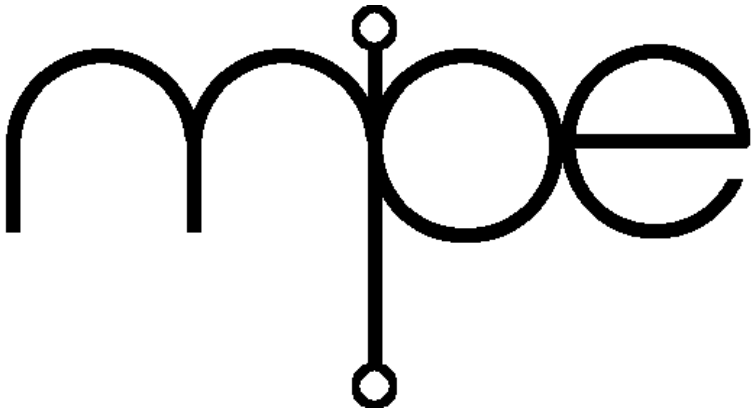
Forth 7 Cross Compiler

with VFX code generation

Microprocessor Engineering Limited



Published by Microprocessor Engineering

MPE VFX Forth Cross Compiler
User manual
Manual revision 7.5
1 October 2016

Software
Software version 7.5

For technical support
Please contact your supplier

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

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

Table of Contents

1 Licence terms

1.1 Distribution of application programs

Providing that the end user has no access to the underlying Forth and its text interpreter except for engineering and maintenance access only, applications compiled with the Forth 7 cross-compiler may be distributed without royalty. An acknowledgement will be gratefully appreciated. No part of the cross-compiler or the target source code may be further distributed without written permission from MicroProcessor Engineering.

If you need to ship applications with an open Forth system, or wish to check what constitutes engineering and maintenance access, please contact MPE. An OEM version of ROM PowerForth is available for distribution with your products, and includes documentation on disc.

1.2 Evaluation and Lite compilers

These terms apply to the compilers supplied free of charge as the evaluation or Lite editions.

Commercial use of evaluation or Lite compilers is not permitted. If you sell an application written with VFX Forth, that is commercial use. If you are paid to write software with VFX Forth, that is commercial use. If you are a teacher and want to use VFX Forth in a class, that is commercial use, so contact us an we will give you written permission.

If you think that you are a special case, please contact us and we will consider your case.

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 fax or email 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 also let us know the serial number of your system and its version number. We will then send you an update when we have fixed the problem. The level of technical support that we can offer may depend on the Support Policy bought with the product.

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

Make as many copies as you need for backup and security. The issue discs or CD are not copy protected. The code is copyrighted material and only ONE copy of it should be use at any one time. Contact MPE or your vendor for details of multiple copy terms and site licensing.

As this copy is sold direct and through dealers and purchasing departments, we cannot keep track of all our users. Please contact tech-support@mpeforth.com to register your compiler. We need the compiler type and serial number. This way we will be able to keep you informed of updates and new extensions, as they become available. If you need technical support from us we will need these details in order to respond to you. You will find the serial number of the system on the original issue discs or as part of the download instructions.

2 Installing the system

The installer helps you through the installation process and will make sure you have all the files you need.

2.1 System requirements

To install and use the development system you need:

- Mac OS X 10.6 or above.
- x86 PC with Linux
- Windows XP/Vista/Windows7 with 512 Mb or more of RAM. The Windows version will probably work on Windows 2000, but if it doesn't we aren't likely to fix it unless you have a compelling reason that convinces us to fix it.
- At least 20-60 Mbytes of free disc space, depending on the amount of CPU specific documentation provided.

2.2 Installation and configuration

This section covers general installation of the cross compiler. The target-specific manual will detail how to install tools such as JTAG programmers for ARM/Cortex or MSP430 CPUs.

2.2.1 Windows

Installation

The software is usually supplied as a download. The main file is an installer. Run the installer. The installer will prompt you for all the information it needs, offering defaults. The installer will also create a new start menu program group for you that contains shortcuts to tools and help files.

Some versions of the installers will ask you for a licence key. When entering the 12 digit key, remove all punctuation and separators. Just enter 12 decimal digits.

Configuration

Everything you need can be accessed through the Aide shell. Many people find it useful to put a shortcut to *<xCPU>\AIDE\AIDE.EXE* on the desktop. Configuration of Aide is discussed in the Aide manual in the *Docs* folder.

Aide will have a couple of projects already installed in Aide's main toolbar. Click the button to run them.

