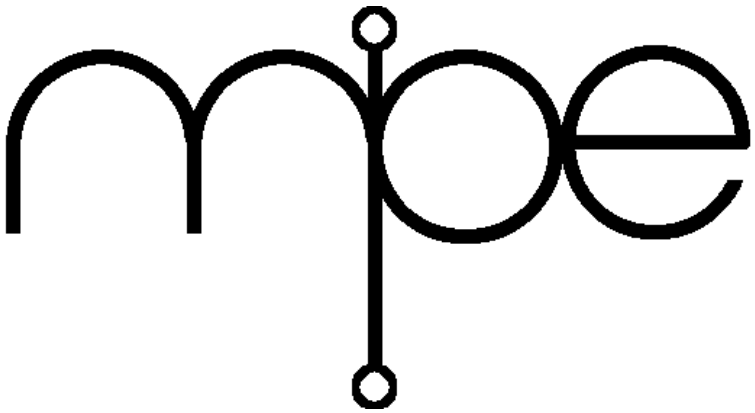
MSP430 Lite Target Code

v7.3

Microprocessor Engineering Limited



Published by Microprocessor Engineering

MSP430 Lite Target Code v7.3
User manual
Manual revision 7.3
10 November 2014

Software
Software version 7.3

For technical support
Please contact your supplier

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

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

Table of Contents

1 Lite version licence terms

1.1 Compiler

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

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

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

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

1.2 Distribution of application programs

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

1.3 Warranties and support

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

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

2 Introduction

This manual documents the MPE Forth kernel for the MSP430 Launchpad with an MSP430G2553 processor.

2.1 Before you start

The MPE Forth kernel uses the hardware UART for serial communications. This is run through the USB facilities on the Launchpad. Make sure that the five links of J3 and the two links of J5 on the Launchpad are set as below.

Figure 2.1: Launchpad links

Check that the TI DLLs, *MSP430.dll* and *HIL.dll* are installed where the MPE cross-compiler can find them. Installation is dicussed in chapter 4, "JTAG and and the MSP430" of the MSP40 cross-compiler manual, *Docs/MSP430man.pdf*. If the DLLs are incorrectly installed, you will not be able to program the Launchpad with the Forth kernel.

2.2 Working with AIDE

It seems to be essential that every language has an IDE. If you do not like IDEs, you can skip this section. AIDE has three parts:

- Editor - ForthEd2 is a trivial editor just so that AIDE has one. There are many far better programming editors. You can get AIDE to use a different one using the *IDE -> Configure/Locate* menu entry. See the AIDE manual for more details.
- Tool Capture Display - the cross compiler window in most cases. Note that this is software running on the host PC. For a standalone Forth target there is no connection to the target, except for Flash programming and all target commands must be entered in a terminal emulator, usually AIDE's PowerTerm. For an Umbilical Forth target, you can enter target commands and the cross compiler will try to execute them on the target.
- PowerTerm - a terminal emulator used with standalone Forth systems. It is specially adapted for debugging complex targets and features up to eight individually controllable cursor-addressable displays - ideal for debugging multi-tasking targets.

2.3 Producing the kernel

The Forth kernel is built using a cross compiler running in the AIDE environment. AIDE contains three primary windows: compiler, terminal emulator and text editor. The compilation results can be seen in the compiler window. During compilation, the compilation results can be seen in the compiler/tool window.

To compile the standalone Forth kernel, find the "liteLP2553sa" button in the main toolbar and click it to compile from the control file *liteLP2553sa.ctl*. The compiler will ask you whether you want to program the Launchpad with the new kernel. Answer 'Y' to the download and erase prompts.

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

1. Go to Windows Control Panel -> System -> Device Manager -> Ports. One of the entries will be something like: **MSP430 Application UART (COM10)**. COM10 will be the UART you will use.
2. Go to AIDE's PowerTerm configuration dialog. Select the drop down list in the COM Port# group. It will list all the available COM ports by COM number and name. The one you want includes: **COMxx MSP430 Application UART**.

Now select the required port in the PowerTerm configuration dialog and ensure that it is set to 9600 baud, 8 data bits, 1 stop bit, no parity and file server enabled. Save the configuration and press OK. Start the PowerTerm connection and you should be talking to the kernel.

2.4 About the standalone kernel

The control file *liteLP2553sa.ctl* specifies the use of the target Flash as follows:

```
Kernel: $D000..FFE0, then vectors to $FFFF
App:    $C000..CFFF
```

The App space is where the target kernel compiles code.

