

# MPE ARM USB Stamp

---

v1.0



MPE ARM USB Stamp  
User manual  
Manual revision 1.00  
6 September 2004

Software  
Software version 6.20

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@mpeltd.demon.co.uk](mailto:mpe@mpeltd.demon.co.uk)  
[tech-support@mpeltd.demon.co.uk](mailto:tech-support@mpeltd.demon.co.uk)  
web: [www.mpeltd.demon.co.uk](http://www.mpeltd.demon.co.uk)

# Table of Contents

<b>1</b>	<b>Introduction .....</b>	<b>1</b>
1.1	About the USB Stamp board.....	1
1.2	About Forth .....	1
1.3	About the manual.....	1
1.4	Getting started.....	2
1.5	If disaster strikes.....	3
1.6	Using other terminal emulators.....	4
1.7	Technical support .....	4
1.8	Licensing.....	4
<b>2</b>	<b>ARM code definitions .....</b>	<b>7</b>
2.1	Notes.....	7
2.2	Register usage .....	7
2.3	Configuration .....	7
2.4	Logical and relational operators .....	7
2.5	Control flow .....	9
2.6	Arithmetic.....	9
2.7	Stack manipulation.....	12
2.8	String and memory operators .....	13
2.9	Miscellaneous words .....	15
2.10	Portability helpers.....	15
2.11	Runtime for VALUE.....	16
2.12	Defining words and runtime support .....	16
2.13	Structure compilation.....	17
2.14	Branch constructors .....	17
2.15	Main structure compilers .....	18
2.16	Miscellaneous .....	18
<b>3</b>	<b>High level kernel KERNEL62.FTH .....</b>	<b>21</b>
3.1	User variables.....	21
3.2	System Constants .....	22
3.3	System VARIABLEs and Buffers .....	22
3.3.1	Variables.....	22
3.4	Deferred words .....	22
3.5	Predefined Vocabularies.....	23
3.6	Vectored I/O handling .....	23
3.6.1	Introduction.....	23
3.6.2	Building a vector table.....	23
3.6.3	Generic I/O words.....	23
3.7	String and memory operations.....	24
3.8	Dictionary management.....	25
3.9	String compilation.....	26
3.10	Pre-ANS Exception handlers .....	26
3.11	ANS words CATCH and THROW.....	27
3.11.1	Example implementation.....	27
3.11.2	Example use .....	28
3.11.3	Gotchas.....	29

3.12	Formatted and unformatted i/o .....	29
3.12.1	Setting number bases .....	29
3.12.2	Numeric output .....	30
3.12.3	Numeric input .....	30
3.13	String input and output .....	31
3.14	Source input control .....	31
3.15	Text scanning .....	32
3.16	Miscellaneous .....	32
3.17	Wordlist control .....	33
3.18	Control structures .....	34
3.19	Target interpreter and compiler .....	35
3.20	Compilation and Caches .....	37
3.21	Startup code .....	38
3.21.1	Cold chain .....	38
3.21.2	The COLD sequence .....	38
3.22	Kernel error codes .....	38
3.23	Differences between the v6.1 and 6.2 kernels .....	39
3.23.1	Error handling .....	39
3.23.2	Terminal input buffer and ACCEPT .....	40
<b>4</b>	<b>Target VALUE and local variables .....</b>	<b>41</b>
<b>5</b>	<b>Development tools .....</b>	<b>43</b>
<b>6</b>	<b>Debugging tools .....</b>	<b>45</b>
6.1	Implementation dependencies .....	45
6.2	Miscellaneous .....	46
6.3	Stack checking .....	47
6.4	Assertions .....	48
<b>7</b>	<b>Interrupt handlers .....</b>	<b>51</b>
7.1	Configuration .....	52
7.2	Interrupt management .....	52
7.3	SWI handler .....	53
7.4	Support for complex abort handlers .....	53
7.5	Undefined instruction handler .....	54
7.5.1	Simple UNDEF handler .....	54
7.5.2	Complex UNDEF handler .....	55
7.6	Prefetch Abort handler .....	55
7.6.1	Simple PABORT handler .....	55
7.6.2	Complex PAbort handler .....	55
7.7	Data Abort handler .....	55
7.7.1	Simple DAbort handler .....	55
7.7.2	Complex DAbort handler .....	56
7.8	Reserved (26 bit address exception) handler .....	56
7.9	Generic IRQ handler .....	56
7.10	Generic FIQ handler .....	57
7.11	AT91 IRQ and FIQ handlers .....	57
7.12	Samsung S3C4510 IRQ and FIQ handlers .....	59
7.13	ARM PL190 IRQ and FIQ handlers .....	59
<b>8</b>	<b>Character Queues .....</b>	<b>63</b>

<b>9</b>	<b>Serial driver</b> .....	<b>65</b>
9.1	Configuration .....	65
9.2	Serial interrupt service routines .....	65
9.3	Initialisation .....	65
9.4	Serial primitives .....	65
9.5	Generic i/o assignments .....	66
9.6	Forth Stamp specific code .....	66
<b>10</b>	<b>GPIO initialisation and USB driver</b> .....	<b>67</b>
10.1	GPIO initialisation .....	67
10.2	USB comms driver .....	68
<b>11</b>	<b>LPC software I2C driver</b> .....	<b>69</b>
11.1	Introduction .....	69
11.2	Timing .....	69
11.3	I2C bit functions .....	69
<b>12</b>	<b>I2C generic primitives</b> .....	<b>71</b>
<b>13</b>	<b>AT24C512 I2C driver</b> .....	<b>73</b>
13.1	Introduction .....	73
13.2	24C512 primitives .....	73
13.3	Simple utilities .....	73
13.4	EDSL primitives .....	74
13.5	EDSL functions .....	74
<b>14</b>	<b>Software Floating Point</b> .....	<b>75</b>
14.1	Introduction .....	75
14.2	Source code .....	75
14.3	Entering floating-point numbers .....	75
14.4	The form of floating-point numbers .....	75
14.5	Creating variables .....	75
14.6	Accessing variables .....	76
14.7	Creating constants .....	76
14.8	Using the supplied words .....	76
14.8.1	Calculating sines, cosines and tangents .....	76
14.8.2	Calculating arc sines, cosines and tangents .....	76
14.8.3	Calculating logarithms .....	76
14.8.4	Calculating powers .....	77
14.9	Degrees or radians .....	77
14.10	Displaying floating-point numbers .....	77
14.11	Changes from v6.0 to v6.1 .....	77
14.11.1	32 bit targets: software floating point .....	77
14.11.2	16 bit targets: software floating point .....	78
14.12	Glossary .....	78
14.12.1	Basic stack and memory operators .....	78
14.12.2	Floating point defining words .....	79
14.12.3	Type conversions .....	79
14.12.4	Arithmetic .....	79
14.12.5	Relational operators .....	80
14.12.6	Rounding .....	80
14.12.7	Miscellaneous .....	81

14.12.8	Floating point output .....	81
14.12.9	Floating point input .....	82
14.12.10	Trigonometric functions .....	83
14.12.11	Power and logarithmic functions .....	84
14.13	ARM coded primitives .....	84
<b>15</b>	<b>Periodic Timers .....</b>	<b>87</b>
15.1	The basics of timers .....	87
15.2	Considerations when using timers .....	87
15.3	Implementation issues .....	88
15.4	Timebase glossary .....	88
<b>16</b>	<b>LPC210x Ticker Interrupt .....</b>	<b>91</b>
16.1	Configuration equates .....	91
16.2	Ticker interrupt handler .....	91
16.3	Time base extensions .....	92
<b>17</b>	<b>ARM multitasker .....</b>	<b>93</b>
17.1	Configuration - normally performed earlier .....	93
17.2	TCB data structure layout .....	93
17.3	Task handling primitives .....	93
17.4	Event handling .....	93
17.5	Message handling .....	94
17.6	Task structure management .....	94
17.7	semaphores .....	95
17.8	TASK and START: .....	95
17.9	Debugging tools .....	96
<b>18</b>	<b>Vocabulary and wordlist tools .....</b>	<b>97</b>
<b>19</b>	<b>ROM PowerForth utilities .....</b>	<b>99</b>
19.1	Introduction .....	99
19.2	Compiling text files .....	99
19.2.1	The required files .....	99
19.2.2	Compiling a specified text file .....	99
19.3	XMODEM binary image download .....	99
19.4	XMODEM binary image upload .....	100
19.5	IODEF.FTH .....	100
19.5.1	AIDE support .....	100
19.6	Miscellaneous .....	100
19.7	INCLUDE source code from AIDE .....	100
19.8	Simple source file loader .....	101
<b>20</b>	<b>XMODEM Receiver and Transmitter .....</b>	<b>103</b>
20.1	Introduction .....	103
20.2	Words in XMODEMTXRX.FTH .....	103
20.2.1	Configuration .....	103
20.2.2	Constants and variables .....	103
20.2.3	Common code .....	103
20.2.4	XMODEM transmission .....	103
20.2.5	XMODEM reception .....	104

<b>21</b>	<b>Access to the IAP routines .....</b>	<b>105</b>
	21.1 Gotchas .....	105
<b>22</b>	<b>LPC2000 Flash tools .....</b>	<b>107</b>
	22.1 Flash primitives .....	107
	22.2 Flash driver .....	107
<b>23</b>	<b>Reflashing the system .....</b>	<b>109</b>
	23.1 Introduction .....	109
	23.2 Code in main application .....	109
<b>24</b>	<b>Rebooting the CPU .....</b>	<b>111</b>
<b>25</b>	<b>Creating turnkey applications .....</b>	<b>113</b>
	25.1 Introduction .....	113
	25.2 Saving applications .....	113
	25.3 Gotchas .....	114
	25.4 Application License .....	114
<b>26</b>	<b>Examples directory .....</b>	<b>117</b>
	26.1 Main directory .....	117
	26.2 Contributions subdirectory .....	117
	26.3 I2C subdirectory .....	118
	26.4 SPI subdirectory .....	118





# 1 Introduction

This manual documents the MPE PowerForth system supplied with your USB ARM Stamp. The USB ARM Stamp hardware is documented separately. PDF files in the DOCS folder are provided for the circuit diagram, component layout and the default CPLD schematic.

## 1.1 About the USB Stamp board

The MPE ARM USB Stamp consists of several main blocks:

<b>Power</b>	All power is taken from the USB port. On board regulators generate 3.3 and 1.8 volt supplies.
<b>CPU</b>	Philips LPC2106 with 128k Flash and 64k RAM,
<b>USB</b>	FTDI FT245BM provides a fast comms link to the host PC, Mac or Linux machine.
<b>CPLD</b>	Xilinx XC32/64 which is user programmable using the Xilinx WebPack software, downloadable free from <a href="http://www.xilinx.com">www.xilinx.com</a> or on CD for a minimal cost.
<b>EEPROM</b>	Atmel AT24C512 (24C128 on prototypes). This can be used for program storage and for configuration.

The hardware is complemented by its default software. The MPE PowerForth system provided with the board in the first 64k of the Flash contains a Forth compiler and interpreter, multitasker, timebase, floating point, comms utilities, flash utilities and maintenance tools.

## 1.2 About Forth

Forth is an interactive programming language widely used for embedded systems ranging from bomb disposal machines to embedded web servers, seismic data loggers and safety critical medical equipment. The DOCS folder on the CD includes a Forth primer. Also included on the CD is an evaluation version of MPE's VFX Forth for Windows. The latest version is available for free download from

<http://www.mpeltd.demon.co.uk>

To run VFX Forth for Windows, send an email with your name, address and contact details to: <mailto://vfxtrial@mpeltd.demon.co.uk>

An installation key will then be provided.

## 1.3 About the manual

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

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

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

## 1.4 Getting started

Run the installer on the CD. This will prompt you for a directory/folder into which it will install all the issue files. It will also add a group called "MPE ARM USB Stamp" on your Start menu. This includes a short cut to AIDE (see later).

Install the Philips ISP programmer software from the USBSTAMP\PHILIPSISP folder. This requires a serial connection to the DB9 connector on the board. To use it, the link marked BOOT on the board must be fitted. To run all other software this link must be open. The Philips ISP software is only needed if the on-board Forth software becomes corrupted.

N.B. If you have problems with the on-board Flash programming routines, check the LPC2106 bootloader version using the Philips ISP software or by typing

```
IAPBootVer .dword
```

which will give something of the form:

```
0000:xxyy
```

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

```
www.semiconductors.philips.com
/files/products/standard/microcontrollers/utilities/
lpc2000_flash_utility.zip
lpc2000_bl_update.zip
```

Note that v1.52 is only for the LPC2104/5/6 and v1.63 is required for other parts such as the LPC2119/2129. A PDF file in the update describes how to perform the update.

Install the FTDI Windows USB drivers from the CD if not already installed. If you are using a Mac or a Linux host, drivers are available free of charge from:

```
www.ftdichip.com
```

The Windows driver makes a USB connection appear as a COM port. To install it:

- . Unzip the ZIP file in the USBDRIVERS folder to a new folder.
- . Connect the USB Stamp to a USB port.
- . If Windows asks you for a driver, point Windows to the folder you created in the first step. The driver will then install. In some cases you may have to run the "Add New Hardware/Programs" wizard. The FTDI application note AN232-03.PDF in the USBDRIVERS folder describes the process in more detail.

AIDE is an Integrated Development Environment (IDE) that includes a simple editor for your

source code and a terminal emulator (PowerTerm) tuned for use with the PowerForth on the board. Use the Properties button on the PowerTerm toolbar to select the COM port. The baud rate is irrelevant for the USB stamp board, but we normally set it to 115200. Note that the USB COM port is not available until the board has been connected. On the Properties -> Configure Console Window page, ensure that the Enable File Server box is checked. When AIDE is closed, these settings will become the defaults next time.

The board communicates to the host via the USB COM port mechanism provided by the FTDI drivers. Connect the board to a USB port, which also provides the power for it. Press the Connect button on the PowerTerm toolbar. Reset the board using the little button on the side. PowerForth will sign on.

Commands typed directly into the Forth interpreter do not execute until the ENTER/CR key is pressed.

Write a simple Forth word, e.g.

```
: hello    \ --  
  cr cr ." Hello, world!" cr  
;
```

Execute it:

```
hello
```

It will run. You can put the same code in a text file, conventionally with a .FTH extension such as hello.fth. Compile the file (using AIDE and PowerTerm) with:

```
include hello.fth
```

The file will be compiled on the board and you can execute the word by typing HELLO again.

Save the compiled image:

```
0 turnkey
```

Either reset the board using the reset button or by typing:

```
reboot
```

The board will reboot, and the word HELLO will already be part of the system.

You can clear out your previous work by typing EMPTY and rebooting:

```
empty reboot
```

## 1.5 If disaster strikes

If you get the board into a bad state and it will not sign on, you may need to reload the kernel program. Reprogram the board using the Philips ISP utility. The file to load is BINARIES\USBSTAMP.HEX.

If the board still misbehaves, reload the flash with BINARIES\USBRECOVER.HEX and run the board. This empties the serial EEPROM before signing on. Once you have seen the recovery messages and PowerForth has signed on, you can use

```
reflash
```

to reload USBSTAMP.IMG and carry on in the normal way.

## 1.6 Using other terminal emulators.

AIDE and PowerTerm are designed for use with PowerForth and include a source file server. If you prefer, you can use other terminal emulators, but will lose some facilities.

Set HyperTerm or another terminal emulator to 115200 baud, 8 data bits, no parity, 1 or 2 stop bits. Select the relevant COM port for the USB ARM Stamp and reset it. It will sign on. If it does not sign on, repeat the process with a serial cable that has pins 2 and 3 swapped, e.g. a null modem cable. If all else fails, reflash the system as described elsewhere in this manual.

Please be aware that the standard Windows version of HyperTerm is very slow. A much faster alternative is HyperTerminal Personal Edition from:

<http://www.hilgraeve.com>

## 1.7 Technical support

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

```
tel: +44 (0)23 8063 1441
fax: +44 (0)23 8033 9691
net: mpe@mpeltd.demon.co.uk
     tech-support@mpeltd.demon.co.uk
web: www.mpeltd.demon.co.uk
```

From North America, our telephone and fax numbers are:

```
011 44 23 8063 1441
011 44 23 8033 9691
```

## 1.8 Licensing

You may only use the supplied code with boards manufactured by MicroProcessor Engineering Ltd. and New Micros Inc. If you want to distribute the code you have two options.

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



## 2 ARM code definitions

### 2.1 Notes

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

```
n EQU <name>
```

```
PROC <name>
```

```
L: <name>
```

### 2.2 Register usage

On the ARM the following register usage is the default:

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

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

### 2.3 Configuration

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

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

### 2.4 Logical and relational operators

```
CODE AND \ x1 x2 -- x3
```

Perform a logical AND between the top two stack items and retain the result in top of stack.

```
CODE OR \ x1 x2 -- x3
```

Perform a logical OR between the top two stack items and retain the result in top of stack.

```
CODE XOR \ x1 x2 -- x3
```

Perform a logical XOR between the top two stack items and retain the result in top of stack.

```
CODE NOT \ x -- x'
```

Perform a bitwise NOT on the top stack item and retain result.

```
CODE INVERT \ x -- x'
```

Perform a bitwise NOT on the top stack item and retain result.



```

CODE 0=      \ x -- flag
Compare the top stack item with 0 and return TRUE if equals.
CODE 0<>     \ x -- flag
Compare the top stack item with 0 and return TRUE if not-equal.
CODE 0<      \ x -- flag
Return TRUE if the top of stack is less-than-zero.
CODE 0>      \ x -- flag
Return TRUE if the top of stack is greater-than-zero.
CODE =       \ x1 x2 -- flag
Return TRUE if the two topmost stack items are equal.
CODE <>      \ x1 x2 -- flag
Return TRUE if the two topmost stack items are different.
CODE <       \ n1 n2 -- flag
Return TRUE if n1 is less than n2.
CODE >       \ n1 n2 -- flag
Return TRUE if n1 is greater than n2.
CODE <=      \ n1 n2 -- flag
Return TRUE if n1 is less than or equal to n2.
CODE >=      \ x1 x2 -- flag
Return TRUE if n1 is greater than or equal to n2.
CODE U>      \ u2 u2 -- flag
An UNSIGNED version of >.
CODE U<      \ u1 u2 -- flag
An UNSIGNED version of <.
CODE DU<     \ ud1 ud2 -- flag
Returns true if ud1 (unsigned double) is less than ud2.
CODE D0<     \ d -- flag
Returns true if signed double d is less than zero.
CODE D0=     \ xd -- flag
Returns true if xd is 0.
CODE D=      \ xd1 xd2 -- flag
Return TRUE if the two double numbers are equal.
CODE D<      \ d1 d2 -- flag
Return TRUE if the double number d1 is < the double number d2.
CODE DMAX    \ d1 d2 -- d3 ; d3=max of d1/d2
Return the maximum double number from the two supplied.
CODE DMIN    \ d1 d2 -- d3 ; d3=min of d1/d2
Return the minimum double number from the two supplied.
CODE MIN     \ n1 n2 -- n1|n2
Given two data stack items preserve only the smaller.
CODE MAX     \ n1 n2 -- n1|n2
Given two data stack items preserve only the larger.