Port access under Windows XP onwards

Direct access to I/O ports is required for SPI parallel port drivers, and other target access drivers. If you are using Linux or Windows XP or later, direct port I/O requires a driver that permits this access, otherwise you will trigger an exception with an error message such as "Cannot run privileged instruction".

The directory *COMPILER\XTRA* contains *NTPORT.EXE*, which permits an application to use

any I/O port. Note that this completely bypasses the normal Windows NT I/O port protection mechanism. If you want something more secure there are several utilities available from the Internet.

To install NTPORT perform the following procedure. Our thanks go to Graham Gollings of LMS bv for this description of the process.

Run the NTPORT utility located in your *COMPILER\XTRA* folder. This puts various files in the right places, but does not install the driver itself. Loading a driver is performed by the LOADDRV utility.

Run *LOADDRV.EXE* from your *COMPILER\XTRA* folder. In the window "Full pathname of driver" point to *GIVEIO.SYS*.

Tick on INSTALL (It should say operation was successful)

Tick on RUN (It should say operation was successful)

Now run TSTIO.exe (and a tune should play). At this point *GIVEIO.SYS* is running, but the next time the system is started from cold it will be loaded at system start up but will not run, as it is configured as manual. We need it to be loaded and running from cold start. In order to set this up, run *REGEDIT*. Look to path:

`HKEY_LOCAL_MACHINE | SYSTEM | CURRENT CONTROL SET | SERVICES | GIVEIO`

Right click on GIVEIO, and change the START REG_DWORD from 3 (manual) to 2 (automatic)

To test the installation, run the compiler directly from the *COMPILER* directory with no command line. In the console, type the following incantation:

```
ALSO C-C     \ add C-C vocabulary to search order
PIO-INIT     \ initialise driver access
PIO-TEST     \ should play a tune
PREVIOUS     \ remove C-C from search order
BYE          \ exit from compiler
```

When using the compiler, you must add the directive `NT-ACCESS-PORTS` to your control file before any direct access to hardware is required. A good place to add it is after the `CROSS-COMPILE` directive in the section in which the compiler is configured.

2.2.2 Linux and Mac OS X

Installation

The compiler is supplied as a zipped tarball. Unzip this to somewhere sensible for your machine, e.g. a directory in your home directory.

```
$ tar -xvvzf xCPU.tar.gz
```

Inside the compiler's root directory will be a shell script called *InstallMeLin.sh* or *InstallMeOsx.sh*. Switch to the compiler's root directory and run the script:

```
$ cd <xCPUroot>
$ ./InstallMeLin.sh
```

This script copies the required executables and shared libraries, by default to */usr/bin* and */usr/lib*. If your distribution requires different destinations, edit the script before running it. If you use Ubuntu or are not running with administrator privileges you will have to use *sudo* or another dark art.

Check the installation by runnng the compiler from any directory:

```
$ x<Cpu><ver>
```

For example the Stamp and Developer versions of the ARM cross compiler are called *xArmStamp* and *xArmDev* respectively.

We apologise for not generating proper installers for Linux, but it's a huge amount of work to get it right for all distributions. At present, we would have to generate five packages (deb32, deb64, rpm32, rpm64 and tarball) for each version of each compiler. Until a sane and affordable packaging solution is available for Linux, tarballs are what we can do for Linux cross compilers.

Configuration

Cross compilers are not supplied with an editor. If you want to set one, use:

```
editor-is <editor>
```

e.g.

```
editor-is emacs
editor-is /bin/vi
```

After this, you can launch the editor from the compiler with:

```
edit <file>
```

Do not be surprised if launching a GUI-based editor generates error messages on the Forth console - this appears to be normal Linux behaviour. *Kate* is a particular offender.

`SetLocate` tells the host VFX Forth how your editor can be called to go a particular file and line. Use in the form:

```
SetLocate  <rest of line>
```

where the text after `SetLocate` is used to define how parameters are passed to the editor, e.g. for Emacs, use:

```
SetLocate +%l% "%f%"
```

```
EMACS        +%l% "%f%"
             --no-wait +%l% "%f%"
Kate        --use --line %l% "%f%" &
```

Thanks to Charles Curley for the additional EMACs information. See

http://www.charlescurley.com. He also notes that you should add the following to your .emacs file:

```
(if (or (string-equal system-type "gnu/linux")
        (string-equal system-type "cygwin"))
    (server-start)
         (message "emacsserver started."))
```

It is essential to place the quote marks around the %f% macro if your source paths include spaces.

Once set up, you can view the source of a word with:

```
locate <name>
```

If the editor is not set up `locate <name>` will tell you where it is.

The configuration information is preserved between sessions in a configuration file, by default *~/.VfxForth.ini*.