The Forth kernel you have made and installed is a cut-down version of the full MPE Forth kernel. It has extensions so that code is compiled directly into the MSP430 Flash. When you want to add code, you can compile it directly on the Forth kernel by using the phrase

```
include <filename>
```

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

To start over just type `EMPTY`, the application region of the Flash will be cleaned, and the CPU is rebooted. To reuse your compiled code at the next reboot or power up, use the word `COMMIT`. If you just want to preserve the code, use:

```
0 COMMIT
```

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

```
' APP COMMIT
```

2.5 About Umbilical Forth

The control file *liteLP2553uf.ctl* specifies the use of an Umbilical Forth system in which the cross compiler provides all the interactivity. Because the TI Launchpad uses a **very** slow USB connection, the interaction takes a very long time. It was designed for use with a 115200 baud serial line.

Umbilical Forth provides a minimal Forth system on the target for situations where every byte matters. You can put far more functionality onto a 16 kb Umbilical Forth system than on a 16 kb standalone Forth.

2.6 Gotchas

2.6.1 Flash problems

If you forget to use `COMMIT` and `EMPTY` appropriately, the MSP430 Flash may not be correctly erased and so application compilation may fail. Try to use `EMPTY` and if that fails, reinstall a new kernel.

The Flash programming DLLs are unreliable on many PCs. Just accept this as the cost of cheap hardware and software. Eventually, the Flash will be programmed, but you may need to recompile a few times to achieve this.

The Launchpad is built down to a price. The JTAG facilities are provided by a secondary CPU on the Launchpad and by Windows DLLs. In the past this combination has shown a few problems, especially if the Launchpad has been unexpectedly disconnected and reconnected.

If you come across such a problem, shut down all the software that could be using the MSP430 DLLs. Then restart the software. If problems persist, disconnect the Launchpad, reboot the PC, reconnect the Launchpad, and only then restart the software.

2.6.2 Serial lockup

Occasionally the connection to the USB UART refuses to open properly and AIDE's PowerTerm appears to lock up. DO NOT CLOSE THE CONNECTION. Instead, remove the USB cable and wait for PowerTerm to detect this and close the connection. Then reconnect the USB cable and attempt to open the UART again. You may have to do this a few times to establish a good connection.

3 MSP430G2553 start up

The file *MSP430\Hardware\Launchpad2553\start2553.fth* contains Forth start up code and initialisation tables for a Launchpad board with an MSP430G2553 CPU.

3.1 Magic addresses

Some factory calibration data is held in Info Flash in TLV (Tag, Length, Value) format. The most interesting of this are the DCO and ADC calibration data. For startup, we need the DCO calibration data

```
$10FF      CALBC1_1MHz
$10FE      CALDCO_1MHz
$10FD      CALBC1_8MHz
$10FC      CALDCO_8MHz
$10FB      CALBC1_12MHz
$10FA      CALDCO_12MHz
$10F9      CALBC1_16MHz
$10F8      CALDCO_16MHz
$10F7 08   Size (bytes) of value data
$10F6 01   Tag
```

Port 1 is used as follows:

```
P1.0       Green LED, high=on
P1.1       UART Rx
P1.2       UART Tx
P1.3       Button Switch input
P1.4       --
P1.5       --
P1.6       Red LED, high=on
P1.7       --
```

3.2 Start of Forth

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

`L: ECLD`
The entry point after reset. This code is for a TI Launchpad board run from the DCo at 8 MHz. It uses LEDs on P1.6 (red) and P1.0 (green) for debugging.

`XT2-speed [if]`
If the equate `XT2-speed` is set non-zero in the control file, the main clock is taken from the high frequency oscillator at the frequency specified by `XT2-speed`.

3.3 Default Interrupt vectors

All interrupt vectors are set to point to the `ECLD` entry point above. Later code may modify these settings.

```
ECLD reset_vec     !
ECLD nmi_vec       !
ECLD timer1_A0_vec !
ECLD timer1_A1_vec !
ECLD CompA_vec     !
ECLD WDT_vec       !
ECLD timer0_a0_vec !
ECLD timer0_a1_vec !
ECLD UCx0Rx_vec    !
ECLD UCx0Tx_vec    !
ECLD ADC10_vec     !
ECLD P2_vec        !
ECLD P1_vec        !
```

3.4 Reset values for user and system variables

The equate `SP-GUARD` in the control file defines how many guard cells are provided at the top of the data stack to give some protection against system crashes if code underflows the data stack. If `SP-GUARD` is undefined at this point, a default value of 0 is defined.

```
0 equ sp-guard  \ -- n
```

Default number of data stack guard cells.
The section of `USER` variable initialisation values is generated for standalone targets if the equate `UMBILICAL?` is udefined or set to zero.

4 MSP430 code definitions

The file *MSP430\Code430lite.fth* contains all the code definitions needed for a small standalone Forth kernel. Such a kernel is practical for the MSP430G2553 that has enough Flash and RAM. See *liteLP2553sa.ctl* for the control file.

