered via the interpreter. Not compiled if the flag `TINY?` is set.

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

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

6.16 Startup code

6.16.1 Cold chain

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

```
' <wordname> AtCold
```

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

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

```
l: ColdChainFirst       \ -- addr
```
Dummy first entry in ColdChain.

```
variable ColdChain      \ -- addr
```
Holds the address of the last entry in the cold chain.

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

```
: WalkColdChain         \ --                                    MPE.0000
```
Execute all words added to the cold chain. Note that the first word added is executed first.

6.16.2 The COLD sequence

At power up, the target executes COLD or the word specified by MAKE-TURNKEY <name>, or the word specified as the action of an application compiled by the target.

```
: (INIT)        \ --
```
Performs the high level Forth startup. See the source code for more details. INTERNAL.

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

```
: Commit        \ xt|0 --
```
Preserve the compiled image. If xt is non-zero, that word will be executed when the application starts.

```
: Empty         \ --
```
Wipe the application and perform a cold restart.

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

6.17 Kernel error codes

-1 ABORT

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

7 Debug tools

Some simple debug tools can be found in *dump.fth*.

`: dump  \ addr len --`
Display the given block of memory in hex and ASCII.

`: pdump  \ addr len --`
Display the given block of memory as hex 32 bit words. Not on LPC812.

`: .S  \ i*x -- i*x`
Display the stack contents

8 Compile source code from AIDE

The file *include.fth* provides support for compiling a source file from the AIDE server.

```
variable disk-error     \ -- addr
```
Receives transfer error status from the host.

```
: end-load      \ -- ; switch back to keyboard input
```
This word is automatically performed at the end of a download to tidy up the comms.

```
: include       \ "<filename>" -- ; load file from host
```
Compile a file across the serial line from the AIDE file server. Use in the form:

```
  include <filename>
```

The filename extension must be supplied.

```
0 value ShowLines?      \ -- flag
```
Set this non-zero to display source lines during compilation.

```
: include       \ "<filename>" -- ; load file from host
```
Compile a file across the serial line from the AIDE file server. Use in the form:

```
  include <filename>
```

The filename extension must be supplied.

9 Minimal Umbilical code definitions

The file *Cortex/MinM0M1.fth* contains the minimum code definitions required to support Umbilical Forth for Cortex-M0/M1 variants. If additional words are required, they may be copied to a new file from *Cortex/CodeM0M1.fth* or from *Common/Kernel62.fth*.