First run

To compile a project, find a control file (with a *.ctl* extension). Then write a simple script to compile it, or run the relevant command from a shell. The following example comes from the ARM compiler. It compiles code for the MPE USB ARM Stamp hardware in the directory *<xArmxxx>/ARM/Hardware/LPC210x/*. The script file is called *xusbstamp.sh*.

```
#! /bin/bash
# Linux shell script to compile the USB Stamp
bindir=/usr/bin

if [ -e $bindir/xArmStamp ]; then
  echo "Using STAMP compiler"
  xArmStamp include USBstamp.ctl
  exit
fi

if [ -e $bindir/xArmDev ]; then
  echo "Using DEV compiler"
  xArmDev include USBstamp.ctl
  exit
fi

echo "Compiler not found"
exit 1
```

Don't forget to make the script file executable!

```
$ chmod +x xusbstamp.sh
```

If your CPU has an Open Source or freely distributable Flash programming tool, it will be available in the *Tools* directory.

Common compilation problems

The cross compilers come from a code tree developed under Windows. We simply do not have the resources to retest every project under Linux and OS X. Common problems that you will find are usually to do with case-sensitity of file names and the use of '\' in path names rather than '/'. We are working to fix these problems.

1. Windows file systems are case insensitive. Consequently, when running a project that was developed on Windows, file names may be invalid under Linux that were acceptable on Windows.
2. Windows uses the '\' or the '/' character as the directory separator. Linux only uses '/'.
3. Some auxiliary files (not Forth source code) may have been processed by Windows tools and inadvertently acquired CR/LF line endings rather than the Unix LF line ending. Windows tools are not good about preserving line endings and Linux tools are not good at accepting line endings other than LF. There are times when we just don't notice which box we're developing on!
4. Files edited on Windows may end up with mangled permissions.

Direct port access

The following notes are for developers working under x86-32 versions of Linux. Under normal use, direct access to I/O ports is forbidden. However, if you are running with root privilege, you can use the glibc functions **ioperm()** and **iopl()** to enable and disable port access.

```
code pc@        \ port -- b ; read port
```
Read a byte from the hardware control port supplied.

```
code pc!        \ b port -- ; write port
```
Write the supplied byte to the selected hardware control port.

```
code pw@        \ port -- w ; read port
```
Read a 16 bit word from the hardware control port supplied.

```
code pw!        \ w port -- ; write port
```
Write the supplied 16 bit word to the selected hardware control port.

```
code pl@        \ port -- x ; read port
```
Read 32 bits from the hardware control port supplied.

```
code pl!        \ x port -- ; write port
```
Write the supplied 32 bits to the selected hardware control port.

```
: +Ports        \ port #ports -- ior
```
Enable access to a range of ports starting at *port*. Return 0 on success. Port numbers must be in the range 0..$3FF. You must have root permissions.

```
: -Ports        \ port #ports -- ior
```
Disable access to a range of ports starting at *port*. Return 0 on success. Port numbers must be in the range 0..$3FF. You must have root permissions.

```
: PlayNote      \ hertz ms --
```
Play a note on the internal PC speaker. Ports $42, $43 and $61 must be enabled first.

`: pio-test \ --`
A test routine for hardware access. Enables ports $40..$6F and confirms access. If you hear a familiar tune, all is well! Of course, you must have an old-style PC speaker!

2.3 Release notes

Late changes to the compiler and target code are documented in release note files. These are called *Release.xxx.txt* and will be found in the relevant directories. They are of particular value when upgrading from one version of the compiler to the next. Please read them!

The most important of these are the compiler and target CPU release notes which are kept in the *Docs* or *Doc* directory. They will be called *Release.XC7.txt* and *Release.cpu.txt*, where for example *Release.51.txt* refers to the 8051 compiler and *Release.Cortex.txt* refers to the ARM Cortex compiler.

3 System components

Now that you have installed the development system, you may be wondering what you have got. The development system consists of:

- MPE Forth cross-compiler with source code. Note that VFX compilers are only supplied with source code after a non-disclosure agreement (NDA) has been signed. The source NDA is in the supplied package.
- Source code for generating a target Forth that includes a standalone Forth interpreter useful for debugging with a terminal. Treat the target code as a resource for you to read and extend.
- Source code for generating an Umbilical Forth that needs the cross-compiler for interactivity and debugging. An Umbilical Forth is smaller than the standalone Forth. Treat the target code as a resource for you to read and extend.
- Drivers for CPU or chip specific utilities.
- The AIDE development environment. AIDE is documented in a separate manual.
- Tools directory. This includes file format converters from the memory images generated by the MPE Forth compilers to Motorola S-record format and Intel Hex format. The *OMAKE* make utility is also included.
- Documentation directory. This directory includes much useful documentation, including the ANS Forth specification for target code reference. There are many CPU specific files taken from manufacturers web sites. You will find here the *Release.xc7.txt* and *Release.<cpu>.txt* text files which document late changes since this manual was generated. You will also find PDF files for the latest available version of this manual, *XC70man.pdf* and the CPU specific manual.
- Target Code manuals. From v6.2 onwards, target code code is documented using MPEs DocGen system supplied with VFX Forth. The manual for the common code may be found in *Common\Manual\CommonCode.pdf* and the CPU specific code manual may be found in *<cpu>\Manual\<cpu>Code.pdf* where <> is replaced by a CPU specific reference.