If you are using Umbilical Forth, treat *MSP430\Code430lite.fth* as a source repository from which you can copy required words.

4.1 Register usage

On the MSP430 the following register usage is the default:

```
Forth         MSP430        Comments
-----------------------------------
IP            R0/PC         MSP430 PC
RSP           R1/SP         MSP430 return stack
--            R2/CG1/SR
--            R3/CG2
PSP           R4            data stack pointer
TOS           R5            cached top of data stack
UP            R6            USER area pointer
LP (locals)   R7            points to LOCALs on r. stack
scratch       R8..R13       with R5 forms working stack
codegen       R14           code generator temporary
codegen       R15           preserves SR in shuffle
-----------------------------------
```

The VFX code generator reserves R14 and R15 for internal operations. `CODE` definitions must use R5 as TOS with NOS pointed to by R4 as a descending stack. R8..R15 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.

4.2 Literal and flow of control

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

`proc docreate  \ -- a-addr`
The run time action of `CREATE`.

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

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

`CODE (LOOP)  \ -- ; absolute address follows inline`
The run time action of `LOOP` compiled on the target.

`CODE (+LOOP)  \ n --`

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

`code i \ -- n ; return DO ... LOOP index`
Return the current index of the inner-most DO..LOOP.

`code j \ -- n ; return outer DO ... LOOP index`
Return the current index of the outer DO..LOOP.

`code unloop \ -- ; discard DO ... LOOP parameters ; ANS 6.1.2380`
Remove the DO..LOOP control parameters from the return stack.

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

4.3 Flash operations

`: FlErase \ addr len --`
Erase the Flash sectors in the given range.

`: c!f \ b addr --`
Program any address, including Flash, with an 8 bit value.

`: !f \ w addr --`
Program any address, including Flash, with a 16 bit value.

4.4 Digits and strings

`code DIGIT \ char base -- 0 | n true`
If the ASCII value *char* can be treated as a digit for a number in the given *base* then return the digit and a TRUE flag, otherwise just return FALSE. For bases greater than 10, the letters A..Z are used, e.g. 0..9,A..F for hexadecimal.

`: /string \ addr len n -- addr+n len-n`
Modify a string address and length to remove the first N characters from the string.

`CODE CMOVE \ source dest len -- ; copy memory areas`
Copy U bytes of memory forwards from C-ADDR1 to C-ADDR2.

`CODE CMOVE> \ source dest len -- ; copy memory areas`
As CMOVE but working in the opposite direction, copying the last character in the string first.

`code fill \ addr len char --`
Fill *len* bytes of memory starting at *addr* with the byte specified as *char*.

`: ERASE \ addr len --`
Fill *len* bytes of memory starting at *addr* with zero.

`code s= \ addr1 addr2 count -- flag`
Compare two same-length strings/memory blocks, returning TRUE if they are identical.

`code SKIP \ c-addr u char -- 'c-addr 'u`
Modify the string description by skipping over leading occurrences of 'char'.

`code scan \ caddr u char -- caddr2 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.

`: count \ addr -- addr+1 len`
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.

```
: (")           \ -- addr ; in-line string follows caller, and skip 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.

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

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

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

4.5 Arithmetic

4.5.1 Basics

```
: 1+ 1+ ;       \ n -- n+1
```
Add one to top-of stack.

```
: 2+ 2+ ;       \ n -- n+2
```
Add two to top-of stack.

```
: 1- 1- ;       \ n -- n-1
```
Subtract one from top-of stack.

```
: 2- 2- ;       \ n -- n-2
```
Subtract two from top-of stack.

```
: 2* 2* ;       \ n1 -- n2
```
Signed multiply top of stack by 2.

```
: u2/ u2/ ;     \ n1 -- n2 ; unsigned
```
Unsigned divide top of stack by 2.

```
: 2/ 2/ ;       \ n1 -- n2 ; signed
```
Signed divide top of stack by 2.

```
: - - ;         \ n1 n2 -- n1-n2
```
Subtract two single precision integer numbers. N3|u3=n1|u1-n2|u2.

```
: + + ;         \ n1 n2 n1+n2
```
Add two single precision integer numbers.

```
: negate        \ n1 -- -n1
```
Negate a single precision integer number.

```
: abs           \ n1 --  |n1|
```
If n is negative, return its positive equivalent (absolute value).

```
: dnegate       \ d1 -- -d1
```
Negate a double number.