9.1 Register usage

For Cortex-M0/M1 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         --
r11         up      user area pointer
r10         --
r9          lp      locals pointer
r8          --
r7          tos     cached top of stack
r6          psp     data stack pointer
r0-r5       scratch
```

The VFX optimiser reserves R0 and R1 for internal operations. `CODE` definitions must use R7 as TOS with NOS pointed to by R6 as a full descending stack in ARM terminology. R0..R5, R12 are free for use by `CODE` definitions and need not be preserved or restored. You should assume that any register can be affected by other words.

9.2 Flow of control

`CODE (DO)  \ limit index --`
The run time action of `DO` compiled on the target. The branch target address is in-line and must have the T bit set. INTERNAL.

`CODE (?DO)  \ limit index --`
The run time action of `?DO` compiled on the target. The branch target address is in-line and must have the T bit set. INTERNAL.

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

9.3 Stack operations and maths

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

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

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

`CODE WITHIN  \ n1|u1 n2|u2 n3|u3 -- flag`

The ANS version of WITHIN?. This word uses unsigned arithmetic, so that signed compares are treated as existing on a number circle.

9.4 Multiplication

`code UM*  \ u1 u2 -- ud`

Perform unsigned-multiply between two numbers and return double result.

`: *  \ n1 n2 -- n3`

Standard signed multiply. N3 = n1 * n2.

`: m*  \ n1 n2 -- d`

Signed multiply yielding double result.

9.5 Division

ARM Cortex-M0 provides no division instructions.

`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`

Full 64 by 32 unsigned division subroutine. This routine uses a loop for code size.

`: fm/mod  \ d n -- rem quot ; floored division`

Perform a signed division of double number d by single number n and return remainder and quotient using floored division. See the ANS Forth specification for more details of floored division.

`: sm/rem  \ d n -- rem quot ; symmetric division`

Perform a signed division of double number d by single number n and return remainder and quotient using symmetric (normal) division.

`: /mod  \ n1 n2 -- rem quot`

Signed symmetric division of N1 by N2 single-precision returning remainder and quotient. Symmetric.

`: /  \ n1 n2  -- n3`

Standard signed division operator. n3 = n1/n2. Symmetric.

`: u/  \ u1 u2  -- u3`

Unsigned division operator. u3 = u1/u2.

`: MOD  \ n1 n2 -- n3`

Return remainder of division of N1 by N2. n3 = n1 mod n2.

`: */MOD  \ n1 n2 n3 -- n4 n4`

Multiply n1 by n2 to give a double precision result, and then divide it by n3 returning the remainder and quotient. The point of this operation is to avoid loss of precision.

`: */  \ n1 n2 n3 -- n4`

Multiply n1 by n2 to give a double precision result, and then divide it by n3 returning the quotient. The point of this operation is to avoid loss of precision.

`: M/  \ d n1 -- n2`

Signed divide of a double by a single integer.

9.6 Miscellaneous math

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

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

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

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

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

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

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

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

9.7 Strings

`code cmove \ asrc adest len --`
Copy *len* bytes of memory forwards from *asrc* to *adest*.

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

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

`: erase \ c-addr u -- ; wipe memory`
Set U bytes of memory starting at C-ADDR with zeros.

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

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

`: (C") \ -- c-addr`
The run time action compiled by C".

`: (S") \ -- c-addr u`
The run time action compiled by S".

9.8 Return address manipulations

`CODE >RR \ x -- ; R: -- x`
Push the current top item of the data stack onto the top of the return stack as a return address

`CODE RR> \ -- x ; R: x --`

Pop the caller's return address from the return stack.

```
CODE RR@        \ --  x  ; R:  x -- x
```

Copy the top item from the return stack to the data stack.

9.9 Umbilical versions of defining words

```
here is-action-of constant
```

The runtime code for a CONSTANT.

```
here is-action-of variable
```

The runtime action for a VARIABLE.

```
here is-action-of value
```

The runtime action of a VALUE.

```
here is-action-of user
```

The runtime action of a USER variable.

```
: u#            \ "<name>"-- u
```

An `INTERPRETER` word that returns the index of the `USER` variable whose name follows, e.g.

```
  u# S0
```

```
: CRASH         \ --  ; used as action of DEFER
```

The default action of a DEFERed word. A NOOP.

```
here is-action-of DEFER \ Comp: "<spaces>name" -- ; Run: i*x -- j*x
```

The runtime action of a DEFERred word.

9.10 Display words

```
: SPACE         \ --
```

Output a blank space (ASCII 32) character.

```
: SPACES        \ n --
```

Output 'n' spaces, where 'n' > 0. If 'n' < 0, no action is taken.

```
: .nibble       \ n --
```

Convert a nibble to a hex ASCII digit and display it.

```
: .BYTE         \ b --
```

Display the byte *b* as a 2 digit hex number.

```
: .WORD         \ w --
```

Display w as a 4 digit unsigned hexadecimal number.

```
: .dword        \ x --
```

Display x as an 8 digit unsigned hexadecimal number.

```
: .lword        \ x --
```

A synonym for `.DWORD` above.

9.11 Multitasker hook

```
defer pause     \ -- ; multitasker hook
```

Allows the sytem multitasker to get a look in. If the multitasker has not been compiled, `PAUSE` is set to `NOOP`.

9.12 Miscellaneous

`: setMask \ value mask addr -- ; cell operation`
Clear the *mask* bits at *addr* and set (or) the bits defined by *value*.

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

10 ARM Cortex specific library code

The code in *Cortex/LibCortex.fth* is conditionally compiled by the following code fragment to be found at the end of many control files.