```

CODE WITHIN? \ n1 n2 n3 -- flag

Return TRUE if N1 is within the range N2..N3. This word uses signed arithmetic.

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

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

CODE LSHIFT \ x1 u -- x2

Logically shift X1 by U bits left.

CODE RSHIFT \ x1 u -- x2

Logically shift X1 by U bits right.

CODE << \ x1 u -- x1<<u

Logically shift X1 by U bits left. Obsolete - replaced by LSHIFT.

CODE >> \ x1 u -- x1<<u

Logically shift X1 by U bits right. Obsolete - replaced by RSHIFT.

## 2.5 Control flow

CODE EXECUTE \ xt --

Execute the code described by the XT. This is a Forth equivalent to an assembler JSR/CALL instruction.

CODE BRANCH \ --

The run time action of unconditional branches compiled on the target. INTERNAL.

CODE ?BRANCH \ n --

The run time action of conditional branches compiled on the target. INTERNAL.

CODE (OF) \ n1 n2 -- n1|--

The run time action of OF compiled on the target. INTERNAL.

CODE (LOOP) \ --

The run time action of LOOP compiled on the target. INTERNAL.

CODE (+LOOP) \ n --

The run time action of +LOOP compiled on the target. INTERNAL.

CODE (DO) \ limit index --

The run time action of DO compiled on the target. INTERNAL.

CODE (?DO) \ limit index --

The run time action of ?DO compiled on the target. INTERNAL.

CODE LEAVE \ --

Remove the current DO..LOOP parameters and jump to the end of the DO..LOOP structure.

CODE ?LEAVE \ flag --

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

CODE I \ -- n

Return the current index of the inner-most DO..LOOP.

CODE J \ -- n

Return the current index of the second DO..LOOP.

CODE UNLOOP \ -- ; R: loop-sys --

Remove the DO..LOOP control parameters from the return stack.

## 2.6 Arithmetic

CODE S>D            \ n -- d

Convert a single number to a double one.

CODE D>S            \ d -- n

Convert a double number to a single.

CODE NOOP           \ --

A NOOP, null instruction. )

proc umul32\*32 \ tos:r0 = r0 \* tos, corrupts r1, r2, r3

The unsigned 32 \* 32 -> 64 bit multiply primitive for CPUs without the long multiply instructions.

CODE UM\*            \ u1 u2 -- ud

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

CODE \*              \ n1 n2 -- n3

Standard signed multiply. N3 = n1 \* n2.

CODE m\*             \ n1 n2 -- d

Signed multiply yielding double result.

proc udiv64/32 \ r0:r1 / tos ; 64/32 unsigned divide -> tos=quot, r0=rem

Division subroutine - unrolled for a bit of extra speed.

CODE UM/MOD        \ ud un -- urem uquot

Perform unsigned division of double number UD by single number U and return remainder and quotient.

: MU/MOD           \ d n -- rem d#quot

Perform an unsigned divide of a double by a single, returning a single remainder and a double quotient.

CODE FM/MOD        \ d1 n2 -- rem quot ; floored division

Perform a signed division of double number D1 by single number N2 and return remainder and quotient using floored division. See the ANS Forth specification for more details of floored division.

CODE SM/REM        \ d1 n2 -- rem quot ; symmetric division

Perform a signed division of double number D1 by single number N2 and return remainder and quotient using symmetric (normal) division.

CODE /MOD          \ n1 n2 -- rem quot

Signed division of N1 by N2 single-precision yielding remainder and quotient.

CODE M/MOD         \ d1 n2 -- rem quot

A synonym for FM/MOD for Forth-83 compatibility. Obsolete - Use FM/MOD instead.

: /                 \ n1 n2 -- n3

Standard signed division operator. n3 = n1/n2.

: MOD              \ n1 n2 -- n3

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

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

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

: \*/                \ n1 n2 n3 -- n4

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

```
: M/          \ d n1 -- n2
```

Signed divide of a double by a single integer.

```
CODE TNEGATE  \ t1 -- -t1 ; negate value of triple cell number
```

Negate a triple (3 cell) number. INTERNAL.

```
: M*/        \ d1 n1 +n2 -- d2
```

Multiply double  $d1$  by single  $n1$  to give a triple precision result, and then divide it by  $n2$  returning the quotient. The point of this operation is to avoid loss of precision.

```
CODE M+      \ d1|ud1 n -- d2|ud2
```

Add double  $d1$  to sign extended single  $n$  to form double  $d2$ .

```
CODE 1+      \ n1|u1 -- n2|u2
```

Add one to top-of stack.

```
CODE 2+      \ n1|u1 -- n2|u2
```

Add two to top-of stack.

```
CODE 4+      \ n1|u1 -- n2|u2
```

Add four to top-of stack.

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

Subtract one from top-of stack.

```
CODE 2-      \ n1|u1 -- n2|u2
```

Subtract two from top-of stack.

```
CODE 4-      \ n1|u1 -- n2|u2
```

Subtract four from top-of stack.

```
CODE 2*      \ x1 -- x2
```

Signed multiply top of stack by 2.

```
CODE 4*      \ x1 -- x2
```

Signed multiply top of stack by 4.

```
CODE 2/      \ x1 -- x2
```

Signed divide top of stack by 2.

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

Unsigned divide top of stack by 2.

```
CODE 4/      \ x1 -- x2
```

Signed divide top of stack by 4.

```
CODE U4/     \ x1 -- x2
```

Unsigned divide top of stack by 4.

```
CODE +       \ n1|u1 n2|u2 -- n3|u3
```

Add two single precision integer numbers.

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

Subtract two single precision integer numbers.  $N3|u3=n1|u1-n2|u2$ .

```
CODE NEGATE  \ n1 -- n2
```

Negate a single precision integer number.

```
CODE D+      \ d1 d2 -- d3
```

Add two double precision integers.

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

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

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

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

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

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

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

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

## 2.7 Stack manipulation

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

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

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

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

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

CODE -ROT          \ x1 x2 x3 -- x3 x1 x2  
The inverse of ROT.

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

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

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

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

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

CODE 2R@           \ -- x1 x2 ; R: x1 x2 -- x1 x2

Copy the top two return stack items to the data stack.

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

Perform ROT operation on three double numbers.

```
CODE SWAP     \ x1 x2 -- x2 x1
```

Exchange the top two data stack items.

```
CODE DUP      \ x -- x x
```

DUPLICATE the top stack item.

```
CODE OVER     \ x1 x2 -- x1 x2 x1
```

Copy NOS to a new top-of-stack item.

```
CODE DROP     \ x --
```

Lose the top data stack item and promote NOS to TOS.

```
CODE 2DROP    \ x1 x2 -- )
```

Discard the top two data stack items.

```
CODE 2SWAP    \ x1 x2 x3 x4 -- x3 x4 x1 x2
```

Exchange the top two cell-pairs on the data stack.

```
CODE ?DUP     \ x -- | x
```

Duplicate the top stack item only if it non-zero.

```
CODE 2DUP     \ x1 x2 -- x1 x2 x1 x2
```

DUPLICATE the top cell-pair on the data stack.

```
CODE 2OVER    \ x1 x2 x3 x4 -- x1 x2 x3 x4 x1 x2
```

Similar to OVER but works with cell-pairs rather than cell items.

```
CODE SP@     \ -- x
```

Get the current address value of the data-stack pointer.

```
CODE SP!     \ x --
```

Set the current address value of the data-stack pointer.

```
CODE RP@     \ -- x
```

Get the current address value of the return-stack pointer.

```
CODE RP!     \ x --
```

Set the current address value of the return-stack pointer.

## 2.8 String and memory operators

```
CODE COUNT    \ c-addr1 -- c-addr2' u
```

Given the address of a counted string in memory this word will return the address of the first character and the length in characters of the string.

```
CODE /STRING  \ c-addr1 u1 n -- c-addr2 u2
```

Modify a string address and length to remove the first N characters from the string.

```
CODE SKIP     \ c-addr1 u1 char -- c-addr2 u2
```

Modify the string description by skipping over leading occurrences of 'char'.

```
CODE SCAN     \ c-addr1 u1 char -- c-addr2 u2
```

Look for first occurrence of CHAR in string and return new string. C-addr2/u2 describe the string with CHAR as the first character.

```
CODE S=       \ c-addr1 c-addr2 u -- flag
```

Compare two same-length strings/memory blocks, returning TRUE if they are identical.

```

: compare      \ c-addr1 u1 c-addr2 u2 -- n                17.6.1.0935
Compare two strings. The return result is 0 for a match or can be -ve/+ve indicating string
differences. If the two strings are identical, n is zero. If the two strings are identical up to the
length of the shorter string, n is minus-one (-1) if u1 is less than u2 and one (1) otherwise. If
the two strings are not identical up to the length of the shorter string, n is minus-one (-1) if
the first non-matching character in the string specified by c-addr1 u1 has a lesser numeric value
than the corresponding character in the string specified by c-addr2 u2 and one (1) otherwise.

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

CODE CMOVE    \ c-addr1 c-addr2 u --
Copy U bytes of memory forwards from C-ADDR1 to C-ADDR2.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

CODE W!      \ w a-addr --

```

Store the word (16 bit) quantity w at memory ADDR.

CODE C!            \ char c-addr --

Store the character CHAR at memory C-ADDR.

CODE UPPER        \ c-addr u --

Convert the ASCII string described to upper-case. This operation happens in place.

CODE TEST-BIT    \ mask c-addr -- flag

AND the mask with the contents of addr and return true if the result is non-zero (-1) or false (0) if the result is zero. Byte operation.

CODE SET-BIT     \ mask c-addr --

Apply the mask ORred with the contents of c-addr. Byte operation.

CODE RESET-BIT  \ mask c-addr --

Apply the mask inverted and ANDed with the contents of c-addr. Byte operation.

CODE TOGGLE-BIT \ u c-addr --

Invert the bits at c-addr specified by the mask. Byte operation.

## 2.9 Miscellaneous words

CODE NAME>       \ nfa -- cfa

Move a pointer from an NFA to the CFA or "XT" in ANS parlance.

CODE >NAME       \ cfa -- nfa

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

: SEARCH-WORDLIST    \ c-addr u wid -- 0|xt 1|xt -1

Search the given wordlist for a definition. If the definition is not found then 0 is returned, otherwise the XT of the definition is returned along with a non-zero code. A -ve code indicates a "normal" definition and a +ve code indicates an IMMEDIATE word.

CODE DIGIT        \ char n -- 0|n true

If the ascii value CHAR can be treated as a digit for a number within the radix N then return the digit and a TRUE flag, otherwise return FALSE.

## 2.10 Portability helpers

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

CODE CELL+        \ a-addr1 -- a-addr2

Add the size of a CELL to the top-of stack.

CODE CELLS        \ n1 -- n2

Return the size in address units of N1 cells in memory.

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

Decrement an address by the size of a cell.

CODE CELL         \ -- n

Return the size in address units of one CELL.

CODE CHAR+        \ c-addr1 -- c-addr2

Increment an address by the size of a character.



CODE CHARS        \ n1 -- n2

Return size in address units of N1 characters.

## 2.11 Runtime for VALUE

CODE VAL!        \ n -- ; store value address in-line

Store n at the inline address following this word. INTERNAL.

CODE VAL@        \ -- n ; read value data address in-line

Read n from the inline address following this word. INTERNAL.

## 2.12 Defining words and runtime support

L: DOCREATE      \ -- addr

The run time action of CREATE. INTERNAL.

CODE LIT         \ -- x

Code which when CALLED at runtime will return an inline cell value. INTERNAL.

CODE (")         \ -- a-addr ; return address of string, skip over it

Return the address of a counted string that is inline after the CALLING word, and adjust the CALLING word's return address to step over the inline string. See the definition of (".) for an example. INTERNAL.

: aligned        \ addr -- addr'

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

: CONSTANT      \ x "<spaces>name" -- ; Exec: -- x

Create a new CONSTANT called "name" which has the value "x". When "NAME" is executed the value is returned on the top-of-stack.

: VARIABLE      \ "<spaces>name" -- ; Exec: -- a-addr

Create a new variable called "name". When "Name" is executed the address of the data-cell is returned for use with @ and ! operators.

: USER          \ u "<spaces>name" -- ; Exec: -- addr ; SFP009

Create a new USER variable called "name". The 'u' parameter specifies the index into the user-area table at which to place the data. USER variables are located in a separate area of memory for each task or interrupt. Use in the form: "\$400 USER TaskData".

: DOCOLON,       \ --

Compile the runtime entry code required by colon definitions. INTERNAL.

: !call          \ dest addr --

Install a BL DEST opcode at addr.

: scall,         \ addr --

Compile a machine code BL to addr. No range checking is performed.

: compile,       \ xt --

Compile the word specified by xt into the current definition.

: call,          \ dest --

Compile a BL or long CALL into the current definition.

: >BODY          \ xt -- a-addr

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

```
: :          \ C: "<spaces>name" -- colon-sys ; Exec: i*x -- j*x ; R: -- nest-sys
```

Begin a new definition called "name".

```
: :NONAME    \ C: -- colon-sys ; Exec: i*x -- i*x ; R: -- nest-sys
```

Begin a new code definition which does not have a name. After the definition is complete the semi-colon operator returns the XT of newly compiled code on the stack.

```
: DOCREATE,  \ --
```

Compile the run time action of CREATE. INTERNAL.

```
: (;CODE)    \ -- ; R: a-addr --
```

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

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

Begin definition of the runtime-action of a child of a defining word. See the section about defining words in the Five minute Forth chapter. You should not use RECURSE after DOES>.

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

The default action of a DEFERred word. This will simply THROW a code back to the system.

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

Creates a new DEFERred word. A default action, CRASH, is automatically assigned.

```
: 2CONSTANT  \ Comp: x1 x2 "<spaces>name" -- ; Run: -- x1 x2
```

A two-cell equivalent to CONSTANT.

```
: 2VARIABLE  \ Comp: "<spaces>name" -- ; Run: -- a-addr
```

A two-cell equivalent to VARIABLE.

```
: FIELD      \ size n "<spaces>name" -- size+n ; Exec: addr -- addr+n
```

Create a new field within a structure definition of size n bytes.

## 2.13 Structure compilation

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

```
: >mark      \ -- addr
```

Mark the start of a forward branch. HIGH LEVEL CONSTRUCTS ONLY. INTERNAL.

```
: >resolve   \ addr --
```

Resolve absolute target of forward branch. HIGH LEVEL CONSTRUCTS ONLY. INTERNAL.

```
: <mark      \ -- addr
```

Mark the start (destination) of a backward branch. HIGH LEVEL CONSTRUCTS ONLY. INTERNAL.

```
: <resolve   \ addr --
```

Resolve a backward branch to addr. HIGH LEVEL CONSTRUCTS ONLY. INTERNAL.

```
synonym >c_res_branch >resolve \ addr -- ; fix up forward referenced branch
```

See >RESOLVE. INTERNAL.

```
synonym c_mrk_branch< >mark \ -- addr ; mark destination of backward branch
```

See >MARK INTERNAL.

## 2.14 Branch constructors

Used when compiling code on the target.

```
: c_branch<      \ addr --
```

Lay the code for BRANCH. INTERNAL.

```
: c_?branch<    \ addr --
```

Lay the code for ?BRANCH.

```
: c_branch>     \ -- addr
```

Lay the code for a forward referenced unconditional branch. INTERNAL.

```
: c_?branch>    \ -- addr
```

Lay the code for a forward referenced conditional branch. INTERNAL.

## 2.15 Main structure compilers

```
: c_lit         \ lit --
```

Compile the code for a LITERAL. INTERNAL.

```
: c_drop        \ --
```

Compile the code for DROP. INTERNAL.

```
: c_exit        \ --
```

Compile the code for EXIT. INTERNAL.

```
: c_do          \ C: -- do-sys ; Run: n1|u1 n2|u2 -- ; R: -- loop-sys
```

Compile the code for DO. INTERNAL.

```
: c_?DO        \ C: -- do-sys ; Run: n1|u1 n2|u2 -- ; R: -- | loop-sys
```

Compile the code for ?DO. INTERNAL.

```
: c_LOOP        \ C: do-sys -- ; Run: -- ; R: loop-sys1 -- | loop-sys2
```

Compile the code for LOOP. INTERNAL.

```
: c_+LOOP       \ C: do-sys -- ; Run: -- ; R: loop-sys1 -- | loop-sys2
```

Compile the code for +LOOP. INTERNAL.

```
variable NextCaseTarg \ -- addr
```

Holds the entry point of the current CASE structure. INTERNAL.

```
: c_case        \ -- addr
```

Compile the code for CASE. INTERNAL.

```
: c_OF          \ C: -- of-sys ; Run: x1 x2 -- | x1
```

Compile the code for OF. INTERNAL.

```
: c_?OF        \ C: -- of-sys ; Run: flag --
```

Compile the code for ?OF. INTERNAL.

```
: c_ENDOF       \ C: case-sys1 of-sys -- case-sys2 ; Run: --
```

Compile the code for ENDOF. INTERNAL.