```
: dabs          \ d1 -- |d1|
```
If d is negative, return its positive equivalent (absolute value).

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

```
CODE D-         \ d1 d2 -- d1-d2
```

Subtract two double precision integers. D3=D1-D2.

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

`: D< \ d1 d2 -- t/f`
Return TRUE if the double number d1 is < the double number d2.

`: d> \ d1 d2 -- t/f`
Return TRUE if the double number d1 is > the double number d2.

`: d0= \ d -- t/f`
Returns true if d is 0.

`: d= \ d1 d2 -- t/f`
Return TRUE if the two double numbers are equal.

4.5.2 Multiplication

`code um* \ u1 u2 -- ud ; unsigned multiply`
Perform unsigned-multiply between two numbers and return double result.

`code * \ n1 n2 -- n1*n2 ; signed multiply, : * um* drop ;`
Standard signed multiply. N3 = n1 * n2.

`code m* \ n1 n2 -- d ; signed multiply`
Signed multiply yielding double result.

4.5.3 Division

`code um/mod \ u32 u16 -- urem uquot`
Perform unsigned division of double number UD by single number U and return remainder and quotient.

`code sm/rem \ d n -- 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.

`: /mod \ n1 n2 -- rem quot`
Signed division of N1 by N2 single-precision yielding remainder and quotient.

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

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

`: MU/MOD \ ud1 u2 -- u3 ud4`
Perform an unsigned divide of a double ud1 by a single u2, returning a single remainder u3 and a double quotient ud4.

4.6 Logic

`: AND AND ; \ n1 n2 -- n3`
Perform a logical AND between the top two stack items and retain the result in top of stack.

`: OR OR ; \ n1 n2 -- n3`
Perform a logical OR between the top two stack items and retain the result in top of stack.

`: XOR XOR ; \ n1 n2 -- n3`
Perform a logical XOR between the top two stack items and retain the result in top of stack.

```
: INVERT  INVERT ;      \ n1 -- n2
```
Perform a bitwise NOT on the top stack item and retain result.

4.7 Shifts

```
code lshift     \ x count -- x'
```
Left shift x by count bits.

```
code rshift     \ x count -- x'
```
Right shift x by count bits.

4.8 Return stack words

```
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 -- x
```
Copy the top item from the return stack to the data stack.

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

4.9 Comparisons

```
: =             \ n1 n2 -- flag
```
Return TRUE if the two topmost stack items are equal.

```
: <>            \ n1 n2 -- flag
```
Return TRUE if the two topmost stack items are different.

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

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

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

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

```
: U<            \ n1 n2 -- flag
```
An UNSIGNED version of <.

```
: U>            \ n1 n2 -- flag
```
An UNSIGNED version of >.

```
: <             \ n1 n2 -- t/f
```
Return TRUE if n1 is less than n2.

```
: >             \ n1 n2 -- t/f
```
Return TRUE if n1 is greater than n2.

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

```
: >=            \ n1 n2 -- t/f
```
Return TRUE if n1 is greater than or equal to n2.

4.10 Stack primitives

```
: OVER over ;   \ n1 n2 -- n1 n2 n1
```
Copy NOS to a new top-of-stack item.

```
: 2OVER         \ n1 n2 n3 n4 -- n1 n2 n3 n4 n1 n2
```
Similar to `OVER` but works with cell-pairs rather than cell items.

```
: DROP drop ;   \ n1 --
```
Lose the top data stack item and promote NOS to TOS.

```
: 2DROP 2drop ; \ n1 n2 --
```
Discard the top two data stack items.

```
: SWAP swap ;   \ n1 n2 -- n2 n1
```
Exchange the top two data stack items.

```
: 2SWAP         \ n1 n2 n3 n4 -- n3 n4 n1 n2
```
Exchange the top two cell-pairs on the data stack.

```
: DUP dup ;     \ n1 -- n1 n1
```
`DUP`licate the top stack item.

```
: 2DUP 2dup ;   \ n1 n2 -- n1 n2 n1 n2
```
`DUP`licate the top cell-pair on the data stack.

```
: ?dup          \ n1 -- n1 [n1]
```
`DUP`licate the top stack item only if it non-zero.

```
: nip nip ;     \ n1 n2 -- n2
```
Dispose of the second item on the data stack.

```
: tuck          \ n1 n2 -- n2 n1 n2
```
Insert a copy of the top data stack item underneath the current second item.

```
: pick          \ nn..n0 n -- nn..n0 nn
```
Get a copy of the Nth data stack item and place on top of stack. `0 PICK` is equivalent to `DUP`.

```
: ROT           \ n1 n2 n3 -- n2 n3 n1
```
ROTate the positions of the top three stack items such that the current top of stack becomes the second item. See also `ROLL`.

```
: -rot          \ n1 n2 n3 -- n3 n1 n2
```
The inverse of `ROT`.

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

```
: @ @ ;         \ addr -- n
```
Fetch and return the CELL at memory ADDR.

```
: 2@ 2@ ;       \ addr -- d
```
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.

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

```
: !             \ n addr --
```
Store the CELL quantity N at memory ADDR.