```
libraries       \ to resolve common forward references
  include %CpuDir%/LibCortex
  include %CommonDir%/library
end-libs
```

Each definition in a library file is surrounded by a phrase of the form:

`[required] <name> [if] : <name> ... ;  [then]`

The phrase `[REQUIRED] <name>` returns true if `<name>` has been forward referenced and is still unresolved. The code between `LIBRARIES` and `END-LIBS` is repeatedly processed until no further references are resolved.

10.1 I/O initialisation

`: init-io       \ addr --`
Copy the contents of the I/O set up table to an I/O device. Each element of the table is of the form addr (cell) followed by data (cell). The table is terminated by an address of 0. A table of a single 0 address performs no action.

10.2 interrupt enable and disable

`code di         \ --`
Disable interrupts.

`code ei         \ --`
Enable interrupts.

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

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

10.3 Miscellaneous

`: @OFF          \ addr -- x`
Read cell at addr, and set it to 0.

`: @on           \ addr -- val`
Fetch contents of cell at addr and set it to -1.

11 Device drivers

This chapter documents a number of simple device drivers that can be added to the system, either by cross-compilation or by direct compilation onto the target.

11.1 STM32 GPIO utilities

The code in *CortexLite\Drivers\gpioSTM32.fth* provides utility words for accessing GPIO pins on a bit by bit basis for the STM32 CPUs. The code is written for ease of use rather than performance.

11.1.1 Defining I/O pins

FIO/GPIO acess is defined using a bit number. Bits 0..31 form P0.0 to P0.31, bits 32..63 form P1.0 to P1.31 and so on. Although ST name their ports from A, here we number them from 0. Note also that ST usually only provide 16 I/O bits per port. Later we define constants for the port numbers.

```
$0400 equ /Port          \ -- u
```

Separation in memory between GPIO port base addresses.

```
: PIO:           \ port# bit# -- ; -- struct
```

Define a port I/O bit by name. For example

```
  3 12 pio: LED1   \ PD.12
  PD 12 pio: LED1  \ PD.12
```

defines an I/O bit on GPIOD, bit 12 as `LED1`. At run time, the I/O bit structure is returned.

Define the ports as constants. So that we can use names rather than numbers, e.g.

```
  PD 12 pio: LED1   \ PD.12
```

```
0 constant PA
1 constant PB
2 constant PC
3 constant PD
4 constant PE
5 constant PF
```

11.1.2 GPIO pin access

```
: setPin         \ struct --
```

Set the pin high.

```
: clrPin         \ struct --
```

Set the pin low.

```
: getPin         \ struct -- 0/1
```

Read the pin state.

11.1.3 IO pin configuration

```
: isPinMode     \ mode struct --
```
Sets the GPIO mode for a pin. Use with the constants below.
Pins can be in one of four modes, 0..3.

```
0 constant InputMode    \ -- mode#
```
The pin is an input.

```
1 constant OutputMode   \ -- mode#
```
The pin is an output.

```
2 constant AFmode       \ -- mode#
```
The pin is for one of the alternate function modes.

```
3 constant AnalogMode   \ -- mode#
```
The pin is an analogue pin.

```
: isInput       \ struct --
```
Set the pin to GPIO mode and input.

```
: isOutput      \ struct --
```
Set the pin as an output.

```
: isFunction    \ af# struct --
```
Set the pin to be used with one of the alternate functions.

```
: isAnalog      \ struct --
```
Set the pin as an analog pin.

```
: isPinOD       \ 0/1 struct --
```
Enable (nz) or disable (0) the open drain driver.

```
: isPinPuPd     \ mode struct --
```
Sets the pull up/down for a pin.

```
: NotPulled     \ struct --
```
No pull up or pull down.

```
: PulledUp      \ struct --
```
Enable the pin pull up resistor.

```
: PulledDown    \ struct --
```
Enable the pin pull down resistor.

```
: isPinSpeed    \ mode struct --
```
Sets the output speed for a pin according to the constants below.

```
0 constant LowSpeed     \ -- mask
```
The default is 2MHz, low speed.

```
1 constant MediumSpeed  \ -- mask
```
10 MHz, medium speed.

```
3 constant HighSpeed    \ -- mask
```
50 MHz, fast speed.

```
: initGPIO      \ --
```
Enable clocks to all GPIO ports and take them out of reset. performed in the cold chain. No longer performed because GPIOs are initialised in the start up code.

11.1.4 Test code for STM32F072 Discovery board

```
pa 8 pio: PA8   \ -- struct
```
Structure for pin PA8, which can be used as the oscillator output MCO.