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

Compile the code to resolve the forward branches at the end of a CASE structure. INTERNAL.

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

Compile the code for ENDCASE. INTERNAL.

```
: c_END-CASE    \ C: case-sys -- ; Run: x --
```

Compile the code for END-CASE. INTERNAL.

## 2.16 Miscellaneous

```
code clz      \ x -- #1z
```

Count the number of leading zeros in x.



## 3 High level kernel KERNEL62.FTH

### 3.1 User variables

variable next-user        \ -- addr

Next valid offset for a USER variable created by +USER.

: +user                \ size --

Creates a USER variable size bytes long at the offset given by NEXT-USER and updates NEXT-USER.

tcb-size +user SELF                \ task identifier and TCB

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

cell +user S0                        \ base of data stack

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

cell +user R0                        \ base of return stack

Holds the initial setting of the return stack pointer.

cell +user #TIB                      \ number of chars currently in TIB

Holds the number of characters currently in TIB.

cell +user 'TIB                      \ address of TIB

Holds the address of TIB, the terminal input buffer.

cell +user >IN                      \ offset into TIB

Holds the current character position being processed in the input stream.

cell +user XON/XOFF                 \ true if XON/XOFF protocol in use

True when console is using XON/XOFF protocol.

cell +user ECHOING                 \ true if echoing

True when console is echoing input characters.

cell +user OUT                      \ number of chars displayed on current line

Holds the number of chars displayed on current output line. Reset by CR.

cell +user BASE                     \ current numeric conversion base

Holds the current numeric conversion base

cell +user HLD                      \ used during number formatting

Holds data used during number formatting

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

Holds the number of cells converted by NUMBER?

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

Holds the number of digits converted by NUMBER?

cell +user DPL                      \ position of double number character id

Holds the number of characters after the double number indicator character. DPL is initialised to -1, which indicates a single number, and is incremented for each character after the separator.

cell +user HANDLER                 \ used in catch and throw

Holds the address of the previous exception frame.

cell +user OPVEC                    \ output vector

Holds the address of the I/O vector for the current output device.

cell +user IPVEC \ input vector

Holds the address of the I/O vector for the current input device.

cell +user 'AbortText \ Address of text from ABORT"

Set by the run-time action of ABORT" to hold the address of the counted string used by ABORT" <text>"

#64 chars dup +user PAD

Holds the

## 3.2 System Constants

Various constants for the internal system.

FALSE The well formed flag version for a logical negative

TRUE The well formed flag version for a logical positive

BL An internal constant for blank space

C/L Max chars/line for internal displays under C/LINE

#VOCS Maximum number of Vocabularies in search order

VSIZE Size of CONTEXT area for search order

XON XON character for serial line flow control

XOFF XOFF character for serial line flow control

## 3.3 System VARIABLES and Buffers

### 3.3.1 Variables

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

WIDTH maximum target name size

FENCE protected dictionary

DP dictionary pointer

VOC-LINK links vocabularies

RP Harvard targets only. The equivalent of DP for DATA space.

SCR If BLOCKS? true; for mass storage

BLK If BLOCKS? true; user input dev: 0 for keyboard, >0 for block

CURRENT Vocabulary/wordlist in which to put new definitions

STATE Interpreting=0 or compiling=-1

CSP Preserved stack pointer for compile time error checking

CONTEXT Search order array

LAST Points to name field of last definition

#THREADS Default number of threads in new wordlists

### 3.4 Deferred words

```
defer NUMBER? \ addr -- d/n/- 2/1/0
```

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

```
defer ERROR \ n -- ; error handler
```

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

### 3.5 Predefined Vocabularies

FORTH Is the standard general purpose vocabulary

ROOT This vocabulary stores the bare minimum functions

### 3.6 Vectored I/O handling

#### 3.6.1 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.

#### 3.6.2 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 !
```



### 3.6.3 Generic I/O words

: KEY? \ -- flag ; check receive char

Return true if a character is available at the current input device.

: KEY \ -- char ; receive char

Wait until the current input device receives a character and return it.

: EMIT \ -- char ; display char

Display char on the current I/O device. OUT is incremented before executing the vector function.

: TYPE \ caddr len -- ; display string

Display/write the string on the current output device. Len is added to OUT before executing the vector function.

: CR \ -- ; display new line

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

: TYPEC \ caddr len -- ; display string

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

: SPACE \ --

Output a blank space (ASCII 32) character.

: SPACES \ n --

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

: FlushKeys \ --

Compiled for 32 bit systems to flush any pending input that might be returned by KEY.

: SetConsole \ device --

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

## 3.7 String and memory operations

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

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

`char = byte = address-unit`

: PLACE \ c-addr1 u c-addr2 -- ; copies uncounted string to counted

Place the string c-addr1/u as a counted string at c-addr2.

: BOUNDS \ addr len -- addr+len addr

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

: upc \ char -- char' ; convert to upper case

If char is in the range 'a' to 'z' convert it to upper case. Note that this word is language specific and is written to handle English only.

: UPPER            \ c-addr u --

Convert the ASCII string described to upper-case. This operation happens in place. Note that this word is language specific and is written to handle English only.

: ERASE            \ a-addr u --

Erase U bytes of memory from A-ADDR with 0.

: BLANK            \ a-addr u --

Blank U bytes of memory from A-ADDR using ASCII 32 (space).

### 3.8 Dictionary management

: HERE            \ -- addr

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

: ALLOT            \ n --

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

: aligned         \ addr -- addr'

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

: ALIGN            \ --

ALIGN dictionary pointer using the same rules as ALIGNED.

: LATEST          \ -- c-addr

Return the address of the name field of the last definition.

: SMUDGE          \ --

Toggle the SMUDGE bit of the latest definition.

: ,                \ x --

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

: W,               \ w --

Place the WORD value X into the dictionary at HERE and increment the pointer. This word is not present on 16 bit implementations.

: C,               \ char --

Place the CHAR value into the dictionary at HERE and increment the pointer.

: there            \ -- addr

Harvard targets only: Return the DATA space pointer.

: allot-ram        \ n --

Harvard targets only: ALLOT DATA space.

: c,(r)            \ b --

Harvard targets only: The equivalent of C, for DATA space.

: ,(r)            \ n --

Harvard targets only: The equivalent of , for DATA space.

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

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

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

The inverse of N>LINK.

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

Move a pointer from an XT to the link field address.

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

The inverse of >LINK.

```
: >VOC-LINK      \ wid -- a-addr
```

Step from a wordlist identifier, wid, to the address of the field containing the address of the previously defined wordlist.

```
: >#THREADS      \ wid -- a-addr ; for XC5 compatibility
```

Step from a wordlist identifier, wid, to the address of the field containing the number of threads in the wordlist.

```
: >THREADS       \ wid -- a-addr
```

Step from a wordlist identifier, wid, to the address of the array containing the top NFA for each thread in the wordlist.

```
: >VOCNAME       \ wid -- a-addr
```

Step from a wordlist identifier, wid, to the address of the field pointing to the vocabulary name field.

```
: FIND           \ c-addr -- c-addr 0|xt 1|xt -1
```

Perform the "SEARCH-WORDLIST" operation on all wordlists within the current search order. This definition takes a counted string rather than a c-addr u pair. The counted string is returned as well as the 0 on failure.

```
: .NAME          \ nfa --
```

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

```
: makeheader     \ c-addr len --
```

Given a word name as string in addr/len form, build a dictionary header for the word.

```
: $CREATE        \ c-addr --
```

Perform the action of CREATE (below) but take the name from a counted string. OBSOLETE: replace by:

```
count makeheader dcreate,
```

```
: CREATE         \ --
```

Create a new definition in the dictionary. When the new definition is executed it will return the address of the definition BODY.

### 3.9 String compilation

```
: (C")           \ -- c-addr
```

The run-time action for C" which returns the address of and steps over a counted string.

```
: (S")           \ -- c-addr u
```

The run-time action for S" which returns the address and length of and steps over a string.

```
: (ABORT")       \ i*x x1 -- | i*x
```

The run time action of ABORT".

```
: (."")          \ --
```

The run-time action of .".

## 3.10 Pre-ANS Exception handlers

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

```
: ABORT          \ i*x -- ; R: j*x --
```

Performs "-1 THROW". This is a compatibility word for earlier versions of the kernel. Unfortunately, the earlier versions gave problems when ABORT was used in interrupt service routines or tasks. The new definition is brutal but consistent.

```
: ABORT"         \ Comp: "ccc<quote>" -- ; Run: i*x x1 -- | i*x ; R: j*x -- | j*x
```

If x1 is non-zero at run-time, store the address of the following counted string in USER variable 'ABORTTEXT, and perform "-2 THROW". The text interpreter in QUIT will (if reached) display the text.

```
: (Error)       \ n --
```

The default action of ERROR. This definition has been removed from v6.2 onwards. See the section about the changes from v6.1 to v6.2.

```
: ?ERROR        \ flag n --
```

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

## 3.11 ANS words CATCH and THROW

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

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

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

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

### 3.11.1 Example implementation

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

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

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

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

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

```

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

```

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

### 3.11.2 Example use

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

Typical use:

```

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

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

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

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

```

### 3.11.3 Gotchas

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

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

```
: 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.

## 3.12 Formatted and unformatted i/o

### 3.12.1 Setting number bases

```
: HEX        \ --
```

Change current radix to base 16.

: DECIMAL \ --

Change current radix to base 10.

: OCTAL \ --

Change current radix to base 8. 32 bit targets only.

: BINARY \ --

Change current radix to base 2.

### 3.12.2 Numeric output

: HOLD \ char --

Insert the ascii 'char' value into the pictured numeric output string currently being assembled.

: SIGN \ n --

Insert the ascii 'minus' symbol into the numeric output string if 'n' is negative.

: # \ ud1 -- ud2

Given a double number on the stack this will add the next digit to the pictured numeric output buffer and return the next double number to work with. PLEASE NOTE THAT THE NUMERIC OP STRING IS BUILT FROM RIGHT(lsd) to LEFT(msd).

: #S \ ud1 -- ud2

Keep performing # until all digits are generated.

: <# \ --

Begin definition of a new numeric output string buffer.

: #> \ xd -- c-addr u

Terminate definition of a numeric output string. Returns address and length of the ascii result.

: -TRAILING \ c-addr u1 -- c-addr u2

Modify a string address/length pair to ignore any trailing spaces.

: D.R \ d n --

Output the double number 'd' using current radix, right justified to 'n' characters. Padding is inserted using spaces on the left side.

: D. \ d --

Output the double number 'd' without padding.

: . \ n --

Output the cell signed value 'n' without justification.

: U. \ u --

As with . but treat as unsigned.

: U.R \ u n --

As with D.R but uses a single-unsigned cell value.

: .R \ n1 n2 --

As with D.R but uses a single-signed cell value.

### 3.12.3 Numeric input

: SKIP-SIGN \ addr1 len1 -- addr2 len2 t/f ; true if sign=negative

Inspect the first character of the string, if it is a '+' or '-' character, step over the string. Returning true if the character was a '-', otherwise return false.

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

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

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

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

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

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

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

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

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

If any BASE override prefixes or suffixes are used in the input string, set BASE accordingly and return the string without the override characters.

```
: Integer?    \ $addr -- value type | 0
```

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

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

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

### 3.13 String input and output

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

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

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

If pos is non-zero and ECHOING is set, perform BS and return

```
: ?EMIT      \ char -- ; emit if echoing enabled
```

If ECHOING is set, EMIT the character, otherwise discard it.

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

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

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

Output the text upto the closing double-quotes character.

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

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

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

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



### 3.14 Source input control

0 value SOURCE-ID \ -- n ; indicates input source

Returns an indicator of which device is generating source input. See the ANS specification for more details.

: TIB \ -- c-addr ; return address of terminal i/p buffer

Returns the address of the terminal input buffer. Note that tasks requiring user input must initialise the USER variable 'TIB. New code should use SOURCE and TO-SOURCE instead for ANS Forth compatibility.

: TO-SOURCE \ c-addr u --

Set the address and length of the system terminal input buffer. These are held in the user variables 'TIB and #TIB.

: SOURCE \ -- c-addr u ; returns address and length of input source buffer

Returns the address and length of the current terminal input buffer.

: SAVE-INPUT \ -- xn..x1 n

Save all the details of the input source onto the data stack. If it later becomes necessary to discard the saved input, NDROP will do the job. If you want to move the data to the return stack, N>R and NR> are available in some 32 bit implementations.

: RESTORE-INPUT \ xn..x1 n -- flag

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

: QUERY \ -- ; fetch line into TIB

Reset the input source specification to the console and accept a line of text into the input buffer.

: REFILL \ -- flag ; refill input source

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

### 3.15 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 in input stream, 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 occurred, 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 and in winprocs.

: WORD \ char "<chars>ccc<char>" -- c-addr

Similar behaviour to the ANS PARSE definition but the returned string is described as a counted string.

### 3.16 Miscellaneous

: HALT? \ -- flag

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

```
: origin-      \ addr -- addr' ; normalise NFA to base of primary CDATA section
```

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

```
: origin+      \ addr -- addr' ; denormalise NFA again
```

If addr is non-zero, add the start address of the first defined CDATA section. This word is only compiled if the start address of the first defined CDATA section is non-zero.

```
: nfa-buff     \ -- addr+len addr ; make a buffer for holding NFAs
```

Form a temporary buffer for holding NFAs. A factor for WORDS.

```
: MAX-NFA      \ -- addr c-addr ; returns addr and top nfa
```

Return the thread address and NFA of the highest word in the NFA buffer. A factor for WORDS.

```
: COPY-THREADS \ addr --
```

Copy the threads of the CONTEXT wordlist to a temporary NFA buffer for manipulation. A factor for WORDS.

```
: WORDS        \ --
```

Display the names of all definitions in the wordlist at the top of the search-order.

```
: MOVE         \ addr1 addr2 u -- ; intelligent move
```

An intelligent memory move, chooses between CMOVE and CMOVE> at runtime to avoid memory overlap problems. Note that as ROM PowerForth characters are 8 bit, there is an implicit connection between a byte and a character.

```
: DEPTH        \ -- +n
```

Return the number of items on the data stack, excluding the count.

```
: UNUSED       \ -- u ; free dictionary space
```

Return the number of bytes free in the dictionary.

```
: .FREE        \ --
```

Return the free dictionary space.

### 3.17 Wordlist control

```
: WORDLIST     \ -- wid
```

Create a new wordlist and return a unique identifier for it.

```
: VOCABULARY   \ -- ; VOCABULARY <name>
```

Create a VOCABULARY which is implemented as a named wordlist.

```
: FORTH        \ --
```

Install FORTH wordlist into search-order.

```
: FORTH-WORDLIST \ -- wid
```

Return the unique WID for the main FORTH wordlist.

```
: GET-CURRENT  \ -- wid
```

Return the WID for the Wordlist which holds any definitions made at this point.

```
: SET-CURRENT  \ wid --
```

Change the wordlist which will hold future definitions.

```
: GET-ORDER    \ -- widn...wid1 n
```

Return the list of WIDs which make up the current search-order. The last value returned on top-of-stack is the number of WIDs returned.

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

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

```
: ONLY           \ --
```

Set the minimum search order as the current search-order.

```
: ALSO          \ --
```

Duplicate the first WID in the search order.

```
: PREVIOUS      \ --
```

Drop the current top of search-order.

```
: DEFINITIONS   \ --
```

Set the current top WID of search-order as the current definitions wordlist.

### 3.18 Control structures

```
: ?PAIRS        \ x1 x2 --
```

If  $x1 <> x2$ , issue and error. Used for on-target compile-time error checking.

```
: !CSP          \ x --
```

Save the stack pointer in CSP. Used for on-target compile-time error checking.

```
: ?CSP          \ --
```

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

```
: ?COMP         \ --
```

Error if not in compile state.

```
: ?EXEC         \ --
```

Error if not interpreting.

```
: DO            \ C: -- do-sys ; Run: n1|u1 n2|u2 -- ; R: -- loop-sys
```

Begin a DO ... LOOP construct. Takes the end-value and start-value from the data-stack.

```
: ?DO           \ C: -- do-sys ; Run: n1|u1 n2|u2 -- ; R: -- | loop-sys
```

Compile a DO which will only begin loop execution if the loop parameters are not the same. Thus `0 0 ?DO ... LOOP` will not execute the contents of the loop.

```
: LOOP         \ C: do-sys -- ; Run: -- ; R: loop-sys1 -- | loop-sys2
```

The closing statement of a DO..LOOP construct. Increments the index and terminates when the index crosses the limit.

```
: +LOOP        \ C: do-sys -- ; Run: n -- ; R: loop-sys1 -- | loop-sys2
```

As with LOOP except you specify the increment on the data-stack.

```
: BEGIN        \ C: -- dest ; Run: --
```

Mark the start of a BEGIN..[while]..UNTIL / AGAIN / [REPEAT] construct.

```
: 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 ..ELSE..THEN conditional.

```
: endif      \ C: orig -- ; Ru: -- ; synonym for THEN
```

An alias for THEN. Note that ANS Forth describes THEN not ENDIF.

```
: AHEAD      \ C: -- orig ; Run: --
```

Start an unconditional forward branch which will be resolved later.

```
: ELSE       \ C: orig1 -- orig2 ; Run: --
```

Begin the failure condition code for an IF.

```
: CASE       \ C: -- case-sys ; Run: --
```

Begin a CASE..ENDCASE construct. Similar to 'C' language 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.

```
: END OF     \ 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 off the stack.

```
: ?OF        \ C: -- of-sys ; Run: flag --
```

Begin conditional block for CASE, executed when the flag is true.

```
: END-CASE   \ C: case-sys -- ; Run: --
```

A Version of ENDCASE which does not drop the switch value. Used when the switch value itself is consumed by a DEFAULT condition.

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

Terminate a CASE..NEXTCASE construct. DROPS the switch value from the stack and compiles a branch back to the top of the loop at CASE.

```
: RECURSE   \ Comp: --
```

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

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

to compile a reference to FOO from inside FOO.