```
: 2!            \ d addr --
```

Store the two CELLS x1 and x2 at memory ADDR. X2 is stored at ADDR and X1 is stored at ADDR+CELL.

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

`code noop  \ -- ; dummy`
A NOOP, null instruction.)

`: within  \ n1|u1 n2|u2 n3|u3 -- flag ; ANS 6.2.2440`
The ANS version of `WITHIN?`. This word uses unsigned arithmetic, so that signed compares are treated as existing on a number circle.

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

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

`: bounds  \ addr len -- addr+len addr`
Convert an address and length to adress+length and address as required for `DO..LOOP` to use `I` as the current address.

`: name>  \ nfa -- cfa ; convert name address to CFA`
Move a pointer from an NFA to the CFA or "XT" in ANS parlance.

`: >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.

4.11 Portability words

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

`cell constant cell  \ -- 2`
Return the size in address units of one CELL.

`: cells  \ n -- n*2`
Return the number of bytes required to hold the given number of 16 bit cells.

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

`: >body  \ xt -- pfa ; step from code field to parameter field`
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.

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

4.12 Defining words

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

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

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

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

`: VARIABLE  \ "<spaces>name" -- ; Exec: -- a-addr`
Create a new variable called `name`. When `Name` is executed the address of the RAM is returned for use with `@` and `!`. The RAM is not initialised.

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

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

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

or (when compiled)

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

4.13 Miscellaneous

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

`code reboot  \ --`
Starts the watchdog and waits until the CPU reboots.

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

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

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

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

`: or!  \ mask addr --`

Set *mask* bits in the word/cell at *addr*.

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

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

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

5 High level kernel - kernel72lite.fth.

The Forth kernel words documented here are entirely written in high-level Forth. The kernel is reduced in size to match available code size in small devices such as the MSP430G2553 in the TI Launchpad.

5.1 User variables

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

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

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

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

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

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

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

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

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

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

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

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

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

5.2 System data

5.2.1 Constants

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

5.2.2 System variables and data

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

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

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

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

`variable LAST \ -- addr`
Points to name field of last definition

5.3 Vectored I/O handling

5.3.1 Introduction

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

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

5.3.2 Building a vector table

The example below is taken from an ARM implementation.

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

Console1 opvec !  Console1 ipvec !
```

5.3.3 Generic I/O words

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

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

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

`: TYPE \ caddr len -- ; display string`

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

```
: CR            \ -- ; display new line
```

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

```
: SPACE         \ --
```

Output a blank space (ASCII 32) character.

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

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

5.4 Laying data in memory

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

```
: HERE          \ -- addr
```

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

```
: ORG           \ addr --
```

Set the current dictionary pointer.

```
: ALLOT         \ n --
```

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

```
: RHERE         \ -- addr
```

Return the current RAM dictionary pointer.

```
: RALLOT         \ n --
```

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

```
: ROM           \ --
```

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

```
: RAM           \ --
```

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

```
  RAM ... ROM
```

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

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

```
: ALIGN         \ --
```

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

```
: ,             \ x --
```

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

```
: C,            \ char --
```

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

5.5 Dictionary management

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

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

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

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

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

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

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

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

`: CREATE \ --`
Create a new definition in the dictionary. When the new definition is executed it will return the address of the definition's data area. As compilation is into Flash, `CREATE` cannot be used with `DOES>` and `<BUILDS ... DOES> ...` must be used instead.

`: <BUILDS \ --`
Always used in the form:

```
: defword <BUILDS ... DOES> ... ;
```

When `defword` is executed a new definition is created with the data defined between `<BUILDS` and `DOES>` and the action defined between `DOES>` and `;`. You must use `<BUILDS` and `DOES>` together, otherwise there will be a crash. Treat `<BUILDS` as a special case of `CREATE` for use with `DOES>` and compilation into Flash.

5.6 String compilation

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

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

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

The run time action of ABORT". INTERNAL.

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

The run-time action of ."". INTERNAL.

5.7 ANS words CATCH and THROW

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

CATCH and THROW provide a reliable mechanism for handling exceptions, without having to propagate exception flags through multiple levels of word nesting. It is similar in spirit to the "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.

5.7.1 Example use

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

Typical use:

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

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

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

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

5.7.2 Gotchas

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

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

5.7.3 User words

`: CATCH  \ i*x xt -- j*x 0|i*x n`

Execute the code at XT with an exception frame protecting it. `CATCH` returns a 0 if no error has occurred, otherwise it returns the throw-code passed to the last `THROW`.

`: THROW  \ k*x n -- k*x|i*x n`

Throw a non-zero exception code n back to the last `CATCH` call. If n is 0, no action is taken except to `DROP` n.