By default the installer creates the directory structure shown in the figure below. Note that the AIDE directory is not shown as this can be installed to anywhere on your system. If you have more than one cross compiler, you only need to use a single copy of AIDE.

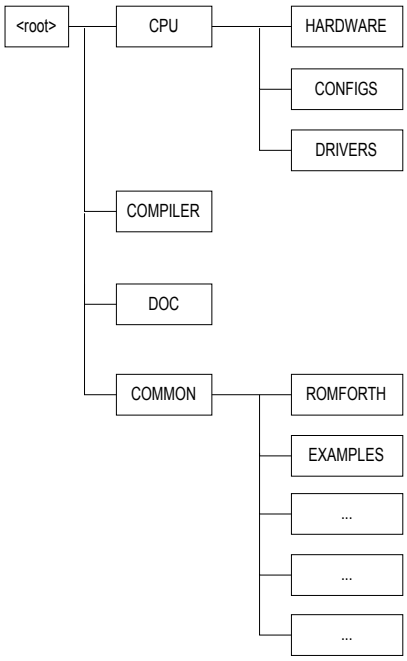


Figure 3.1: Installed directory structure

3.1 MPE Forth cross-compiler

The cross-compiler can generate either a ROM target Forth or an Umbilical Forth from your source code. The source code for the cross-compiler is supplied so that you can extend the compiler and rebuild it from scratch if required. Source code for VFX compilers is available after a non-disclosure agreement has been signed.

The compiler can automate the generation of paged targets and also has a built-in cross-assembler and disassembler (STC/NCC) targets only. The compiler executable and associated files are in the directory *COMPILER* and the source is in the directory *COMPILER\SOURCE* if provided.

3.2 Standalone target Forth

A standalone target Forth is supplied as source code with all compilers except the IRTC verions.

The Forth generated can have a multitasker and software floating-point. The standalone Forth can be debugged through a serial port or other link using a terminal, terminal emulator or Telnet. This permits on-site debugging without the cross-compiler very easy, and the Forth can be used for debugging, maintenance, and configuration.

A stand-alone Forth has a bigger wordset than an Umbilical Forth (see below), and consequently requires more memory. The installer places the target source code in the directories *Common* and <*cpu*>. See the chapter on *Generating your Forth kernel* for details.

3.3 Umbilical Forth

An Umbilical Forth is one in which the interactivity of Forth is provided by the cross compiler talking to the target while the target is running. Because all the name searcches are performed on the host PC,an Umbilical Forth kernel can be much smaller than a standalone Forth, typically 2kb for an 8/16 bit CPU.

Umbilical Forth does not have all words defined in a stand-alone target Forth, but is useful if code space is at a premium. The Umbilical Forth source code is in the directories *Common* and <*cpu*>. In most cases (except for Harvard architectures such as 8051 and Z8) the code for Umbilical Forth systems is compatible with the standalone Forth source code, so additional words required can be taken from the standalone Forth code base.

3.4 Documentation directory

Much of the documentation is available in the *DOCS* directory. In particular note the *ANSFORTH* directory. If you need it the ANS specification is provided in HTML format in the *DOCS\ANSFORTH* directory. Start with

The generic cross compiler and CPU specific manuals are supplied as PDF files. The use of PDF manuals enables us to update our manuals on a regular basis to incorporate suggestions made by you, the users.

A number of CPU manuals are also provided in PDF form to avoid you having to download them.

3.5 Control files

In nearly all cases, the cross compilation process is controlled by a master file that we call a control file. The control file defines the characteristics of the target hardware and memory layout and specifies which files to compile. You will find several in the <*CPU*>*CONFIGS* or <*CPU*>*Hardware* directories. For your job, copy one of the existing files and modify it as required.

You can make your life much easier, especially when you go on site with a laptop, if you use the text macro system described in the next chapter to handle the directory structure for your applcation code and the MPE kernel code.