### 3.19 Target interpreter and compiler

```
: ?STACK     \ --
```

Error if stack pointer out of range.

```
: ?UNDEF     \ x --
```

Word not defined error if x=0.

```
: (compile)  \ -- ; compiles in line xt
```

The run-time action for COMPILE and friends.

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

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

: S" \ Comp: "ccc<quote>" -- ; Run: -- c-addr u

Describe a string. Text is taken upto the next double-quote character. The address and length of the string are returned.

: C" \ Comp: "ccc<quote>" -- ; Run: -- c-addr

As with S" except the address of a counted string is returned.

: #LITERAL \ n1 .... nn n -- ; put in dictionary n1 first

Compile n1..nn as literals so that the same stack order results when the code executes.

: LITERAL \ Comp: x -- ; Run: -- x

Compile a literal into the current definition. Usually used in the form [ <expression > ] LITERAL inside a colon definition. Note that LITERAL is IMMEDIATE.

: 2LITERAL \ Comp: x1 x2 -- ; Run: -- x1 x2

A two cell version of LITERAL.

: CHAR \ "<spaces>name" -- char

Return the first character of the next token in the input stream. Usually used to avoid magic numbers in the source code.

: [CHAR] \ Comp: "<spaces>name" -- ; Run: -- char

Compile the first character of the next token in the input stream as a literal. Usually used to avoid magic numbers in the source code.

: sliteral \ c-addr u -- ; Run: -- c-addr2 u ; 17.6.1.2212

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

: [ \ --

Switch compiler into interpreter state.

: ] \ --

Switch compiler into compilation state.

: IMMEDIATE \ --

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

: ' \ "<spaces>name" -- xt

Find the xt of the next word in the input stream. An error occurs if the xt cannot be found.

: ['] \ Comp: "<spaces>name" -- ; Run: -- xt

Find the xt of the next word in the input stream, and compile it as a literal. An error occurs if the xt cannot be found.

: [COMPILE] \ "<spaces>name" --

Compile the next word in the input stream. [COMPILE] ignores the IMMEDIATE state of the word. Its operation is mostly superseded by POSTPONE.

: ( \ "ccc<paren>" --

Begin an inline comment. All text upto the closing bracket is ignored.

: \ \ "ccc<eol>" --

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

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

Parse text up to the closing quote and compile into the dictionary at HERE as a counted string. The end of the string is aligned.

```
: .(          \ "cc<paren>" --
```

A documenting comment. Behaves in the same manner as ( except that the enclosed text is written to the console at compile time.

```
: ASSIGN      \ "<spaces>name" --
```

A state smart word to get the XT of a word. The source word is parsed from the input stream. Used as part of a ASSIGN xxx TO-DO yyy construct.

```
: (TO-DO)     \ -- ; R: xt -- a-addr'
```

The run-time action of TO-DO. It is followed by the data address of the DEFERred word at which the xt is stored.

```
: TO-DO       \ "<spaces>name" --
```

The second part of the ASSIGN xxx TO-DO yyy construct. This word will assign the given XT to be the action of a DEFERred word which is named in the input stream.

```
: exit        \ R: nest-sys -- ; exit current definition
```

Compile code into the current definition to cause a definition to terminate. This is the Forth equivalent to inserting an RTS/RET instruction in the middle of an assembler subroutine.

```
: ;           \ C: colon-sys -- ; Run: -- ; R: nest-sys --
```

Complete the definition of a new 'colon' word or :NONAME code block.

```
: INTERPRET   \ --
```

Process the current input line as if it is text entered at the keyboard.

```
: N>R         \ xn .. x1 N -- ; R: -- x1 .. xn n
```

Transfer N items and count to the return stack.

```
: NR>         \ -- xn .. x1 N ; R: x1 .. xn N --
```

Pull N items and count off the return stack.

```
: EVALUATE    \ i*x c-addr u -- j*x ; interpret the string
```

Process the supplied string as though it had been entered via the interpreter.

```
: .throw      \ throw# --
```

Display the throw code. Values of 0 and -1 are ignored.

```
: QUIT        \ -- ; R: i*x --
```

Empty the return stack, store 0 in SOURCE-ID, and enter interpretation state. QUIT repeatedly ACCEPTs a line of input and INTERPRETs it, with a prompt if interpreting and ECHOING on. Note that any task that uses QUIT must initialise 'TIB, BASE, IPVEC, and OPVEC.

## 3.20 Compilation and Caches

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

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

interactively on the target and then executing them before the code has got into the Icache. Only executable code has to be synchronised, not data.

If the word FLUSHCACHE ( - ) is provided before KERNEL62.FTH is compiled, it will be executed by the text interpreter before each line is processed. FLUSHCACHE is also executed by ';

## 3.21 Startup code

### 3.21.1 Cold chain

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

```
' <wordname> AtCold
```

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

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

```
1: ColdChainFirst      \ -- addr
```

Dummy first entry in ColdChain.

```
variable ColdChain    \ -- addr
```

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

```
: AtCold              \ xt --
```

Specify 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.

### 3.21.2 The COLD sequence

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

```
: (INIT)              \ --
```

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

```
: COLD                \ --
```

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

## 3.22 Kernel error codes

-4	Data 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.

## 3.23 Differences between the v6.1 and 6.2 kernels

### 3.23.1 Error handling

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

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

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

The new version is:

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

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

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

is now part of QUIT. The phrase "S0 @ SP!" must now be provided by the THROW handler.



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

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

### 3.23.2 Terminal input buffer and `ACCEPT`.

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

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

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

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

Removed: `?EMIT SOURCE-STRING`

## 4 Target VALUE and local variables

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

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

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

```
: OPERATOR      \ n -- ; define an operator in the cross compiler
```

An interpreter definition that build new operators such as "to" and "addr".

```
: VALUE          \ n -- ; -- n ; n VALUE <name>
```

Creates a variable of initial value n that returns its contents when referenced. To store to a child of VALUE use "n to <child>".

```
: (LOCAL)        \ Comp: c-addr u -- ; Exec: -- x ; define local var
```

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

```
: LOCALS|        \ "name...name |" --
```

Create named local variables <name1> to <namen>. At run time the stack effect is ( xn..x1 – ), such that <name1> is initialised with x1 and <namen> is initialised with xn. Note that this means that the order of declaration is the reverse of the order used in stack comments! When referenced, a local variable returns its value. To write to a local, precede its name with TO.

In the example below, a and b are named inputs.

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



## 5 Development tools

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

1 equ simple? \ -- n

Set this flag non-zero to generate .xWORD to avoid divisions. On some CPUs, a division operation is slow.

: .nibble \ n --

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

: .BYTE \ b --

Display b as two hex digits.

: .WORD \ w --

Display w as four hex digits.

: .LWORD \ x --

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

: .DWORD \ x --

A synonym for .LWORD.

: .ASCII \ char --

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

: DUMP \ addr len --

Display (dump) len bytes of memory starting at addr.

: LDUMP \ addr len -- ; dump 32 bit long words

Display (dump) len bytes of memory starting at addr as 32 bit words.

: WDUMP \ addr len -- ; dump 16 bit half words

Display (dump) len bytes of memory starting at addr as 16 bit half-words.

: .S \ --

Display the contents of the data stack without affecting it.

: ? \ a-addr --

Display contents of a memory location as a cell.



## 6 Debugging tools

Copyright (c) 1996-2004  
 MicroProcessor Engineering  
 133 Hill Lane  
 Southampton S015 5AF  
 England

tel: +44 (0)23 8063 1441  
 fax: +44 (0)23 8033 9691  
 net: mpe@mpeltd.demon.co.uk  
 tech-support@mpeltd.demon.co.uk  
 web: www.mpeltd.demon.co.uk

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

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

### 6.1 Implementation dependencies

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

```
: consoleIO      \ --
```

Select debug console for output. By default this is the `CONSOLE` device.

```
  console opvec ! console ipvec !
  Echoing on Xon/Xoff off
```

```
;
```

```
: name?          \ addr -- flag                                MPE.0000
```

Check to see if the supplied address is a valid NFA, returning true if the address appears to be a valid NFA. This word is implementation dependent. For MPE cross compilers, a valid NFA for MPE embedded systems satisfies the following:

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

```
count                                     \ c-addr u --
dup $9F and $81 $9F within? 0=           \ NFA first byte = 1SIxxxxx, count = xxxxxx
                                          \ mask                = 10011111

if 2drop 0 exit then
$01F and bounds ?do
  i c@ #33 #126 within? 0=               \ check all ascii chars
  if unloop FALSE exit then
loop
```



```

    drop
;
: .decimal      \ n --
Display a value as a decimal number.
    base @ >r decimal . r> base !
;
: .hex          \ n --
Display a value as a hexadecimal number.
    base @ >r hex u. r> base !
;
: [con         \ -- ; R: -- consys
Saves BASE and the current i/o vectors on the return stack, and then switches to the console
and decimal.
    r>
    base @ >r opvec @ >r ipvec @ >r
    ConsoleIO decimal
    >r
;
: con]         \ -- ; R: consys --
Restores BASE and the current i/o vectors from the return stack.
    r>
    r> ipvec ! r> opvec ! r> base !
    >r
;
: CheckFailed  \ ip caddr len --
Given the address at the fault occurred and a string, output the string and some diagnostic
information.
    [con
        cr type ." failed at "
        dup .dword ." in " ip>nfa .name
    con]
;

```

### 6.3 Stack checking

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

```

: ?StackDepth  \ +n --
If the stack depth before +n is not n, issue a console warning message and clear the stack. Note
that this word is implementation dependent.
    dup 2+ depth =
    if drop exit endif          \ no failure
    [con
        cr ." *** Stack fault: depth = " depth 1- 0 .r ." (d) "
        ." in task " self .task          \ indicate current task
    con]
;

```



```

>r s0 @ sp! r> 0 ?do 0 loop      \ set required depth
cr ."  Stack updated."
con]
;
: ?StackEmpty    \ --
If the stack depth is non-zero, issue a console warning message and clear the stack.
0 ?StackDepth
;
: TaskChecks     \ --
Use in task to check for creeping stacks and so on. This word can be extended to provide
additional internal consistency checks.
?StackEmpty
;
: SF{           \ n -- ; R: -- depth
n SF{ ... }SF will check for stack faults. n describes the stack change between SF{ and }SF.
If the stack change is different, an error message is generated. This word will work on most
systems in which the return address is held on the return stack.
r> swap depth 2- + >r >r
;
: }SF          \ -- ; R: depth -- ; perform stack check
The end of an SF{ ... }SF structure. This word is not strictly portable as it assumes that the
Forth return stack holds a valid return address. In the vast majority of cases the assumption is
true, but beware of some 8051 implementations. See SF{
r>
r> depth 2- <> if
  dup s" Stack check" CheckFailed
endif
>r
;

```

## 6.4 Assertions

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

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

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

```
-1 value assert?      \ -- n
```

Returns non-zero if asserts will be tested.

```
: (assert)          \ flag --
```

If flag is zero, report an ASSERT error.

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

```

```
: [assert          \ --
```

Compile the code to start an assert.

```
?comp                                \ must be compiling
postpone assert? postpone if
; immediate
: assert]                             \ --
Compile the code to end an assert.
?comp                                \ must be compiling
postpone (assert) postpone then
; immediate
```

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

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



## 7 Interrupt handlers

The file `ARM\INTARM3.FTH` contains generic ARM interrupt handlers in the v6.2 style, plus alternative handlers for CPUs with vectored interrupt controllers. `INTARM3.FTH` requires `ARM\ARMDEF.FTH` to be compiled before the `SFRxxxx` file for your particular CPU. The file `ARM/STACKDEF.FTH` provides default main task and stack layouts and should be compiled from the control file or copied into your control file. Default stack initialisation code is provided by the assembler routine `INITSTACKS` in `ARM\INITSTACKS.FTH`. This routine can be referenced from your initialisation code.

You are strongly recommended to use `INTARM3` for new code. The `INTARM3.FTH` interrupt handlers are much more paranoid (and hence safer) than previous releases. The default diagnostic code gives much more information when an abort occurs. The files `INTARM.FTH`, `INTARM2.FTH`, `INTS3C4510.FTH` and `INTAT91.FTH` remain in the distribution but will not be supported in future releases.

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

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

The `IRQ` handlers all share a common "stack of stacks". On entry to the interrupt handler, Forth system registers are allocated. Your initialisation code **MUST** set up `R13` for each CPU mode. See `ARM\CONFIGS\LPC2106U.CTL` (contains stack layout) and `ARM\HARDWARE\INITLPC210X.FTH` (contains initialisation code) for examples.

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

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

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

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

```
UP > PSP > RSP
```

Because of this layout, data stack underflows may corrupt the first part of the `USER` area. You

have been warned. During testing, it may be as well to set the equate SP-GUARD to 2 or 3 in your control file to leave a few guard cells on the data stack.

## 7.1 Configuration

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

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

specifies that no vectored interrupt controller is present.

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

specifies that the AT91 AIC vectored interrupt controller is present.

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

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

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

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

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

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

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

If the equate ARM-VIC? is undefined before this file is compiled, the generic (no vectored controller) will be compiled for FIQ and IRQ handling.

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

Non-zero to compile test and diagnostic code.

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

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

```
0 equ Reserved_ISR? \ -- n
```

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

```
: !call \ xt addr --
```

An INTERPRETER word to create a call opcode to xt at addr.

## 7.2 Interrupt management

```
code EI \ -- ; enable interrupts
```

Global enable interrupts, both FIQ and IRQ.

```
code DI \ -- ; disable interrupts
```

Global disable interrupts, both FIQ and IRQ.

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

```
code I]          \ R: x -- ; restore int. status from r. stack
```

Restore interrupt status saved by [I from the return stack.

```
code SAVE-INT   \ -- x ; return interrupt status and disable interrupts
```

Get return interrupt status and then disable interrupts. Use [I and I] for new code. Now in LIBARM.FTH.

```
code RESTORE-INT \ x -- ; restore state returned by SAVE-INT
```

Restore state returned by SAVE-INT. Use [I and I] for new code. Now in LIBARM.FTH.

### 7.3 SWI handler

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

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

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

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

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

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

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

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

An example SWI handler.

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

Test execution of SWI 0.

### 7.4 Support for complex abort handlers

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

```

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

```

```
struct /exframe \ -- n
```

structure defining the exception frame

```
: .item      \ addr -- addr+4
```

Display a cell item at addr and increment addr.

```
: .items     \ addr n -- addr+4n
```

Display n items at addr and increment addr.

```
: .frame     \ ^frame --
```

Display the CPU state pointed to by the data frame.

```
: name?     \ addr -- flag
```

Check to see if the supplied address is a valid NFA. This word is implementation dependent. A valid NFA for MPE embedded systems satisfies the following:

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

```
: ip>nfa    \ addr -- nfa
```

Attempt to move backwards from an address within a definition to the relevant NFA.

```
: check-aligned \ addr -- addr'
```

Check addr for cell alignment, report and correct if misaligned.

```
: ?Clip32   \ xsp xspTop -- xsp xspTop'
```

Clip the stack display to 32 items.

```
: .rsframe  \ ^frame --
```

Display the return stack indicated by the frame, assuming the Forth RSP=R13 and RUP=R11.

```
: .psframe  \ ^frame --
```

Display the data stack indicated by the frame, assuming the Forth PSP=R12 and RUP=R11.

## 7.5 Undefined instruction handler

### 7.5.1 Simple UNDEF handler

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

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

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

Test handler for Undefined Instructions.

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

Causes an Undefined Instruction exception.

### 7.5.2 Complex UNDEF handler

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

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

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

Test handler for Undefined Instructions.

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

Causes an Undefined Instruction exception.

## 7.6 Prefetch Abort handler

### 7.6.1 Simple PABORT handler

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

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

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

Test PAbort handler.

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

Run the test PAbort handler.

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

Run the test PAbort handler.

### 7.6.2 Complex PAbort handler

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

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

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

Test PABORT handler.

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

Run test PAbort handler.



## 7.7 Data Abort handler

### 7.7.1 Simple DAbort handler

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

`: testdabort \ ^ins -- fixed?`

Test DABORT handler.

`: run-dabort $04000000 @ ; \ hardware specific`

Run test PAbort handler.

### 7.7.2 Complex DAbort handler

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

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

Test DABORT handler.

`: run-dabort $05000000 @ drop ; \ hardware specific`

Run test DAbort handler.

## 7.8 Reserved (26 bit address exception) handler

This exception is only used for systems which support the 26 bit PC mode of early ARM cores.

`defer Unused_handler \ ^ins -- fixed?`

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

`PROC Unused_exception \ --`

The UNUSED exception handler calls UNUSED\_handler above.

`: testunused \ --`

Test code for the UNUSED exception.

## 7.9 Generic IRQ handler

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

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

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

```
variable next-irq      \ -- addr
```

Holds pointer to next ISR to run. ISRs are added to this chain.

```
PROC IRQ_exception    \ -- ; IRQ entry code
```

The IRQ handler entry point which executes the chain. The USER variables `S0` and `R0` are initialised.

```
proc IRQ_reti         \ -- ; IRQ exit code
```

This code cleans up after the IRQ and is the first item added to the IRQ chain.

```
: br24,              \ xt opcode --
```

Given an opcode (B or BL) and an execution address, compiles a branch or call to it.

```
: add-isr           \ xt chain -- ; ' <name> <chain_var> ADD-ISR adds name to the ISR handler list
```

An action is added to the IRQ or FIQ chain by a phrase of the form:

```
  ' <name> <chain_var> ADD-ISR
```

```
: add-irq          \ xt -- ; ' <name> ADD-IRQ adds name to the IRQ handler list
```

An action is added to the IRQ chain by a phrase of the form:

```
  ' <name> ADD-IRQ
```

## 7.10 Generic FIQ handler

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

```
variable next-fiq     \ -- addr
```

Holds pointer to next ISR to run. FIQ ISRs are added to this chain

```
PROC FIQ_exception    \ --
```

FIQ entry code.

```
proc FIQ_reti         \ --
```

FIQ exit code.