`: ?throw  \ flag throw-code -- ; SFP017`

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

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

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

5.8 Formatted and unformatted i/o

5.8.1 Setting number bases

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

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

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

5.8.2 Numeric output

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

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

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

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

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

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

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

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

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

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

5.8.3 Numeric input

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

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

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

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

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

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

5.9 String input and output

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

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

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

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

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

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

5.10 Source input control

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

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

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

5.11 Text scanning

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

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

buffers are used PARSE-WORD is more reliable than WORD for text scanning in multi-threaded applications. INTERNAL.

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

5.12 Miscellaneous

```
: 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.

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

5.13 Wordlist control

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

5.14 Control structures

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

5.15 Target interpreter and compiler

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

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

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

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

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

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

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

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

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

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

```
: 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. `[COMPILE]` is mostly superceded by `POSTPONE`.

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

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

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

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

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

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

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

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

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

```
: .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 is on. Note that any task that uses QUIT must initialise 'TIB, BASE, IPVEC, and OPVEC.

5.16 Startup code

5.16.1 The COLD sequence

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

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

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

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

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

5.17 Kernel error codes

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

6 Time Delays

The code in *Delays.fth* allows you to handle time delays specified in milliseconds.

```
: ticks          ( -- n )  <ticks> @  ;
```
Return current clock value in milliseconds. This value can treated as a 16 bit unsigned value that wraps when it overflows.

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

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

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

7 Debug tools

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

`: dump  \ addr len --`
Display the given block of memory.

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

8 Compile source code from AIDE

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

`: 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 `INCLUDE`d.

`: $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.

9 Minimal Umbilical code definitions

The file *Min430.fth* contains the minimum code definitions required to support Umbilical Forth. If additional definitions are required, they may be copied to a new file from *Code430lite.fth* or *kernel72lite.fth*.

Note that many of the words documented here are only included by the early non-optimising compiler. These words will be removed from this file in a future release. If needed, they may then be copied from another file.

9.1 Logical and relational operators

`: min \ n1 n2 -- min(n1,n2)`
Given two data stack items preserve only the smaller.

`: max \ n1 n2 -- max(n1,n2)`
Given two data stack items preserve only the largerr.

`code within? \ x a b -- t/f ; true if a<=x<=b`
Return TRUE if N1 is within the range N2..N3. This word uses signed arithmetic.

`code within \ n1|u1 n2|u2 n3|u3 -- flag ; ANS 6.2.2440`
The ANS version of `WITHIN?`. Return TRUE if N1 is within the range N2..N3-1. This word uses unsigned arithmetic, so that signed compares are treated as existing on a number circle.

9.2 Control flow

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

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

`CODE (LOOP) \ -- ; absolute address follows inline`
The run time action of `LOOP` compiled on the target.

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

`code i \ -- n ; return DO ... LOOP index`
Return the current `DO ... LOOP` index.

`code j \ -- n ; return DO ... LOOP index`
Return the outer `DO ... LOOP` index.

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

`Code Noop \ -- ; used by multi-tasker`
A NOOP, null instruction.)

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

`code um* \ u1 u2 -- ud ; unsigned multiply`
Perform unsigned-multiply between two numbers and return double result.

```
code *          \ n1 n2 -- n1*n2 ; signed multiply, : * um* drop  ;
```
Standard signed multiply. N3 = n1 * n2.

```
code m*         \ n1 n2 -- d ; signed multiply
```
Signed multiply yielding double result.

```
code um/mod     \ u32 u16 -- urem uquot
```
Perform unsigned division of double number UD by single number U and return remainder and quotient.

```
code sm/rem      \ d n -- 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 fm/mod     \ d n -- frem fquot ; 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.

```
: /mod          \ n1 n2 -- rem quot
```
Signed division of N1 by N2 single-precision yielding remainder and quotient.

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

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

```
: MU/MOD        \ ud u -- urem udquot
```
Perform an unsigned divide of a double by a single, returning a single remainder and a double quotient.

```
: */MOD         \ n1 n2 n3 -- rem quot
```
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 -- quot
```
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 n -- quot
```
Signed divide of a double by a single integer.

```
CODE negate     \ n1 -- -n1
```
Negate a single number.

```
: ?negate       \ n1 t/f -- n1/-n1
```
If flag is negative, then negate n1.

```
: abs           \ n1 --  |n1|
```
If n is negative, return its positive equivalent (absolute value).

```
code dnegate    \ d1 -- -d1
```
Negate a double number.

```
: ?dnegate      \ d1 t/f -- d1/-d1
```
If flag is negative, then negate d1.

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

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

9.3 Stack manipulation

Many of the standard Forth stack manipulation words are just code generators in the cross compiler and have no target versions.

`: ROLL  \ n1 n2 .. nk n -- wierd`
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`. N.B. Very slow.
The standard Forth comparison words words are just code generators in the cross compiler and have no target versions.