```
: init-mco      \ --
```
Initialise PA8 as the MCO output.

11.2 XMC1xxx GPIO utilities

The code in *CortexLite\Drivers\gpioXMC1xxx.fth* provides utility words for accessing GPIO pins on a bit by bit basis for Infineon XMC CPUs. The code is written for ease of use rather than performance.

11.2.1 Defining I/O pins

FIO/GPIO acess is defined using a bit number. Bits 0..31 form P0.0 to P0.31, bits 32..63 form P1.0 to P1.31 and so on. Although ST name their ports from A, here we number them from 0. Note also that ST usually only provide 16 I/O bits per port. Later we define constants for the port numbers.

```
$0100 equ /Port         \ -- u
```
Separation in memory between GPIO port base addresses.

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

```
  1 12 pio: LED1   \ P1.12
  P1 12 pio: LED1  \ P1.12
```

defines an I/O bit on PORT1, bit 12 as `LED1`. At run time, the I/O bit structure is returned.

Define the ports as constants. So that we can use names rather than numbers, e.g.

```
  PD 12 pio: LED1   \ PD.12
0 constant P0
1 constant P1
2 constant P2
```

11.2.2 GPIO pin access

```
: setPin        \ struct --
```
Set the pin high.

```
: clrPin        \ struct --
```
Set the pin low.

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

11.2.3 IO pin configuration

```
: isPinMode     \ mode struct --
```
Sets the GPIO mode for a pin. Use with the constants below.
Pins can be in one of eight modes or alternate functions. Some modifiers can be applied.

```
%00000 constant DirectIN        \ direct input
%00100 constant InvertIN        \ inverted input
: +PD ( x -- x' ) %00001 or  ;  \ with pull down (i/p)
: +PU ( x -- x' ) %00010 or  ;  \ with pull down (i/p)
: +PS ( x -- x' ) %00011 or  ;  \ with sample on OUT (i/p)

%10000 constant ModeOUT         \ output with direct input
%10001 constant AF1             \ alternate functions
%10010 constant AF2
%10011 constant AF3
%10100 constant AF4
%10101 constant AF5
%10110 constant AF6
%10111 constant AF7
: +PP ( x -- x' ) ;             \ select push pull
: +OD ( x -- x' ) %01000 or  ;  \ select open drain


: isInput       \ struct --
```

Set the pin to be an input (`DirectIN`).

```
: isOutput      \ struct --
```

Set the pin as an output (`ModeOUT`).

11.2.4 Test code for XMC2Go board

```
P1 0 pio: LED1  \ -- struct
```

Structure for pin 1.0 connected as LED1.

```
P1 1 pio: LED2  \ -- struct
```

Structure for pin 1.1 connected as LED2.

11.3 STM32F0 polled serial driver

The serial driver *Cortex/Drivers/serSTM32F0xxp.fth* provides polled serial drivers for the on-chip USARTs. The supplied driver implements USART1 on PA9/10 and USART3 on PD8 (Tx) and PD9 (Rx).

When you copy this file for your own hardware do not forget update the GPIO pin assignments and alternate function selections.

11.3.1 Baud rate generation

The baud rate routines use 16 times sampling and the fractional baud rate generator.

$$\text{divisor} = \frac{\text{pclkx}}{16 * \text{baudrate}}$$

We then treat the divisor as an integer and four bit fraction, effectively scaling it by 16 again.

```
          pclkx
BRRval = --------
         baudrate
```

```
: genBRRval     \ baud clock -- divisor
```
Generate the required BRR value.

11.3.2 Shared code

```
: +FaultConsole \ --
```
For use in fault exception handlers, the multi-tasker must be turned off and `EMIT` and friends must run in polled mode.

11.3.3 USART1

USART1 is on the APB bus, so the baud rate is calculated using that bus clock.

```
console1-speed APB-speed genBRRval equ /us1BRR  \ -- u
```
Baud rate divisor for USART1.

```
: selio-ser1    \ --
```
Example to perform clock and pin selection for USART1 on PA9/10. Provide your own version, e.g. in the control file.

```
: init-ser1     \ --
```
Initialise USART1 for polled operation. Pin selection is performed by `selio-ser1`.

```
: serkey?1      \ -- flag
```
`KEY?` for USART1.