```
: add-fiq          \ xt -- ; ' <name> ADD-FIQ adds name to the FIQ handler list
```

An action is added to the FIQ chain by a phrase of the form:

```
  ' <name> ADD-FIQ
```

## 7.11 AT91 IRQ and FIQ handlers

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

This code requires `KERNEL62.FTH` and the equate `COLDCHAIN?` must be set non-zero in the control file.

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

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

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

```
PROC IRQ_entry \ --
```

This is the template code for vectored IRQ interrupt handlers. It is also used as the spurious interrupt handler.

```
PROC FIQ_entry \ --
```

This is the template code for vectored FIQ interrupt handlers.

```
: IRQ:          \ xt "<name>" -- ; -- isr
```

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

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

```
: FIQ: \ xt "<name>" -- ; -- isr
```

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

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

```
: SetDefIRQ     \ xt --
```

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

```
' <word> SetDefIRQ
```

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

```
: InitSPU      \ --
```

Initialise the spurious interrupt vector. This word is executed as part of the cold chain.

```
: EnInt        \ int# --
```

Enable the requested AIC interrupt.

```
: DisInt       \ int# --
```

Disable the requested AIC interrupt.

```
: SetIRQisr    \ isr mode int# --
```

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

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

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

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

```
$000 equ ISR_low      \ -- n
```

Low level sensitive.

```
$020 equ ISR_nedge    \ -- n
```

Negative edge triggered.

```
$040 equ ISR_high     \ -- n
```

High level sensitive.

```
$080 equ ISR_pedge    \ -- n
```

Positive edge triggered.

An example of setting up an interrupt follows.

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

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

```

## 7.12 Samsung S3C4510 IRQ and FIQ handlers

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

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

```
create despatch_table \ -- addr ; 21 entry interrupt despatch table
```

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

```
: SetFIQ \ xt int# --
```

Set the XT to be the action of FIQ INT#.

```
: SetIRQ \ xt int# --
```

Set the XT to be the action of IRQ INT#.

```
: EnInt \ int# --
```

Enable the requested interrupt.

```
: DisInt \ int# --
```

Disable the requested interrupt.

## 7.13 ARM PL190 IRQ and FIQ handlers

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

This code requires `KERNEL62.FTH` and the equate `COLDCHAIN?` must be set non-zero in the control file. The VIC addresses should be defined in the `SFRxxxx` file which defines the peripheral address for the silicon.

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

FIQ nesting is not supported by this code. If required a suitable starting point is the AT91 code. The FIQ stack must be at least `TASK-SIZE` bytes.

```
PROC IRQ_entry \ --
```

This is the template code for vectored IRQ interrupt handlers. It is also used as the default interrupt handler.

```
PROC FIQ_entry \ --
```

This is the template code for FIQ interrupt handlers.

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

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

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

```
: SetDefIRQ     \ xt --
```

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

```
  ' <word> SetDefIRQ
```

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

```
: SetFIQ        \ xt --
```

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

```
  ' <word> SetFIQ
```

```
: InitVIC       \ --
```

Initialise the VIC Default interrupt vector. This word is executed as part of the cold chain.

```
: EnInt         \ src# --
```

Enable the requested VIC interrupt source.

```
: DisInt        \ src# --
```

Disable the requested VIC interrupt source.

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

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

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

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

An example of setting up an interrupt follows.

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

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



## 8 Character Queues

The file COMMON\CQUEUES.FTH provides circular character (byte) queues. If the equate TASKING? is non-zero, the blocking routines will use PAUSE. Interrupts are disabled for the queue empty/full checks.

```
struct /cqueue \ -- size ; character queue structure in idata, buffer in udata
Circular queue data structure.
```

```
    int >qhead          \ Offset of head
    int >qtail          \ Offset of tail
    int >qchars         \ Number of characters in the queue
    int >qmask          \ Mask to apply to pointers
    ptr >qbuffer        \ Base address of character buffer
end-struct
```

```
: cqueue:          \ size -- ; -- cqueue ; size CQUEUE: <name>
```

An interpreter definition to build a character queue of the specified size. The queue data structure is built in the current IDATA space, and the buffer itself is in the current UDATA space. Executing <name> returns the address of the queue data structure. N.B. The size of a queue must be a power of two, e.g. 32, 64 ...

```
: init-cqueue     \ cqueue -- ; initialise queue
```

Initialise the specified queue created by CQUEUE:.

```
: init-hcqueue    \ size cqueue -- ; SFP002
```

Initialise the specified queue created by ALLOCATE. When a queue is allocated from the heap by a phrase of the form "/CQUEUE <size> + ALLOCATE", this word must be used.

```
: (>cqueue)       \ char cqueue -- ; put character on cqueue
```

Put char into the queue with no checks.

```
: (cqueue>)       \ cqueue -- char ; get next character from queue
```

Get the next character from the queue with no checks.

```
: (cqfull?)       \ queue -- flag ; TRUE if queue full
```

Return true if the queue is full. No interrupt protection is provided.

```
: cqfull?         \ queue -- flag ; TRUE if queue full
```

Return true if the queue is full. Interrupt protection is provided.

```
: cqchars         \ queue -- n
```

Return the number of characters in the queue.

```
: cqempty?        \ queue -- flag ; TRUE if queue empty
```

Return true if the queue is empty.

```
: cqnotempty?     \ queue -- flag ; TRUE if queue not empty
```

Return true if the queue is not empty, i.e. if it contains any characters.

```
: >cqueue         \ char cqueue -- ; spins if full
```

Put a character into the queue. If the queue is full, the system waits (blocks) until there is enough space.

```
: cqueue>         \ queue -- char ; spins while queue empty
```

Remove the next character, waiting if the queue is empty.





## 9 Serial driver

By default the serial drivers are initialised to 115200 baud for UART0 which is the serial console, and UART1 is initialised to 38400 baud. These baud rates can be changed by setting the relevant UART DLL and DLM registers. See the chip user manual for more details.

### 9.1 Configuration

The following equates control which UARTs are required and initialised. If the equates are undefined, they are generated and set to 1.

```
1 equ useUART0? \ -- n
Compile UART0 code if non-zero.

1 equ useUART1? \ -- n
Compile UART1 code if non-zero.
```

### 9.2 Serial interrupt service routines

```
: >RxQ          \ char queue --
Put character from UART into queue. INTERNAL.

: FIFO>RxQ      \ base queue --
Given a UART base address and a character queue, empty the UART FIFO into the queue.
INTERNAL.

: ser0-isrh     \ --
UART0 high level ISR. INTERNAL.

' ser0-isrh IRQ: ser0-isr
UART0 interrupt service routine entry point.

: ser1-isrh     \ --
UART1 high level ISR. INTERNAL.

' ser1-isrh IRQ: ser1-isr
UART1 interrupt service routine entry point.
```

### 9.3 Initialisation

```
: gen-baud-rate \ bps-rate -- divisor16
Takes the required serial comms rate (in BPS) and returns a system-dependent number of
parameters that should be sufficient to effect the baud rate selection by initialising the hardware.

: init-ser      \ --
Initialise both UARTs.
```

### 9.4 Serial primitives

```
: (seremit)     \ char base --
Send a character on the UART whose base address is given.

: (serCR)       \ base --
Send a CR/LF pair on the UART whose base address is given.

: (serTYPE)     \ c-addr len base --
Type a string to the UART whose base address is given.
```

```

: serEmit0      \ char --
Transmit a character on UART0.

: serkey?0      \ -- t/f
Return true if UART0 has a character available.

: sertype0      \ addr len --
Type a string to the UART.

: serCR0        \ --
Issue a CR/LF pair to the UART.

: serkey?1      \ -- t/f
Return true if UART1 has a character available.

: serEmit1      \ char --
Transmit a character on UART1.

: serTYPE1      \ c-addr len --
Type a string to the UART.

: serCR1        \ --
Issue a CR/LF pair to the UART.

```

## 9.5 Generic i/o assignments

```
create Console0 \ -- addr ; OUT managed by upper driver
Generic I/O device for UART0.
```

```
create Console1 \ -- addr ; OUT managed by upper driver
Generic I/O device for UART1.
```

```
Console0 constant Console
```

CONSOLE is the device used by the Forth system for interaction. It may be changed by a phrase of the form:

```
<device> dup opvec ! ipvec !
```

## 9.6 Forth Stamp specific code

This code is only compiled if the equate `ArmStamp?` is set true.

```
_UART0 constant UART0 \ -- addr
Base address of the UART0 peripheral.
```

```
_UART1 constant UART1 \ -- addr
Base address of the UART1 peripheral.
```

```
: SetBaud      \ bps-rate uart --
```

Takes the required serial comms rate (in BPS) and the UART base and sets the baud rate.

## 10 GPIO initialisation and USB driver

### 10.1 GPIO initialisation

The following table describes how the GPIO pins are used by the USB stamp board.

PinSel	00	01	10	11	LPC pin	Board
0.1:0	P0.0	TxD0	PWM1	RFU	13	RS232 o/p
0.3:2	P0.1	RxD0	PWM3		14	RS232 i/p
0.5:4	P0.2	SCL	Cap0.0		18	U2p6
0.7:6	P0.3	SDA	Mat0.0		21	U2p5
0.9:8	P0.4	WP	Cap0.1		22	U2p7
0.11:10	P0.5	MISO	Mat0.1		23	usbRD#
0.13:12	P0.6	MOSI	Cap0.2		24	usbWR
0.15:14	P0.7	SSEL	PWM2		28	usbTXE#
0.17:16	P0.8	TxD1	PWM4		29	usbD0
0.19:18	P0.9	RxD1	PWM6		30	usbD1
0.21:20	P0.10	RTS1	Cap1.0		35	usbD2
0.23:22	P0.11	CTS1	Cap1.1		36	usbD3
0.25:24	P0.12	DSR1	Mat1.0		37	usbD4
0.27:26	P0.13	DTR1	Mat1.1		41	usbD5
0.29:28	P0.14	CD1	EINT1		44	usbD6
0.31:30	P0.15	RI1	EINT2		45	usbD7
1.1:0	P0.16	EINT0	Mat0.2		46	usbRXF#
1.3:2	P0.17	Cap1.2	--		47	pTRST
1.5:4	P0.18	Cap1.3	--		48	pTMS
1.7:6	P0.19	Mat1.2	--		1	pTCK
1.9:8	P0.20	Mat1.3	--		2	pTDI
1.11:10	P0.21	PWM5		3		pTD0
1.13:12	P0.22	--			32	CPLD cTMS o/p
1.15:14	P0.23	--			33	CPLD cTCK o/p
1.17:16	P0.24	--			34	CPLD cRTCK i/p
1.19:18	P0.25	--			38	CPLD cTDI o/p
1.21:20	P0.26	--			39	CPLD cNTRST o/p
1.23:22	P0.27	TRST		8		CPLD cTD0 i/p
1.25:24	P0.28	TMS		9		CPLD cNSRST o/p
1.27:26	P0.29	TCK		10		CPLD cX
1.29:28	P0.30	TDI		15		CPLD cM0
1.31:30	P0.31	TDO		16		CPLD cM1

On versions of the MPE USB Stamp board for which the UART1 pins conflict with the USB chip pins, UART1 is not enabled. This is controlled by the equate USEUART1? in the control file.

```
i2cSCL i2cWP or equ i2cHiMask
```

Mask for I2C bits set high at initialisation

```
0 equ i2cLoMask
```

Mask for I2C bits set low at initialisation

```

usbRD# equ UsbHiMask    \ -- mask
Mask for USB bits set high at initialisation

usbWR equ UsbLoMask    \ -- mask
Mask for USB bits set low at initialisation

    equ JtagHiMask      \ -- mask
Mask for JTAG bits set high at initialisation

0 equ JtagLoMask       \ -- mask
Mask for JTAG bits set low at initialisation

    equ GpioDirMask
Mask with bits set for output. 1=o/p, 0=i/p.

    equ GpioHiMask      \ -- mask
Mask for GPIO bits set high at initialisation

    equ GpioLoMask      \ -- mask
Mask for GPIO bits set high at initialisation

create ^gpio          \ -- addr
Contains the base address of the GPIO registers.

: init-gpio           \ --
Initialise the GPIO bits except for the serial lines which are handled by the startup code.

```

## 10.2 USB comms driver

The USB comms driver follows the generic I/O model used by the serial ports.

```

: usbEmit             \ char --
Write a byte to the USB connection.

: usbKey?             \ -- flag
Return non-zero if a character is available to be read from the USB connection.

: usbKey              \ -- char
Read (wait) for a character from the USB connection.

: usbCR               \ --
Send a CR/LF pair on the USB connection.

: usbTYPE             \ c-addr len --
Type a string to the USB connection.

create UsbConsole     \ -- addr ; OUT managed by upper driver
Generic I/O device for USB.

    UsbConsole constant Console
CONSOLE is the device used by the Forth system for interaction. It may be changed by a
phrase of the form:

<device> dup opvec ! ipvec !

```

## 11 LPC software I2C driver

### 11.1 Introduction

The I2C interface is a software bit-banging system using GPIO bits 2, 3 and 4. The initialisation code sets up these bits as open drain outputs.

The driver code in I2CLPCBB.FTH must be compiled before the generic drive code in I2CBASE.FTH and the device specific files (e.g. AT24C512.FTH) are compiled.

```
0 equ TestI2C? \ -- n
```

Set this equate non-zero to include the test code at the end of the file. This equate also controls the compilation of I2C test code in other files.

```
: initI2C \ --
```

Initialise the I2C hardware. Performed after reset.

### 11.2 Timing

```
#10 equ I2CPeriod \ -- us
```

Define the I2C bit time in microseconds.

```
I2Cperiod system-speed 1000000 */ equ I2Cclocks \ -- n
```

The number of CPU clocks in a bit time.

```
1 equ clocks/ins \ -- n
```

The number of CPU clocks per instruction. Tune this to your hardware set up (memory width and speed). On most ARMs, a store takes 2 cycles, a load takes 3, and branch takes 3 and all others take 1. Look at the specific core documentation on the ARM technical reference CD for clock/instruction details. Look at your board schematics and setup code for details of the memory bandwidth. For example a 16 bit bus with two wait states and no cache may lead to six or more clocks per instruction.

```
#20 equ clocks/DO
```

the number of clocks needed to execute DO at the start of DO..LOOP including parameter passing and the call to (DO). This number may be sensitive to the compiler version and switch settings.

```
9 equ clocks/LOOP
```

the number of clocks needed to execute LOOP at the end of DO..LOOP. This number may be sensitive to the compiler version and switch settings.

```
I2Cclocks 4 / clocks/DO - clocks/LOOP / equ /I2Cqbit \ -- n
```

The number n for "n 0 DO LOOP" to generate an I2C quarter bit time.

```
: I2Cdelay \ --
```

Wait one quarter of an I2C bit. Tune the equates above for a one quarter bit time.

### 11.3 I2C bit functions

```
: read_scl \ -- bit
```

Read SCL bit and return in l.s. bit. No delays.

```
: write_scl \ b -- ; output clock bit
```

Write to SCL with no delays.

```
: SCL_low \ --
```

Set SCL low. A quarter cycle delay is performed before and after the transition.

```
: SCL_high      \ --
```

Set SCL high. A quarter cycle delay is performed before and after the transition.

```
: read_sda      \ -- bit
```

Read SCL bit and return in l.s. bit.

```
: write_sda     \ b -- ; output to SDA
```

Write the SDA bit.

```
: sda_low       \ --
```

Set SDA low.

```
: sda_high      \ --
```

Set SDA high.

## 12 I2C generic primitives

The code in I2CBASE.FTH provides the following words for constructing drivers for I2C devices:

```
START_I2C
    Generate a START condition
STOP_I2C  Generate a STOP condition
READ_BYTE_I2C
    Read a byte from the I2C lines
WRITE_BYTE_I2C
    Write a byte to the I2C lines
+ACK_I2C  Generate and test the ACK bit
-ACK_I2C  Do not generate and test the ACK bit.
```

The code assumes that bit banging primitives are used to generate the I2C signals, and that these are provided by low level code. Examples of the low level code are provided in files named I2CxxxDRV.FTH where xxx indicates the hardware.

```
: start_i2c      \ --
Define a start condition, bring SDA high-low while SCL is high
: stop_i2c       \ --
Define a stop condition, bring SDA low-high while SCL is high
1 value ack_i2c?      \ -- n
If a non-zero value is used, the system will generate the ACK bit on SDA for reads, otherwise
SDA is left high. For a non-zero value on writes, the system will check the ACK bit and THROW
if it is non-zero.
: +ack_i2c       \ --
check ACK cycles.
: -ack_i2c       \ --
Do not check ACK cycles.
: read_byte_i2c  \ -- in_byte
Read a byte, MSB first. If the value I2C_ACK? is non-zero, pull SDA low during the ACK bit,
otherwise leave it high
: write_byte_i2c \ byte --
Write a byte by shifting, MSB first. If ACK_I2C? is set, read ack bit on SDA, and THROW if
the ACK bit is high.
```





## 13 AT24C512 I2C driver

### 13.1 Introduction

The code for this 64k\*8 I2C EEPROM was tested on an Atmel EB55 and the MPE ARM USB Stamp. Note that prep-production USB Stamps were fitted with 24C128s rather than 24C512s. The 24C128 is code compatible except for overall and page sizes.

### 13.2 24C512 primitives

`$0A0 constant 24c512_id0 \ -- $A0 ; up to four devices`

The base device address (devaddr) of a 24C512.

`$10000 constant 24c512_size \ -- 64k`

Memory size of a 24C512 device.

`$080 constant 24c512_page \ -- 128`

Size of a 24C512 page to write.

`#10 constant 24c512_twr \ -- n ;`

The write time in milliseconds at 2.7v or above. Note that at 1.8v, this value must be increased to 20 ms.

`: 24c512_send_addr \ Eaddr devaddr --`

Issue a device address and EEPROM address.

`: 24c512_write_byte \ byte Eaddr devaddr --`

Write byte to the EEPROM devaddr at EEPROM address Eaddr. Note that the system must wait at least 24C512.TWR milliseconds before accessing the same device again.