Some of the standard Forth memory manipulation words are just code generators in the cross compiler and have no target versions.

`code c@  \ addr -- b`
Fetch and 0 extend the character at memory ADDR and return.

`code @  \ addr -- n`
Fetch and return the CELL at memory ADDR.

`code 2@  \ addr -- d`
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 C!  \ b addr --`
Store the character CHAR at memory C-ADDR.

`CODE !  \ n addr --`
Store the CELL quantity N at memory ADDR.

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

9.4 String operators

`code count  \ addr -- addr+1 len`
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 CMOVE  \ source dest len -- ; copy memory areas`
Copy U bytes of memory forwards from C-ADDR1 to C-ADDR2.

`CODE CMOVE>  \ source dest len -- ; copy memory areas`
As `CMOVE` but working in the opposite direction, copying the last character in the string first.

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

`: (")  \ -- addr ; in-line string follows caller, and skip it`
Return the address of a counted string that is inline after the CALLING word, and adjust the CALLING word's return address to step over the inline string. See the definition of (.") for an example.

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

The runtime action compiled by C".

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

9.5 Umbilical versions of defining words

```
here is-action-of constant
```
The runtime action for a CONSTANT.

```
here is-action-of variable
```
The runtime action for a VARIABLE.

```
here is-action-of user
```
The runtime action of a USER variable.

```
: u#            \ "<uservar>" -- offset ; u# <uservar>
```
Return the offset of a USER variable in the user area.

```
here is-action-of value \ -- n ; default returns the value
```
The runtime action of a VALUE.

```
: CRASH         \ --  ; used as default action of DEFERred word
```
The default action of a DEFERed word. A NOOP.

```
here is-action-of DEFER \ Comp: "<spaces>name" -- ; Run: i*x -- j*x
```
The runtime action of a DEFERred word.

9.6 Interrupt handling

The interrupt handling words are just code generators in the cross compiler and have no target versions.

10 MSP430 library

The library file LIB430.FTH can be used between LIBRARIES and END-LIBS to resolve outstanding forward references. See the Forth 6 manual for more details of the library mechanism.

```
libraries       \ resolve required forward references
  include %CpuDir%\lib430
\  include %CommonDir%\library \ for standalone apps
  include %CommonDir%\uflib    \ for Umbilical apps
end-libs
```

`code save-int   \ -- n ; save interrupt status`
Geturn interrupt status and then disable interrupts. Use [I and I] for new code.

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

`: init-iow      \ addr --`
Copy the contents of the I/O set up table to an I/O device. The data is written as 16 bit words. See also INIT-IOB. Each element of the table is of the form addr (cell) followed by data (cell). The table is terminated by an address of 0. A table of a single 0 address performs no action.

`: init-iob      \ addr --`
Copy the contents of the I/O set up table to an I/O device. The data is written as 8 bit bytes. See also INIT-IOW. Each element of the table is of the form addr (cell) followed by data (cell). The table is terminated by an address of 0. A table of a single 0 address performs no action.

11 USCI serial driver

The file *MSP430Lite\Drivers\serUSCIp.fth* contains the code for a polled serial driver.

11.1 Baud rate calculation

`: genUSCIlf \ baud clock -- brx brsx`
Generate the baud rate values for low frequency baud rate settings. The word can only be executed while interpreting in the cross compiler.

`: genUSCIos \ baud clock -- brx brfx`
Generate the baud rate values for oversampling baud rate settings. The word can only be executed while interpreting in the cross compiler.

11.2 UART0

`: key?0 \ -- flag`
Return true if UART0 received a character.

`: key0 \ -- char`
Wait for character from UART0 and return it.

`: emit0 \ char --`
Send a character through UART0.

`: type0 \ c-addr len --`
Send a string through UART0.

`: cr0 \ --`
Perform CR on UART0.

`console0-speed system-speed genUSCIlf equ BRS0 equ BRX0`
Generate the baud rate values for UART0.

`: init-ser \ --`
Configure UART0. This word can be hardware dependent. The version here assumes USCI A0 on P1.1 and P1.2.

`create Console0 \ -- addr ; OUT managed by upper driver`
The device vector for UART0.

```
  Console0 constant Console
```
Defines UART0 as the default system console.

11.3 UART1

`: key?1 \ -- flag`
Return true if UART1 received a character.

`: key1 \ -- char`
Wait for character from UART1 and return it.

`: emit1 \ char --`
Send a character through UART1.

`: type1 \ c-addr len --`
Send a string through UART1.

`: cr1 \ --`

Perform CR on UART1.

`console1-speed system-speed genUSCIlf  equ BRS1  equ BRX1`
Generate the baud rate values for UART1.