```
: serkey1       \ -- char
```
`KEY` for USART1.

```
: seremit1      \ char --
```
`EMIT` for USART1.

```
: serType1      \ c-addr len --
```
`TYPE` for USART1.

```
: sercr1        \ --
```
`CR` for USART1.

```
create Console1 \ -- addr ; OUT managed by upper driver
```
Device structure for USART1

```
  ' serkey1 ,           \ -- char ; receive char
  ' serkey?1 ,          \ -- flag ; check receive char
  ' seremit1 ,          \ -- char ; display char
  ' sertype1 ,          \ caddr len -- ; display string
  ' sercr1 ,            \ -- ; display new line


  console1 constant console
```
Device structure for console on USART1.

11.3.4 USART2

USART2 is on the APB bus, so the baud rate is calculated using that bus clock.

```
console2-speed APB-speed genBRRval equ /us2BRR  \ -- u
```

Baud rate divisor for USART2.

`: selio-ser2 \ --`
Example to perform clock and pin selection for USART2 on PA2/3. You cannot use the easy GPIO code for this unless the GPIO ports are clocked and taken out of reset in the start up code.

`: init-ser2 \ --`
Initialise USART2 for polled operation. Pin selection is performed by `selio-ser2`.

`: serkey?2 \ -- flag`
`KEY?` for USART2.

`: serkey2 \ -- char`
`KEY` for USART2.

`: seremit2 \ char --`
`EMIT` for USART2.

`: serType2 \ c-addr len --`
`TYPE` for USART2.

`: sercr2 \ --`
`CR` for USART1.

```
create Console2 \ -- addr ; OUT managed by upper driver
```
Device structure for USART2

```
  ' serkey2 ,           \ -- char ; receive char
  ' serkey?2 ,          \ -- flag ; check receive char
  ' seremit2 ,          \ -- char ; display char
  ' sertype2 ,          \ caddr len -- ; display string
  ' sercr2 ,            \ -- ; display new line


  console2 constant console
```
Device structure for console on USART1.

11.3.5 USART3

USART3 is on the APB1 bus, so the baud rate is calculated using that bus clock. This code is only compiled if the equate `useUSART3` is non-zero}.

`console3-speed APB1-speed genBRRval equ /us3BRR \ -- u`
Baud rate divisor for USART3.

`: selio-ser3 \ --`
Example to perform clock and pin selection for USART3 using GPIO pins PD8/9.

`: init-ser3 \ --`
Initialise USART3 for polled operation on GPIO pins PD8/9.

`: serkey?3 \ -- flag`
`KEY?` for USART3.

`: serkey3 \ -- char`
`KEY` for USART3.

`: seremit3 \ char --`
`EMIT` for USART3.

```
: serType3      \ c-addr len --
```
TYPE for USART3.

```
: sercr3        \ --
```
CR for USART3.

```
create Console3 \ -- addr ; OUT managed by upper driver
```
Device structure for USART3.

```
  ' serkey3 ,           \ -- char ; receive char
  ' serkey?3 ,          \ -- flag ; check receive char
  ' seremit3 ,          \ -- char ; display char
  ' sertype3 ,          \ caddr len -- ; display string
  ' sercr3 ,            \ -- ; display new line


  console3 constant console
```
Device structure if console is on USART3.

11.3.6 Initialisation

```
: init-ser      \ --
```
Initialise primary serial ports.

11.4 System Ticker

The source code discussed in this section may be found in the file *Drivers/SysTickDisco072.fth*.

The system ticker uses the Cortex-M0 *SysTick* timer. Although the *SysTick* timer is common to all Cortex-M0 devices, there is no guarantee that they have the same clocking arrangements. Consequently there are different drivers for different implementations.

The STM32 implementation is fed with the system clock and has an option to divide it by 8, controlled by bit 2 of the SysTick Control and Status Register (ARM DDI 0337, Rev E, page 8-8).

- Bit2=0: SysTick clock is HCLK/8.
- Bit2=1: SysTick clock is HCLK.

This code always uses HCLK/8.

```
variable LedActive      \ -- flag
```
If LedActive contains non-zero, the ticker is used to produce a rotating pattern on the four LEDs.