`: 24c512_write_page \ src len Eaddr devaddr --`

Write len bytes of memory at src to the EEPROM devaddr starting at EEPROM address Eaddr. If len is 0, the address Eaddr is written to the EEPROM and becomes the current address, although no data is written. Note that the system must wait at least 24C512.TWR milliseconds before accessing the same device again.

`: 24c512_read_byte \ Eaddr devaddr -- byte`

Using EEPROM devaddr, read the byte at EEPROM address Eaddr.

`: 24c512_read_block \ dest len Eaddr devaddr --`

Using EEPROM devaddr, read len bytes at EEPROM address Eaddr to the buffer at dest.

`: 24c512_read_seq \ dest len devaddr --`

Read len bytes to dest from the current/next EEPROM address.

### 13.3 Simple utilities

These are simple words to read and write data into the serial EEPROM. They are not optimised for speed, but they do permit easy use. Much faster operation may be achieved using SEEPROM@ and SEEPROM! below.

`: eeb@ \ eaddr -- b`

Read a byte (8 bits) from address EADDR in the chip.

`: eew@ \ eaddr -- w`

Read a word (16 bits) from address EADDR in the chip.

```

: eel@      \ eaddr -- x
Read a long (32 bits) from address EADDR in the chip.
: eeb!      \ b eaddr --
Write a byte (8 bits) to address EADDR in the chip.
: eew!      \ w eaddr --
Write a word (16 bits) to address EADDR in the chip.
: eel!      \ x eaddr --
Write a long (32 bits) to address EADDR in the chip.

```

### 13.4 EDSL primitives

```

: NextBinAlign \ addr n -- addr'
Align an address to the next n-byte boundary, where n must be a power of 2.
: 24c512_read_len \ len seprom -- len'
Determine how much can be read in one go.
: 24c512_write_len \ len seprom -- len'
Determine how much can be written in one go.

```

### 13.5 EDSL functions

These functions depend on the EEPROM page size.

```

: SEEPROM! \ sram-addr length(bytes) seprom-addr --
Write length bytes to EEPROM address seprom-addr from the buffer at sram-addr.
: SEEPROM@ \ sram-addr length(bytes) seprom-addr --
Read length bytes at EEPROM address seprom-addr to the buffer at sram-addr.

```

## 14 Software Floating Point

### 14.1 Introduction

Although most embedded applications only require integer arithmetic, some do require floating-point. Therefore software floating-point is supplied with the cross-compiler and the target Forth. The target floating point wordset is not fully ANS compliant, but satisfies the needs of embedded systems without undue complexity. The Forth data stack and the floating point stack are the same. The floating point data storage format is not IEEE format, but is optimised for performance on small controllers. If you need a separate floating point stack or IEEE format storage, please contact MPE. Any variations in the implementation will be documented in the target specific section of the manual.

The cross-compiler has a more limited floating-point support than the target, this means that some words are available within colon definitions, but not outside them.

### 14.2 Source code

The source code is in two sets of files, one for 32 bit Forth targets, the other for 16 bit targets. The files are:

COMMON\SFP32HI	32 bit primitives
COMMON\SFP32COM	32 bit high level code
COMMON\SFP16HI	16 bit primitives
COMMON\SFP16COM	16 bit high level code

These files use no assembler definitions. Some targets have code versions of the primitives, and these will be found in the CPU specific code directory. A significant increase in performance can be obtained by using the code files.

### 14.3 Entering floating-point numbers

Floating-point numbers can be entered in two forms, 1.234 and 0.1234e1. Floating-point numbers are compiled as literal numbers when in a colon definition and placed on the cross-compiler's stack when outside a definition.

### 14.4 The form of floating-point numbers

A floating-point number is placed on the Forth data stack. In the Forth literature, this is referred to as a combined floating point and data stack. For 32 bit targets, a floating point number consists of two 32-bit numbers, one for the mantissa and one for the exponent. For 16 bit targets, it consists of a 32-bit double mantissa and a single 16-bit exponent. The mantissa is normalised. The exponent is on the top of the stack. Note that for 16 bit targets, number conversion is affected by the cross-compiler directives HOST-MATH and TARGET-MATH. HOST-MATH leaves double numbers and floats in 32-bit form, whereas TARGET-MATH leaves them in 16-bit form.

## 14.5 Creating variables

To create a variable, use `FVARIABLE`. `FVARIABLE` works in the same way as `VARIABLE`. For example, to create a floating-point variable called `VAR1` you code:

```
FVARIABLE VAR1
```

When `VAR1` is used, it returns the address of the floating-point number.

## 14.6 Accessing variables

Two words are used to access floating-point variables, `F@` and `F!`. These are analogous to `@` and `!`.

## 14.7 Creating constants

To create a floating-point constant, use `FCONSTANT`. `FCONSTANT` is analogous to `CONSTANT`. For example, to generate a floating-point constant called `CON1` with a value of 1.234, you enter:

```
1.234 FCONSTANT CON1
```

When `CON1` is executed, it returns 1.234 on the Forth stack.

## 14.8 Using the supplied words

The supplied words split into several groups:

- . sines, cosines and tangents
- . arc sines, cosines and tangents
- . arithmetic functions
- . logarithms
- . powers
- . displaying floating-point numbers
- . inputting floating-point numbers

The following functions only exist as target words so you cannot use them in calculations in your source code when outside a colon definition.

### 14.8.1 Calculating sines, cosines and tangents

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

### 14.8.2 Calculating arc sines, cosines and tangents

To calculate arc sine, cosine and tangent, use `FASIN`, `FACOS`

and `FATAN` respectively. They return an angle in radians.

### 14.8.3 Calculating logarithms

Two words are supplied to calculate logarithms, FLOG and FLN. FLOG calculates a logarithm to base 10 (decimal). FLN calculates a logarithm to base e. Both take a floating-point number in the range from 0 to EINF.

### 14.8.4 Calculating powers

Three power functions are supplied:

```
FE^X F10^X X^Y
```

## 14.9 Degrees or radians

The angular measurement used in the trigonometric functions are in radians. To convert between degrees and radians use RAD>DEG or DEG>RAD. RAD>DEG converts an angle from radians to degrees. DEG>RAD converts an angle from degrees to radians.

## 14.10 Displaying floating-point numbers

Two words are available for displaying floating-point numbers, F. and E.. The word F. takes a floating-point number from the stack and displays it in the form xxxx.xxxxx or x.xxxxxEyy depending on the size of the number. The word E. displays the number in the latter form.

## 14.11 Changes from v6.0 to v6.1

Renamed DINT to F>D for consistency. F>D is the ANS word. The original F>D was just a synonym. Similarly SINT was renamed to F>S.

The word FLOATS that enabled floating point number conversion has been renamed to REALS to avoid a name conflict with the ANS word of the same name.

The F-PACK vocabulary has been removed as no one liked it, and it could be considered contrary to the ANS Forth specification. If you wish to retain the F-PACK vocabulary, add the following lines before and after the compilation of the floating point code:

```
only forth definitions      \ *** added ***
vocabulary f-pack          \ *** added ***
also f-pack definition     \ *** added ***
include %CommonDir%\Sfp32Hi \ primitives
include %CommonDir%\Sfp32Com \ common high level code
previous definitions       \ *** added ***
```

The code enabling floating point to work in degrees or radians has been commented out for ANS compatibility. All trig functions now operate in radians. The commented out code may be uncommented if you need backward compatibility.

### 14.11.1 32 bit targets: software floating point

Overhauled 32 bit software floating point and incorporated improvements contributed by Hiden Analytical. These include more complete special case detection, faster high level code, and more accurate number input and output.

Removed all use of global variables except PLACES to make the floating point code usable in interrupt routines and in multitasked systems. If the output routines are to be multitasked, change the definition of PLACES from:

```
VARIABLE PLACES 8 PLACES !
```

to:

```
CELL +USER PLACES
```

and remember to initialise PLACES before using the floating point output routines.

Many words that are only useful as factors have been made headerless to save target memory space.

### 14.11.2 16 bit targets: software floating point

Note that the 16 bit floating point pack is not re-entrant. If you need to use the floating point pack in a multitasking system, you should convert the global variables to USER variables. The word +USER can be used

```
<size> +USER <name>
```

to define a USER variable of a given size (normally a CELL) at the next free offset in the USER area. Only PLACES will need initialisation.

## 14.12 Glossary

### 14.12.1 Basic stack and memory operators

```
: F!          \ r addr --
```

Stores r at addr

```
: F@          \ addr -- r
```

Fetches r from addr.

```
: F,          \ r --
```

Lays a real number into the dictionary, reserving 8 bytes.

```
: FDUP        \ r -- r r
```

Floating point equivalent of DUP.

```
: FOVER       \ r1 r2 -- r1 r2 r1
```

Floating point equivalent of OVER.

```
: FROT        \ r1 r2 r3 -- r2 r3 r1
```

Floating point equivalent of ROT.

```
: FPICK       \ fu..f0 u -- fu..f0 fu
```

Floating point equivalent of PICK.

: FROLL            \ f1 f2 f3 -- f2 f3 f1

Floating point equivalent of ROLL.

: FSWAP            \ r1 r2 -- r2 r1

Floating point equivalent of SWAP.

: FDROP            \ r --

Floating point equivalent of DROP.

: FNIP             \ r1 r2 -- r2

Floating point equivalent of NIP.

### 14.12.2 Floating point defining words

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

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

: FCONSTANT        \ r "<spaces>name" -- ; Run: -- r

Use in the form: <float> FCONSTANT <name> to create a constant that will return a floating point number.

: FARRAY            \ "<spaces>name" fn-1..f0 n -- ; Run: n -- rn

Use in the form: n FARRAY <name> to create a variable that will hold a default floating point number. When the array name is executed, the index i is used to return the address of the i'th 0 zero-based element in the array. For example, 5 FARRAY TEST will set up 5 array elements each containing 0, and then f n TEST F! will store f in the nth element, and n TEST F@ will fetch it.

### 14.12.3 Type conversions

: NORM             \ n exp -- f

Normalise a single integer and a single exponent to produce a floating point number. INTERNAL.

: DNORM            \ d exp -- fn ; normalise a 64 bit double

Normalise a double integer and a single exponent to produce a floating point number. INTERNAL.

: FSIGN            \ fn -- |fn| flag ; true if negative

Return the absolute value of fn and a flag which is true if fn is negative.

: F>S              \ fn -- n

Converts a float to a single integer. Note that F>S truncates the number towards zero according to the ANS specification. If |fn| is greater than maxint, +/-maxint is returned.

: F>D              \ fn -- d

Converts a float to a single integer. Note that F>D truncates the number towards zero according to the ANS specification. If |fn| is greater than dmaxint, +/-dmaxint is returned.

: FINT             \ f1 -- f2

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

: S>F              \ n -- fn

Converts a single integer to a float.

: D>F              \ d -- fn

Converts a double integer to a float.



#### 14.12.4 Arithmetic

: FNEGATE        \ r1 -- r2

Floating point negate.

: ?FNEGATE       \ fn n -- fn|-fn

If n is negative, negate fn.

: FABS            \ fn -- |fn|

Floating point absolute.

: F\*              \ r1 r2 -- r3

Floating point multiply.

: F/              \ r1 r2 -- r3

Floating point divide.

: F+              \ r1 r2 -- r3

Floating point addition.

: F-              \ r1 r2 -- r3

Floating point subtraction.

: FSEPARATE      \ f1 f2 -- f3 f4

Leave the signed integer quotient f4 and remainder f3 when f1 is divided by f2. The remainder has the same sign as the dividend.

: FFRAC           \ f1 f2 -- f3

Leave the fractional remainder from the division f1/f2. The remainder takes the sign of the dividend.

#### 14.12.5 Relational operators

: F0<             \ f1 -- flag

Floating point 0<.

: F0>             \ f1 -- flag

Floating point 0>.

: F0=             \ f1 -- flag

Floating point 0=.

: F0<>            \ f1 -- flag

Floating point 0<>.

: F=              \ f1 f2 -- flag

Floating point =.

: F<              \ r1 r2 -- flag

Floating point <.

: F>              \ f1 f2 -- flag

Floating point >.

: FMAX            \ r1 r2 -- r1|r2

Floating point MAX.

: FMIN            \ r1 r2 -- r1|r2

Floating point MIN.

### 14.12.6 Rounding

`f# 1.0 fconstant %ONE`

Floating point 1.0.

`: FLOOR \ r1 -- r2`

Floored round towards -infinity.

`: FROUND \ r1 -- r2`

Round the number to nearest or even.

### 14.12.7 Miscellaneous

`: FALIGNED \ addr -- f-addr`

Aligns the address to accept an 8-byte float.

`: FALIGN \ --`

Aligns the dictionary to accept an 8-byte float.

`: FDEPTH \ -- +n`

Returns the number of floats on the stack.

`: FLOAT+ \ f-addr1 -- f-addr2`

Increments `addr` by 8, the size of a float.

`: FLOATS \ n1 -- n2`

Returns `n2`, the size of `n1` floats.

### 14.12.8 Floating point output

`1 s>f 10 s>f f/ fconstant %.1`

Floating point 0.1.

`1 s>f fconstant %1`

Floating point 1.0.

`10 s>f fconstant %10`

Floating point 10.0.

`1250000000 34 fconstant %10^10`

Floating point  $10^{10}$ .

`1844674407 -33 fconstant %10^-10`

Floating point  $10^{-10}$ .

`F# 1.0E256 FCONSTANT %10^256`

Floating point  $10^{256}$ .

`F# 1.0E-1 FCONSTANT %10E-1`

Floating point  $10^{-1}$ .

`F# 1.0E-10 FCONSTANT %10E-10`

Floating point  $10^{-10}$ .

`F# 1.0E-256 FCONSTANT %10^-256`

Floating point  $10^{-256}$ .

`16 FARRAY POWERS-OF-10E1`

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

`17 FARRAY POWERS-OF-10E16`

An array of 17 powers of ten starting at  $10^0$  in steps of 16.

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

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

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

An array of 17 powers of ten starting at  $10^0$  in steps of -16.

```
: RAISE_POWER \ mant exp -- mant' exp'
```

Raise the power in preparation for number formatting.

```
: SINK_FRACTION \ mant exp -- mant' exp'
```

Reduce the power in preparation for number formatting.

```
variable places 8 places ! \ -- addr
```

Number of digits output after the decimal point.

```
: ROUND \ f1 -- f2
```

Rounds least significant eight bits to 0 if higher 2 bits are all 0s or all 1s.

```
: ?10PWR \ exp[2] -- exp[2] exp[10]
```

Generate the power of ten corresponding to the power of two. INTERNAL.

```
: SIGFIGS \ fn n -- d dec_exponent
```

From fn, generate a double number corresponding to n significant digits and a decimal exponent. INTERNAL.

```
: op-prepare \ fn -- d exp sign
```

From fn, generate a double number corresponding to n significant digits, a decimal exponent and a sign indicator (nz=negative). INTERNAL.

```
: .EXP \ exp --
```

Display the exponent. INTERNAL.

```
: N# \ d n -- d'
```

Convert n digits. INTERNAL.

```
: E. \ n exp --
```

Print the f.p. number on the stack in exponential form, x.xxxxxEyy.

```
: REPRESENT \ r c-addr u -- n flag1 flag2
```

Assume that the floating number is of the form +/-0.xxxxxEyy. Place the significand xxxxx at c-addr with a maximum of u digits. Return n the signed integer version of yy. Return flag1 true if f is negative, and return flag2 true if the results are valid. In this implementation all errors are handled by exceptions, and so flag2 is always true.

```
: F. \ f --
```

Print the f.p. number in free format, xxxx.yyyy, if possible. Otherwise display using the x.xxxxxEyy format.

### 14.12.9 Floating point input

```
: FLITERAL \ Comp: r -- ; Run: -- r
```

Compiles a float as a literal into the current definition. At execution time, a float is returned. For example, [ %PI F2\* ] FLITERAL will compile 2PI as a floating point literal. Note that FLITERAL is immediate.

```
: CONVERT-EXP \ c-addr --
```

If the character at c-addr is 'D' convert it to 'E'. INTERNAL.

```
: CONVERT-FPCHAR \ c-addr --
```

Convert the f.p. char '.' to the double char ',' for conversion. INTERNAL.

```
: ALL-BLANKS? \ c-addr len -- flag
```

Return true if string is all blanks (spaces). INTERNAL.

```
: FCHECK      \ -- am lm ae le e-flag .-flag
```

Check the input string at PAD, returning the separated mantissa and exponent flags. The e-flag is returned true if the string contained an exponent indicator 'E' and the .-flag is returned true if a '.' was found. INTERNAL.

```
: MNUM        \ c-addr u -- d 2 | 0
```

Convert the mantissa string to a double number and 2. If conversion fails, just return 0. INTERNAL.

```
: ENUM        \ c-addr u -- n 1 | 0 ; str as above
```

Convert the mantissa string to a single number and 1. If conversion fails, just return 0. INTERNAL.

```
: *10^X       \ float dec_exponent -- float'
```

Generate float' = float \*10^dec\_exp. INTERNAL.

```
: FIXEXP      \ dmant exp -- mant' exp'
```

Convert a double integer mantissa and a single integer exponent into a floating point number. INTERNAL.

```
: FNUMBER?    \ addr -- 0/.../mant exp 2
```

Behaves like the integer version of NUMBER? except that if the number is in F.P. format and BASE is decimal, a floating point conversion is attempted. If conversion is successful, the floating point number is left on the float stack and the result code is 2.

```
: >FLOAT      \ c-addr u -- r true|false
```

Try to convert the string at c-addr/u to a floating point number. If conversion is successful, flag is returned true, and a floating number is returned on the float stack, otherwise just flag=0 is returned.

```
: (F#)        \ addr -- fn 2 | 0
```

The primitive for F# and F#IN below.

```
: F#IN        \ -- fn 2 | 0
```

Attempts to convert a token from the input stream to a floating-point number. Numbers in integer format will be converted to floating-point. An indicator (0 or 2/3) is returned in the same way as an indicator is returned by FNUMBER?.

```
: F#          \ -- [f] ; or compiles it [ state smart ]
```

If interpreting, takes text from the input stream and, if possible converts it to a f.p. number on the stack. Numbers in integer format will be converted to floating-point. If compiling, the converted number is compiled.

```
: REALS       \ -- ; allow f.p input
```

Switch NUMBER? to permit floating point input using FNUMBER?. This action can be reversed by INTEGERS. Both REALS and INTEGERS are in the FORTH vocabulary.

```
: INTEGERS    \ -- ; no f.p input
```

Switch NUMBER? to restore integer only input.