`: init-ser1     \ --`
Configure UART1. This word can be hardware dependent. The version here assumes USCI A1. No port selection is made. This code is only compiled if another version has not been defined.

`create Console1 \ -- addr ; OUT managed by upper driver`
The device vector for UART1.

```
  Console1 constant Console
```
Defines UART1 as the default system console.

11.4 Initialisation

`: init-ser      \ --`
Initialise the serial channels.

12 Ticker using watchdog timer

The ticker interrupt is provided in *LedTickLP2553.fth*.

```
: ticks         ( -- n )  <ticks> @  ;
```
Return current clock value in milliseconds. This can be treated as a 16 bit unsigned value that will wrap when it overflows.

```
  equ TickHz    \ -- hz
```
Ticker speed in Herz

```
1000 TickHz / equ Tick-Ms       \ -- ms
```
Milliseconds per tick.

```
variable LedActive      \ -- addr
```
Set true for LEDs to flash on the timer. Set this to false (zero) when you want to use the LEDs yourself.

```
Proc WDT-isr
```
Assembler coded interrupt service routine.

```
  add .w    # Tick-ms  & <ticks>
  add .w    # Tick-ms  & LedTimer
  cmp       # #1000 & LedTimer          \ 1 second timeout
  eq, if,
    mov .w    # 0 & LedTimer            \ reset timer
    cmp       # 0  & LedActive          \ if enabled
    ne, if,
      xor .b    # $41 & P1OUT           \ toggle LEDs
    endif,
  endif,
  reti
end-code
WDT-isr WDT_vec !


: Start-Clock   \ --
```
Start the ticker interrupt.

```
: Stop-Clock    \ --
```
Stop the ticker interrupt

```
: green-on      \ -- ; P1.6
```
Turn green LED on.

```
: green-off     \ -- ; P1.6
```
Turn green LED off.

```
: red-on        \ -- ; P1.0
```
Turn red LED on.

```
: red-off       \ -- ; P1.0
```
Turn red LED off.

13 Device drivers

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

13.1 Basic port usage

See the file *Drivers\gpio2553.fth*.

The user words are as follows:

- `HI` and `LO` configure the port bit for output.
- `BIT?` does not configure the port bit for input.
- `SETIN` configures the port bit for input.

```
P1IN CONSTANT PORT1     \ -- addr
```
Base of Port 1.

```
P2IN CONSTANT PORT2     \ -- addr
```
Base of Port 2

```
: HI            \ bit port --
```
Set bit high.

```
: LO            \ bit port --
```
Set bit low.

```
: SETIN         \ bit port --
```
Set pin to input. Deselect other functions.

```
: BIT?          \ bit port --- f
```
Return the state of the bit in the port.

13.2 Port counting using interrupts

See the file *Drivers\gpio2553.fth*.

The notation is:

```
  bit port startcount
```

The code assumes that interrupts are enabled elsewhere.

```
VARIABLE P1COUNT        \ -- addr
```
Holds the count for Port 1.

```
VARIABLE P2COUNT        \ -- addr
```
Holds the count for Port 2.

```
: STARTCOUNT    ( bit port -- )
```
Start counting transitions for the given bit and port.

```
: STOPCOUNT     ( port -- )
```
Stop counting transitions on the given port.

13.3 Simple ADC driver

See the file *Drivers\adc2553.fth*. The driver is not clever. It does all the setting up for each conversion - there is no separate set up word. It Leaves I/O pins in analogue mode until the next conversion. Be careful not to do a conversion on a digital pin. The supply voltage is used as the reference. 0 corresponds to 0V, 1023 corresponds to VCC.

```
: ATOD          \ channel# -- value
```
DO a conversion on the given ADC channel (0..7) and return the value.

13.4 PWM

See the file *Drivers\pwm2553.fth*. The code sets up for three possible channels of 10 bit PWM for 20 pin device. The PWM frequency is about 7.5KHz.

Use `P1.6PWM`, `P2.1PWM` or `P2.4PWM` to initialise the relevant port and timer. P1.6 uses Timer0, the other two pins use Timer1. It is best to use P2.1 and P2.4 first to keep the other timer free. Once initialised, use `P1.6DUTY`, `P2.1DUTY` or `P2.4DUTY` to set the duty cycle.

```
: P1.6DUTY      \ value --
```
Set the duty cycle on P1.6, $0..3FF.

```
: P1.6PWM       \ duty --
```
Start PWM on pin 1.6, where *duty* is $0..3FF

```
: P2.1PWM       \ duty --
```
Start PWM on pin 2.1, where *duty* is $0..3FF

```
: P2.4PWM       \ duty --
```
Start PWM on pin 2.4, where *duty* is $0..3FF

Index

@

[

]

\

0

1

2

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X