```
: ticks         \ -- ms
```
Returns a 32 bit number of milliseconds that eventually wraps.

```
PC 6 pio: LDU   \ -- struct
```
Red/Upper LED.

```
PC 9 pio: LDR   \ -- struct
```
Green/Right LED.

```
PC 7 pio: LDD   \ -- struct
```
Blue/Down LED.

`PC 8 pio: LDL  \ -- struct`
Orange/Left LED.

`: SysTicker  \ --`
Ticker ISR action.

`' SysTicker SysTickvec# EXC: SysTickISR \ -- addr`
Setting the high level ISR.

`: start-clock  \ --`
Initialise the system ticker to run with a period of `tick-ms` milliseconds.

`: stop-clock  \ --`
Stop the sytem ticker.

`: later  \ n -- n'`
Generates the timebase value for termination in n millseconds time.

`: timedout?  \ n -- flag ; true if timed out`
Flag is returned true if the timebase value n has timed out. `TIMEDOUT?` does not call `PAUSE`.

`: ms  \ n --`
Waits for n milliseconds.

11.5 Rebooting the CPU

The word `REBOOT` permits the system to be reset by disabling all interrupts and activating the watchdog.

`: reboot  \ --`
Reboots the CPU by activating the watchdog.

12 L3GD20 MEMS Gyro

The STM32F072B Discovery board includes an L3GD20 3-axis MEMS gyro. The ST documentation is a bit sparse. We use the SPI2 peripheral to talk to the gyro. The MEMS code is in the file *Examples\l3gd20.fth* and the SPI driver is in the file *spiSTM32F0hard.fth*.

```
PC 1 pio: MEMS_INT1     \ -- struct
```
MEMS INT1 connected here.

```
PC 2 pio: MEMS_INT2
```
MEMS INT2 connected here.

```
: L3GD20_Write1 \ b reg --
```
Write one byte to the given register.

```
: L3GD20_Read1  \ reg -- b
```
Read one byte from the given register.

```
: L3GD20_ReadN  \ caddr len reg --
```
Read *len* bytes to *caddr* starting from the given register.

```
: setMaskL3     \ value mask reg --
```
Clear the *mask* bits in *reg*. Write *value* into the mask bits in the given L3GD20 register.

```
: borL3         \ mask reg --
```
Set the bits in mask in the given register.

```
: bbicL3                \ mask reg --
```
Clear the bits in mask in the given register.

```
: L3GD20_RebootCmd      \ --
```
Reboot memory content of L3GD20.

```
: L3GD20_GetDataStatus  \ -- b
```
Get status for L3GD20 data.

```
: L3GD20_EXTI_Config    \ --
```
Configure the INT1 MEMS Interrupt line and GPIO in EXTI mode. The GPIO and SPI blocks are already enabled.

```
: L3GD20_INT_Config     \ --
```
Configure L3GD20 interrupts.

```
: init-L3GD20   \ --
```
Initialise L3GD20.

```
: L3GD20_INT1InterruptCmd       \ flag --
```
Enable or disable INT1 interrupt.

```
: L3GD20_INT2InterruptCmd       \ --
```
Enable or disable INT2 interrupt.

12.1 Test and Demo code

```
: tr      \ --
```
Display the L3GD20 identification register.

```
6 buffer: AxisData      \ -- addr
```
Buffer for axis data.

`: ReadAxes \ --`
Read the axes into the `AxisData` buffer. The data consists of three 16 bit signed values for the X, Y and Z axes.

`: .Axes \ --`
Display the axis data.

`: ReadTest \ --`
Display the axis data until a key is pressed.

`PA 0 pio: UserButton \ -- struct`
Pin definition for the User/Wake button on PA0.

`: LedsOff \ --`
Turn off the four LEDs.

`: Xval \ -- n`
Return the X value from the last reading of the axes.

`: Yval \ -- n`
Return the Y value from the last reading of the axes.

`: Zval \ -- n`
Return the Z value from the last reading of the axes.

`: onlyLED \ struct`
Turn on the given LED only and wait 250 ms.

`: DemoMEMS \ --`
Demo for the MEMS sensor.

Index

>

?

@

[

]

\

0

1

2

6

7

8

9

A

B

C

D

E

F

G

H

S

T

U

V

W

X

Y

Z