### 14.12.10 Trigonometric functions

N.B. All angles are in radians.

```
: DEG>RAD     \ n1 -- n2
```

Convert degrees to radians.

```
: RAD>DEG     \ n1 -- n2
```

convert radians to degrees.

```
: FSIN          \ f1 -- f2
```

$f2 = \sin(f1)$ .

```
: FCOS          \ f1 -- f2
```

$f2 = \cos(f1)$ .

```
: FTAN          \ f1 -- f2
```

$f2 = \tan(f1)$ .

```
: FASIN         \ f1 -- f2
```

$f2 = \arcsin(f1)$ .

```
: FACOS         \ f1 -- f2
```

$f2 = \arccos(f1)$ .

```
: FATAN         \ f1 -- f2
```

$f2 = \arctan(f1)$ .

### 14.12.11 Power and logarithmic functions

```
: FLN           \ f1 -- f2
```

Take the logarithm of  $f1$  to base  $e$  and return the result.

```
: FLOG          \ f1 -- f2
```

Take the logarithm of  $f1$  to base 10 and return the result.

```
: FE^X          \ f1 -- f2
```

$f2 = e^{f1}$ .

```
: F10^X         \ f1 -- f2
```

$f2 = 10^{f1}$

```
: FX^N          \ x-real n-integer -- fx^n
```

$fx^n = x^n$  where  $x$  is a float and  $n$  is an integer.

```
: FX^Y          \ x-real y-real -- fn
```

$fn = X^Y$  where  $X$  and  $Y$  are both floats.

```
: FSQR          \ f1 -- f2 ; FSQR by Heron's formula
```

$F2 = \text{sqrt}(f1)$  by Heron's formula.

### 14.13 ARM coded primitives

The software floating point pack requires several support primitives. High level versions are provided in SFP16HI.FTH and SFP32HI.FTH for 16 and 32 bit targets. Some targets have coded versions in the CPU directory and these will provide much better performance. The support file should be compiled before the common file.

```
code <-S        \ n1 carry-in-flag -- n2 carry-out-flag
```

Perform a left shift, applying the carry in to the l.s. bit and returning the carry out as 1 or 0.

```
code S->        \ n1 carry-in-flag -- n2 carry-out-flag
```

Perform a right shift, applying the carry in to the m.s. bit and returning the carry out as 1 or 0.

```
code d<<1       \ xd -- xd<<1
```

One bit double left shift.

```
code d>>1       \ xd -- xd>>1
```

One bit double right shift.

```
code D>>N      \ d m -- d>>m ; SFP002
```

M bit double right shift.



## 15 Periodic Timers

This code provides a timer system that allows many timers to be defined, all slaved from a single periodic interrupt. The Forth words in the user accessible group documented below are compatible with the token definitions for the PRACTICAL virtual machine, with the code supplied with MPE's embedded targets, and with VFX Forth. This code assumes the presence of a global value TICKS which holds a time value incremented in milliseconds. The timebase is approximate, and granularity and jitter are affected by the timer ISR and the time taken by your own code to execute. By default, the timer is set to run every 100ms. The source code is in the file TIMEBASE.FTH.

The timer chain is built using a buffer area, and two chain pointers. Each timer is linked into either the free timer chain, or into the active timer chain.

All time periods are in milliseconds. Note that on a 32 bit system such as ProForth VFX, these time periods must be less than  $2^{31}-1$  milliseconds, say 596 hours or 24 days, whereas if the code is on a 16 bit system, time periods must be less than  $2^{15}-1$  milliseconds, say 32 seconds.

### 15.1 The basics of timers

These basic words are defined for applications to use the timer system. Other words are detailed elsewhere in this chapter.

```
START-TIMERS    \ -- ; must do this first
STOP-TIMERS     \ -- ; closes timers
AFTER           \ xt period -- timerid/0 ; runs xt once after period ms
EVERY           \ xt period -- timerid/0 ; runs xt every period ms
TSTOP          \ timerid -- ; stops the timer
MS              \ period -- ; wait for period ms
```

After the timers have been started, actions can be added. The example below starts a timer which puts a character on the debug console every two seconds. Note that when using generic I/O, the output and input devices MUST be specified.

```
start-timers
: t      \ -- ; will run every 2 seconds
  console opvec !
  [char] * emit
;
' t 2 seconds every          \ returns a timer id, use TSTOP to stop it
```

The item on stack is a timer handle, use TSTOP to halt this timer.

AFTER is very useful for creating timeouts, such as required to determine if something has happened in time. AFTER returns a timerid. If the action you are protecting happens in time, just use TSTOP when the action happens, and the timer will never trigger. If the action does not happen, the timer event will be triggered.



## 15.2 Considerations when using timers

All timers are executed within a single interrupt, and so all timer action words share a common user area. This has some impact on timer action words. Since you do not know in which order timer action words are executed, you must set up any USER variables such as BASE that you may use, either directly or indirectly.

The interrupt that handles all the timers does not set IPVEC and OPVEC to a default value. If you are going to use Forth I/O words such as EMIT and TYPE within a timer action, you MUST set IPVEC and OPVEC before using the I/O. For the sake of other timer action routines that may still be using default I/O, it is polite to save and restore IPVEC and OPVEC in your timer action words.

Do not worry about calling TSTOP with a timerid that has already been executed and removed from the active timer chain; if TSTOP cannot find the timer, it will ignore the request.

Under some conditions, the execution time of all the timer routines may be longer than the requested period of the timer. In addition, the timer interrupt may be subject to jitter.

## 15.3 Implementation issues

The following discussion is relevant if you want to modify this code. Functionally equivalent code is provided with MPE's VFX Forth systems. In the Windows environment, timer interrupts are implemented by callbacks and critical sections.

By default, the word DO-TIMERS is run from within the periodic timer interrupt. If interrupts are not re-enabled after resetting the timer interrupt, you may have latency issues if a number of timers is used, or if the timer routines take a considerable time. In this case, it would be better to set up the timer routine to RESTART a task, which calls DO-TIMERS, e.g.

```
: TIMER-TASK    \ --
  <initialise>
  BEGIN
    DO-TIMERS STOP
  AGAIN
;
```

Such a strategy also permits you to use a fast interrupt, say 1ms, for the clock, and to trigger the TIMER-TASK every say 32 ms.

## 15.4 Timebase glossary

0 value ticks \ -- addr ; holds timer count

Get current clock value in milliseconds.

#8 constant #timers \ -- n ; maximum number of timers

A constant used at compile time to set the maximum number of timers required. Each timer requires RAM as defined by the ITIMER structure.

: after \ xt period -- timerid/0 ; xt is executed once,

Starts a timer that executes once after the given period. A timer ID is returned if the timer could be started, otherwise 0 is returned.

`: every`            `\ xt period -- timerid/0 ; xt is executed periodically`  
Starts a timer that executes every given period. A timer ID is returned if the timer could be started, otherwise 0 is returned. The returned timerID can be used by TSTOP to stop the timer.

`: tstop`            `\ timerid --`  
Removes the given timer from the active list.

`: pause`            `\ -- ; multitasker hook`  
Allows the system multitasker to get a look in. Under Windows this also allows the message queue to be handled.

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

`: expired`          `\ n -- flag ; true if timed out`  
Flag is returned true if the timebase value n has timed out. N.B. Calls PAUSE.

`: timedout?`        `\ n -- flag ; true if timed out`  
Flag is returned true if the timebase value n has timed out. Does not call PAUSE, so can be used in interrupts, winprocs and callbacks. In particular, TIMEDOUT? should be used rather than EXPIRED inside timer action words to reduce timer jitter.

`: ms`                `\ n --`  
Waits for n milliseconds. Uses PAUSE through EXPIRED.



## 16 LPC210x Ticker Interrupt

The ticker is the basis of the TIMEBASE system. In order start the ticker the START-CLOCK word is used. This initialises the TC0 timer counter channel to produce a periodic interrupt which updates the value TICKS. The period is set by the equate TICK-MS in milliseconds, and the value of TICKS is updated by TICK-MS at every interrupt.

Other files named TICKxxx in the DRIVERS folder provide the same functionality for other CPUs.

For the LPC210x, the default ticker uses Timer 0, Match Register 0 being used to reset the count and generate an interrupt. Note that if the PWM outputs are not being used, the PWM unit can be used as it includes the same register set as Timer 0 and Timer 1.

This code requires the interrupt facilities in INTARM3.

### 16.1 Configuration equates

```
_timer0 equ _ticker      \ -- addr
```

Select the timer/counter unit to use for the ticker

```
BIT1 equ TickerBit      \ -- mask
```

The bit to be set in PCONP to power the counter/timer.

```
t0int# equ TickInt#     \ -- src
```

select the source number of the ticker interrupt.

```
#15 equ TickSlot#      \ -- n
```

Select the VIC slot/priority for the ticker interrupt. This is usually the lowest priority interrupt in slot 15.

```
1 equ /preTick        \ -- n
```

Select the prescaler division ratio.

```
/pretick 1- equ init_preDiv
```

The initial value of the prescaler divider.

```
: gen-ticks          \ us clock -- n
```

An INTERPRETER definition that takes a period in microseconds and the clock speed in herz to produce the required value to be loaded in the match register.

```
tick-ms #1000 * system-speed gen-ticks equ #Mrtick
```

The initial value of the match register calculated from the required ticker period TICK-MS in milliseconds and the system clock speed SYSTEM-SPEED in herz.

### 16.2 Ticker interrupt handler

```
0 value ticks        \ -- u ; returns the ticker value
```

Return current clock value in milliseconds. This value can treated as a 32 bit unsigned value that will wrap when it overflows.

```
: ticker-isr        \ -- ; high level interrupt handler
```

The high level interrupt handler for the ticker.

```
' ticker-isr IRQ: ticker-irq    \ -- addr
```

Creates the IRQ handler for the ticker.

```
: start-clock \ -- ; start clock interrupt
```

Start the ticker periodic interrupt.

```
: stop-clock \ -- ; stop clock interrupt
```

Stop the ticker periodic interrupt.

```
: ms \ n --
```

Waits for n milliseconds.

### 16.3 Time base extensions

```
: start-timers \ -- ; Start internal time clock
```

Initialise the TIMEBASE system.

```
: stop-Timers \ -- ; disable timer system
```

Stop the TIMEBASE ticker.

## 17 ARM multitasker

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

### 17.1 Configuration - normally performed earlier

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

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

### 17.2 TCB data structure layout

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

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

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

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

### 17.3 Task handling primitives

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

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

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

Returns true if the tasker is enabled.

```
: single \ --
```

Disable scheduler.

```
: multi \ --
```

Enable scheduler.

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

The software scheduler itself.

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

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

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

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

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

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

```
: stop \ -- ; halt oneself
```

HALT's the current task, and executes PAUSE.

## 17.4 Event handling

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

```
: set-event      \ task --
```

Set the event trigger in task TCB.

```
: event?        \ task -- flag
```

Returns true if true if task has received an event trigger which has not been cleared yet.

```
: clr-event-run \ --
```

Reset the current task's `EVENT_RUN` flag.

```
: to-event      \ xt task -- ; define action of a task
```

Sets XT as the event handler for the task.

## 17.5 Message handling

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

```
: msg?          \ task -- flag
```

Returns true if task has received a message.

```
: send-message  \ addr task --
```

Send a message to a task.

```
: get-message   \ -- addr task
```

Wait for any message and return the message and the task it came from.

```
: wait-event/msg \ --
```

Wait for a message or an event trigger.

## 17.6 Task structure management

```
code init-task \ xt task -- ; Initialise a task stack
```

Initialise a task's stack before running it and set it to execute the word whose XT is given.

```
: add-task      \ task -- ; insert into list
```

Add the task to the list of tasks after the current task.

```
: sub-task      \ task -- ; remove task from chain
```

Remove the task from the task list.

```
: initiate      \ xt task -- ; start task from scratch
```

Start the given task executing the word whose XT is given, e.g.

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

```
: sleeper       \ xt task --
```

Start task from scratch, but leave it HALTED. Use in the form:

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

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

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

Stop a task, and remove it from the list.

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

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

```
: his \ task uservar -- addr ; produce address of user var in another task
```

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

## 17.7 semaphores

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

A SEMAPHORE is an extended variable used for signalling between tasks, and for resource allocation. The counter field is used as a count of the number of times the resource may be used, and the arbiter field contains the TCB of the task that last gained access. This field can be used for priority arbitration and deadlock detection/arbitration.

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

Creates a semaphore which returns its address at runtime. Use in the form:

```
Semaphore <name>
```

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

SIGNAL increments the counter field of a semaphore, indicating either that another item has been allocated to the resource, or that it is available for use again, 0 indicating in use by a task REQUEST waits until the counter field of a semaphore is non-zero, and then decrements the counter field by one. This allows the semaphore to be used as a COUNTED semaphore. For example a character buffer may be used where the semaphore counter shows the number of available characters. Alternatively the semaphore may be used purely to share resources. The semaphore is initialised to one. The first task to REQUEST it gains access, and all other tasks must wait until the accessing task SIGNALs that it has finished with the resource.

## 17.8 TASK and START:

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

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

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

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



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

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

```
variable task-chain    \ -- addr
```

Anchors list of all tasks created by TASK and friends.

```
: task                \ -- ; -- task ; TASK <name> builds a task
```

Note that the cross-interpreter's version of TASK has been modified from v6.2 onwards to leave the current section as CDATA.

```
: task                \ -- ; -- task ; TASK <name> builds a task
```

Creates a new task and data area, returning the address of the user area at run time. The task is also linked into the task chain anchored by TASK-CHAIN.

```
: start:              \ task -- ; exits from caller
```

Used inside a colon definition. The code following START: up to the ending semi-colon forms the action of the task. The word containing START: finishes at START:.

## 17.9 Debugging tools

```
: .task               \ task --
```

Display task's name if it has one.

```
: .tasks              \ task -- ; display all task names
```

Display all the tasks anchored by TASK-CHAIN.

```
: .running            \ --
```

Display running tasks.

## 18 Vocabulary and wordlist tools

: VOC?                \ wid -- flag

Return TRUE if 'wid' is actually a vocabulary.

: .VOC                \ wid --

If wid represents a vocabulary, display its name, otherwise just display its value.

: ORDER              \ --

Display the current search order and definitions vocabularies.

: VOCS                \ --

Display all vocabularies.

: \$FORGET            \ c-addr --

Forgets word name in given string. See FORGET.

: FORGET             \ "<spaces>name" --

Used in the form "FORGET <name>", <name> and all following words are removed from the dictionary. This word is marked obsolescent in the ANS specification, and is replaced by MARKER.

: MARKER             \ "<spaces>name" -- ; Exec: --

MARKER <name> creates a word that when executed removes itself and ALL following definitions from the dictionary. MARKER is the ANS replacement for FORGET. MARKER automatically trims all wordlist and vocabulary based chains.



## 19 ROM PowerForth utilities

### 19.1 Introduction

Supplied as source in the ROMFORTH directory are utilities to:

- . compile source code on your target board from the cross-compiler IDE
- . upload a binary image from your target to your PC
- . download a binary image to your target from your PC.

### 19.2 Compiling text files

Source text files can be compiled from the host PC onto the target system. This saves time in not having to cross-compile the entire source if a small modification is made. The utilities permit text file to be split into pages for better layout when printed. An ASCII Form Feed character (decimal 12) separates one page from another.

#### 19.2.1 The required files

To compile text files from your target board, cross-compile the files IODEF.FTH and TEXTFILE.FTH.

#### 19.2.2 Compiling a specified text file

To compile all or part of a specified text file onto your target, use INCLUDE in the form:

```
INCLUDE <filename>
```

This compiles the file <filename> into the target's dictionary. The file name must include any required extension AIDE's internal file server must be enabled (in the console window configuration), and will be triggered automatically.

### 19.3 XMODEM binary image download

Binary images can be downloaded to your PC using the XMODEM protocol.

#### Required files

To use this utility you must cross-compile the file COMMON\XMODEMTXRX.FTH.

#### Using the XMODEM binary download utility

To download a binary image from the target system to your PC, use BIN-DOWN in the form:

```
addr #bytes BIN-DOWN
```

where addr is the start address and #bytes is the number of bytes to download starting from addr. For example,

```
1200 400 BIN-DOWN
```

sends the area of memory from 1200 to 1599 to your host PC. AIDE's internal file server must be enabled (in the console window configuration), and will be triggered.

## 19.4 XMODEM binary image upload

Binary images can be uploaded from your PC using the XMODEM protocol.

### Required files

To use this utility you must cross-compile the file COMMON\XMODEMTXRX.FTH.

### Using the XMODEM binary upload utility

To download a binary image from the target system to your PC, use BIN-UP in the form:

```
addr #bytes BIN-UP
```

where addr is the start address and #bytes is the number of bytes to download starting from addr. For example,

```
1200 400 BIN-UP
```

loads the area of memory from 1200 to 1599 from your host PC. AIDE's internal file server must be enabled (in the console window configuration), and will be triggered.

## 19.5 IODEF.FTH

Before compiling this file, synonyms may need to be defined for SER-EMIT SER-KEY? and SER-KEY. Add these in the control file before compiling IODEF.FTH.

IODEF.FTH provides equates and protocol primitives for AIDE.

### 19.5.1 AIDE support

```
variable disk-error \ -- addr ; set non-zero on error
```

This variable is set true when a transfer error occurs.

```
: wait-ack \ -- ; wait for ACK character
```

This word waits for the host to send an ACK at the end of part of a transfer. INTERNAL.

```
: wait-ack/nack \ -- t/f ; true for NACK
```

This waits for either a NACK or an ACK from the host and leaves true or false on stack. INTERNAL.

```
: send-block# \ n -- ; send block number to server
```

Sends a single length number as two bytes 00-FF, low byte first. INTERNAL.

```
: synch-to-host \ -- ; sync host to us
```

Waits for a START (0x01) character, flushes the input, and sends an ACK. INTERNAL. INTERNAL.

## 19.6 Miscellaneous

```
: cls          \ --
Clear PowerTerm screen
```

## 19.7 INCLUDE source code from AIDE

The file COMMON\ROMFORTH\TEXTFILE.FTH provides support for compiling a source file from the AIDE server. The code has been updated for AIDE version 2.500 onwards.

```
: end-load     \ -- ; switch back to keyboard input
```

This word is automatically performed at the end of a download to tidy up the comms.

```
: file-error   \ n --
```

Handle an error when a file is being INCLUDED.

```
: $include     \ $addr -- ; compile host file, counted string
```

Given a counted string representing a file name, compile the file from AIDE.

```
: include      \ "<filename>" -- ; load file from host
```

Compile a file across the serial line from the AIDE file server. Use in the form:

```
include <filename>
```

The filename extension must be supplied.

## 19.8 Simple source file loader

The code in COMMON\ROMFORTH\FILETRAN.FTH provides a simple source file loader which can be used with most terminal emulators. The download is controlled by XON/XOFF flow control. When using the PowerTerm terminal emulator in AIDE, use the INCLUDE <filename> system which supports nested files and needs no special termination.

Each file compiled must include a single line

```
END-UP-LOAD
```

at the end to reset the interpreter.

For slow 8 bit CPUs without queued serial input, the terminal server may need to include pacing delays after each character and an additional after CR/LF pairs.

```
: Up-Load     \ -- ; Load ASCII text
```

Compile a file delivered by the terminal emulator. This word is intolerant of compilation errors.

```
: End-Up-Load \ -- ; Finish Up-Loading
```

Used on the target to restore the Forth interpreter after a file has been compiled.



## 20 XMODEM Receiver and Transmitter

### 20.1 Introduction

This file implements the XMODEM 128-byte protocol in both directions. Use with AIDE requires AIDE release 2.500 upwards.

No test code is provided for this file as the system has been tested by comparison of transferred binary files.

### 20.2 Words in XMODEMTXRX.FTH

#### 20.2.1 Configuration

```
1 equ XmodemTx? \ -- n
Non-zero to compile transmit code

1 equ XmodemRx? \ -- n
Non-zero to compile receive code.
```

#### 20.2.2 Constants and variables

```
$0101 equ blkerror
A block number error has occurred.

$0103 equ noreply
There was no reply within one second.

$0104 equ crcerror
Bad CRC or checksum.

$0105 equ overflow
Too many blocks were sent.

$010 equ maxerrs          \ -- n
Maximum number of errors before transfer is aborted.

#128 Buffer: x-buffer     \ -- addr
Holds a 128 byte Xmodem data block.
```

#### 20.2.3 Common code

```
: init-blks              \ addr #bytes -- #blks
Given the size of an image, return the number of complete 128 byte blocks, set the variable CUR-ADDRESS to ADDR and set the variable BLK# to 1.
```

#### 20.2.4 XMODEM transmission

```
: (To-Buffer)          \ -- ; copy 128 bytes to X-BUFFER
Move 128 bytes from the memory pointed to by CUR-ADDRESS into X-BUFFER. This is the default action of TO-BUFFER. The variable CUR-ADDRESS is set by BIN-DOWN and friends.

Defer To-Buffer \ --
Copy the next 128 bytes to transmit to X-BUFFER. They are then transmitted from X-BUFFER. You can change this action as required by your application. The default action is (TO-BUFFER).
```



`: ?Ack \ -- ; wait for char, abort if not ACK`

Wait for a character and terminate the transfer and abort if the character is not an ACK.

`: Send-Block \ Blk# -- ; transmit a block`

Transmit the 128 byte contents of X-BUFFER to the host.

`: Bin-Down \ addr #bytes -- ; transfer memory to host`

Download (transmit) the given block of memory to the host using the XMODEM 128 byte block protocol. On entry, the variable CUR-ADDRESS is set to ADDR on entry and #bytes is rounded up to a 128 byte unit.

## 20.2.5 XMODEM reception

`: ser-flush \ -- ; flush the link input`

Flush all input character from the host/target link.

`: send-ack \ -- ; send ACK`

Transmit an ACK character.

`: send-nak \ -- ; Transmit a NAK character`

Transmit a NAK character.

`: send-can \ -- ; send CAN character`

Transmit a CAN character.

`: Get-BlockData \ -- cksum ; get block from host`

Receive a 128 byte XMODEM data block from the host.

`: toomanyerrs? \ -- T|F ; true if too many errors`

Return true if too many comms errors have occurred.

`: (From-Buffer) \ -- ; copy 128 bytes from X-BUFFER`

Move 128 bytes from X-BUFFER into the memory pointed to by CUR-ADDRESS

`Defer From-Buffer \ -- ; copy 128 bytes from X-BUFFER`

Move 128 bytes from X-BUFFER into the memory pointed to by CUR-ADDRESS. CUR-ADDRESS is set up by BIN-UP and friends. The default action is (FROM-BUFFER). You can modify the action to suit your own application.

`: Get-Block \ -- ;`

Receive an XMODEM 128 byte data block from the host, processing the header and checksum data.

`: waitforresponse \ -- ; wait for host to respond`

Wait for the host to respond for up to 1 second. If the host does not respond, a NOREPLY status is set in the variable RXSTATUS. This word requires the ticker interrupt to be running.

`: Bin-Up \ addr len -- status ; status 0 = GOOD`

Upload (receive) a block of data of the given size into memory using the XMODEM 128 byte block protocol. An error status is returned, 0 indicating success. On entry, the variable CUR-ADDRESS is set to ADDR on entry and len is rounded up to a 128 byte unit. Note that an error return of \$0105 indicates the the file being sent is larger than the LEN input parameter.

`: RecvXmodem \ addr len -- len' status ; status=0=good`

Upload (receive) a block of data of the given size into memory using the XMODEM 128 byte block protocol. The number of bytes correctly received and an error status are returned, 0 indicating success. See BIN-UP above for more details.

## 21 Access to the IAP routines

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

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

```
1 equ FullIAP?          \ -- n
```

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

```
5 cells buffer: IAPcmd \ -- addr ; max 5 cells
```

Command input buffer for IAP routines.

```
3 cells buffer: IAPres \ -- addr ; max 3 cells
```

Result output buffer from IAP routines.

```
code IAP              \ *cmd *res --
```

The primitive to call the IAP routines. Interrupts are disabled for the duration of the IAP call.

```
: IAPprep            \ start end -- res
```

Prepare sectors for erase/write.

```
: IAPcopy            \ Rsrc Fdest len -- res
```

Copy/program len bytes from Rsrc in RAM to Fdest in Flash.

```
: IAPerase          \ start end -- res
```

Erase the inclusive range of sectors.

```
: IAPcheck          \ start end -- res
```

Blank check the inclusive range of sectors.

```
: IAPcompare        \ Rsrc Fdest len -- res
```

Compare len bytes from Rsrc in RAM to Fdest in Flash.

```
: IAPbootver        \ -- bootver
```

Return the 16 bit boot code version. The high byte is the major version and the low byte is the minor version.

```
: IAPpartno         \ -- part#
```

Return the Philips part number.

### 21.1 Gotchas

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

```
IAPBootVer .dword
```

which will give something of the form:

0000:xyy

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

```
www.semiconductors.philips.com  
/files/products/standard/microcontrollers/utilities/  
lpc2000_flash_utility.zip  
lpc2000_bl_update.zip
```

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

## 22 LPC2000 Flash tools

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

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

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

### 22.1 Flash primitives

Although some of these routines are not the most efficient for the LPC2000 series, they are designed to be easily expanded for future LPC2xxx parts with as yet unknown sector sizes.

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

```
create sectab \ -- addr
LPC2106 and 2119 sector table.
```

```
create sectab \ -- addr
LPC2129 table.
```

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

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

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

### 22.2 Flash driver

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

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

```
: Prog512      \ src dest --
Program 512 bytes at src to dest. Increment error counter on error.
```

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

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

`: (ProgFlash) \ src dest len --`

Write len bytes from memory at src to Flash at dest. Note that the Flash is assumed to be erased, and that no verification is performed except when each byte is programmed. DEST is forced to a 512 byte boundary and LEN is rounded up to the next 512 byte unit. SRC must be on a word boundary.

`: (VerifyFlash) \ src dest len --`

Verify len bytes from memory at src to Flash at dest.

`: ProgramFlash \ src dest len --`

Using the RAM memory buffer at src, program the Flash at dest with len bytes. The relevant Flash sectors are erased, the Flash is programmed, and the result verified.

## 23 Reflashing the system

### 23.1 Introduction

If you destroy the application in the internal Flash, you must use the Philips ISP loader to reload an Intel Hex file supplied on the ARM Stamp CD. A suitable baud rate is 38400 baud. To use this, ensure that port P0.14 is low. There is a link on the board that (when installed) enforces this. Then reset the board and use the Philips loader. Then close the Philips loader. Ensure that the P0.14 link is removed before resetting the board. Then connect using AIDE's PowerTerm or HyperTerm.

Once the Forth system is running again, you can use the word `REFLASH ( -- )` to download a new binary image. Note that `REFLASH` only handles binary memory image files.

The LPC210x can only be reflashed from an application by a program running from RAM. A separate application built by a control file `REPROG\REPROG*.CTL` is used to do this. A binary image `REPROG*.IMG` is inserted into the application. When required, the code is copied into the internal RAM and executed from RAM.

Return is by rebooting the system. The contents of the internal RAM and Flash are destroyed, therefore any data that must be preserved should be saved externally to the chip. Alternatively, modify the reprogramming code to use sector 14 (`$0001:C000..0001:DFFF`) to hold data to be preserved.

### 23.2 Code in main application

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

```
create reprog210x.img \ -- addr
```

The start address of the reprogramming code

```
data-file %HwDir%\reprog\reprog210x.img equ /reprog \ -- len
```

The length of the reprogramming code loaded into the dictionary.

```
$40000200 equ reprog \ -- addr
```

The run time address of the reprogramming code. This **MUST** match the start address of the `CDATA` section defined in `REPROG210x.CTL`, and that section **MUST NOT** overlap the run-time stacks and user area.

```
: callit \ xt --
```

Load the reprogramming code and execute the xt.

```
: reflash \ -- ; no exit
```

Copies the reprogramming code to the run-time address and executes it.



## 24 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.





## 25 Creating turnkey applications

### 25.1 Introduction

Applications compiled on the Stamp boards are compiled into RAM. This area of RAM can be saved in the serial EEPROM (USB Stamp, 16k or 64k bytes) or CPU Flash (e.g. TiniARM) and reloaded at power up. Optionally a word can be executed at power up, so making a turnkey application.

An application is saved by the word `TURNKEY ( xt|0 -- )` which saves a header and the compiled image, including previously compiled code and data, into the serial EEPROM or Flash. If you do not want a word to run at power up, a value of 0 is used instead of an XT.

```
' MyApp turnkey    \ will execute MyApp at power up
0 turnkey          \ will just restore the image
```

At power up, the image is copied into RAM. If the internal CRC check fails, the interactive Forth is started but may eventually crash if the image is badly corrupted. If a turnkey word has been set, it will be executed. If the turnkey word returns, the interactive Forth is started.

If you want to remove a previously compiled image, do this with `EMPTY ( -- )`, which resets the header block

### 25.2 Saving applications

```
: Empty          \ --
```

USB Stamp: Erase the header information in the serial EEPROM. This does not modify the software in RAM. In order to run without the previously loaded software use `EMPTY` and then `REBOOT`.

```
: Empty          \ --
```

Serial Stamp, e.g. TiniARM: Erase the header information in the LPC2106 Flash at \$10000. This does not modify the software in RAM. In order to run without the previously loaded software use `EMPTY` and then `REBOOT`.

```
: Proglen        \ -- len
```

Gives the length of the compiled application.

```
CommitXt constant ProgStart    \ -- addr
```

Returns the start address of the RAM program

```
: +CRC16         \ crc b -- crc'
```

Update CRC16 for one byte

```
: genCRC         \ crc addr len -- crc'
```

Generate a CRC for a block of memory, given an initial CRC value.

```
: AppCRC         \ -- crc
```

Generate the application CRC.

```
: KernelCRC     \ -- crc
```

Generate the kernel CRC for the first 64k bytes of Flash.

```
: Commit        \ xt|0 --
```

Prepares the RAM header for saving.

```
: SaveApp      \ --
```

USB Stamp: Write the RAM image to the serial EEPROM.

```
: SaveApp      \ --
```

Serial Stamp, e.g. TiniARM: Write the RAM image to the Flash at 0001:0000h. The ticker interrupt is halted during this operation.

```
: TurnKey      \ xt|0 --
```

Initialise the program header and save the application to the serial EEPROM. An XT of 0 will cause the image to be reloaded without any start up action.

```
: Reload      \ --
```

Reload an application from the serial EEPROM.

```
: CheckApp     \ -- flag ; true if ok
```

Returns non-zero if the CRC of the application in RAM is good.

```
: CheckKernel  \ -- flag ; true if ok
```

Returns non-zero if the CRC of the kernel matches the kernel CRC saved in the application.

```
: ?Restart    \ --
```

Reload the application, check the CRC, and execute a correct application if required. This word is executed at power up. If the application cannot be installed, a warning message is issued and the Forth text interpreter will start.

## 25.3 Gotchas

Once a turnkey application has started, there is no way back to the interactive Forth for debugging unless you provide it. It is common practice to provide a backdoor for engineering and maintenance access.

Because of supply problems, the first ten engineering sample boards only have 16k of serial EEPROM. Do not try to save more than 16k in these systems.

Because the cold chain mechanism is executed before applications are reloaded into RAM, application programs may not use the word ATCOLD. Applications must explicitly perform any initialisation they require.

The saved image is a snapshot of RAM, including all system variables. If your application requires different initial conditions, it must explicitly set these variables.

## 25.4 Application License

This software is provided for use with boards manufactured by MicroProcessor Engineering Ltd (MPE) and New Micros Inc (NMI) only. If you want to run this code on other hardware, you MUST obtain a license from MicroProcessor Engineering Ltd.

Your application may provide access to the open Forth for what we term "engineering and maintenance access" only. If you need your clients to be able to do more than this, e.g. to create new words themselves, you MUST obtain a license from MicroProcessor Engineering Ltd. In many low volume cases, this just consists of purchasing development systems rather than production boards. The advantage of this is that your clients receive all the latest tools and

documentation. You can also choose to purchase an MPE cross compiler which enables you to add to and modify the contents of the Flash image, as well as producing smaller and much faster code.

Note that none of files on the development distribution may be redistributed without permission. If you have any questions or concerns about the licenses, please contact MPE directly.



## 26 Examples directory

The EXAMPLES directory contains much useful code, ranging from simple tools to fully documented extensions. The best way to use the EXAMPLES directory is to browse through the source code. If you want to modify the code, we recommend that you move it to become part of your own application directory structure.

Additional examples may have been added since this manual was generated. Browse the EXAMPLES folder to see what is there.

### 26.1 Main directory

The following is a list of files as of November 2002.

#### CALENDAR.FTH

A perpetual calendar by Christophe Lavarenne. A choice of calendars is provided.

#### DOUBLES.HI

This file implements double and some quad precision number support using the primitives of PowerForth and high level definitions. To obtain better performance some definitions should be coded. These are indicated in the source code.

#### HEXPAD.FTH

Keypad read routine for hex matrix keypad. The example was written for an 8051 port using four input bits and four output bits.

#### MATH.FTH

Miscellaneous math functions.

#### PRIMES.FTH

Eratosthenes sieve - simple prime benchmark.

#### SINCOS.FTH

Integer trig words from Kurt Heinz at Synics. These words provide a simple implementation of sine, cosine, and tangent functions.

#### TESTCODE.FTH

A test harness for verifying the stack effect of of Forth words and phrases.

#### UNIXTIME.FTH

Maintains a Unix style seconds counter.

### 26.2 Contributions subdirectory

This directory contains code contributed by users for others to use, and MPE thanks the contributors.

The contents of this directory are untouched by MPE who provide no warranty at all on this code. Sorry about that.

#### AD.FTH

68HC11 A/D handler.

#### CW.FTH

This program will display text in CW (Morse Code) upon either the system's console or the system's LEDs.

#### DATES.FTH

Conversions between calendar date and Julian day number from ACM# 199. Forth Scientific Library Algorithm #22

**HIDEN.FTH**

This code replaces REQUEST and SIGNAL in the MPE multitasker because they allow a task to lock a semaphore multiple times.

**IEEE.FTH** Converts between MPE software floating point format for 32 bit systems and IEEE 32 bit format.

**LANDER.FTH**

Lunar Landing Simulation.

## 26.3 I2C subdirectory

**I2CLOAD.BLD**

Build file for other I2C files.

**BCD.FTH** BCD to binary conversion and back

**I2CBASE.FTH**

I2C primitives. This file requires an I2C bit-banging I/O driver to have been compiled.

**I2CNOTES.DOC**

I2C documentation in Word format.

**DEVICES\8574DRV.FTH**

Driver for an 8574.

**DEVICES\8583DRV.FTH**

Driver for an 8583.

**DRIVERS\I2CVFXDRV.FTH**

Bit banging parallel port driver for VFX Forth for Windows.

## 26.4 SPI subdirectory

**SPINOTES.DOC**

SPI documentation in Word format.

**SPILOAD.BLD**

Build file that pulls in other SPI files.

**PPDRV.FTH**

PC printer port access for VFX Forth for Windows.

**SPIVFXDRV.FTH**

SPI primitives for VFX Forth for Windows. Requires PPDRV.FTH.

**SPIBASE.FTH**

SPI byte read and write primitives. A lower level driver is required.

**25LCDRV.FTH**

Driver for a Microchip 25LC series SPI EEPROM.