3.6 Compiler versions

There are three versions of the cross compiler, Developer, Standard and Stamp.

- Developer. The full compiler with all tools, cross compiler source code, stand-alone and Umbilical target source code, floating point, multitasker(s), timebase system, heap, state machine compiler, automated test code, NetBoot and SerBoot monitors, PID loops, and support for bank-switched targets. Compilers for 32 bit targets include the FAT12/16/32 file system (including SD/MMC card drivers), PowerFile file system and the PowerView embedded GUI.
- Standard. The same compiler and target as for Developer, but without:
  - compiler source code
  - PowerNet option

  - PowerView
  - FAT12/16/32 file system
  - NetBoot and SerBoot
  - PID loops
  - Bank-switched code support
- Stamp. As Standard, but with resticted Flash (code) and RAM sizes and without:
  - Cross reference tools
  - PowerFile file system
  - Floating point
  - State Machine compiler

3.7 Learning Forth

If you are unfamiliar with Forth, MPE can supply a range of books and training courses. The book Programming Forth is supplied as a PDF in the *Docs* folder. For further details, please contact our office or look at our website (URL at start of manual). See also the Further Information chapter of this manual.

4 How Forth is documented

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

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

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

4.1 Forth words

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

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

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

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

- The definition line.
- The description.

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

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

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

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

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

This is a "quick reference" comment.

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

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

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

4.2 Stack notation

```
before -- after
```

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

An action on the return stack whill be shown

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

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

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

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

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

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

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

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

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

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

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

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

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

4.3 Input text

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

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

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

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

4.4 Other markers

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

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

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

`M` Affected by multi-tasking

`U` A user variable.

5 Configuring with macros

Both the compiler and the IDE can be configured using text macros, which are mostly used to define directory, file and path names. The IDE and the cross compiler each have their own independent sets of macros.

The macro system gives you great flexibility in managing your source code. For example, you can establish projects in which your source code is held quite separately from the issued MPE code. When a project is moved from one machine to another, the directory structure may need to change. With macros this is easy to do by redefining the macros.

5.1 Text macros

Text macros allow a similar function to the role of constructs such as *%PATH%* in MSDOS batch files. In particular, the expansion of these macros are performed on file names submitted to `INCLUDE <name>`, so something like the following piece of code can be included in a control file before the CROSS-COMPILE directive:

```
"" C:\MSD\SRC" SETMACRO ROOT
...
INCLUDE %ROOT%\FILEA
INCLUDE %ROOT%\FILEB
INCLUDE %ROOT%\FILEC
```

When the file name is scanned, the compiler attempts to subsitute text between the % characters. The % characters are not part of the macro name. Note that `C" <text>" SETMACRO <name>` can be placed on the cross compiler command-line and thus you can specify a directory in a short-cut, batch file or shell script.

The compiler can be used independently of AIDE. Consequently most MPE-supplied control files are independent of AIDE, and define any required macros at the start of the control file. The following example is taken from an ARM control file, and shows macros with both relative (to the current directory) and absolute paths.

```
"" ..\..\..\Common"
           setmacro CommonDir   \ where common code lives
"" ..\..\..\ARM"
           setmacro CpuDir      \ where CPU specific code lives
"" ."       setmacro HWDir      \ where board specific code lives
"" ..\..\..\Examples"
           setmacro ExampleDir  \ Examples
"" ."      setmacro AppDir      \ where application code lives
"" C:\buildkit.dev\software\AddOns\PowerNet\Dev"
           setmacro IpStack     \ where PowerNet lives
...
 include %CpuDir%\Drivers\serSTR91xqi  \ queued interrupt driver
```

5.2 Directory structures

For reference, the directory structure of the cross-compiler is listed below with a description of each directorys contents. Because the supplied files are mostly source code, we strongly recommend that you browse the installed system.

```
Directory               Contains
<root>                  Installer files
  CPU (e.g. 8051)       CPU-specific kernel source files
    Configs             Example control source files
    Drivers             Serial and other driver source files
    Hardware            Board and chip specific code
    Manual              CPU specific manual and DocGen files
  COMPILER              Compiler .EXE and error messages files
    CPU (e.g. 8051)     CPU specific cross compiler source code
    CommonVfx           Cross compiler common source code
    VfxForth            Host Forth for the cross compiler
  DOC                   Help files and other documentation
  Common                Non CPU-specific kernel source files
    ROMFORTH            Chip-independent ROMFORTH source files
    Manual              Common code manual and DocGen files
    Tests               MPE and ANS test harnesses
  Examples              Chip-independent test and example source
  AIDE                  AIDE executables, data, configuration files
```

The <root> directory name is selected by the user during installation. Because AIDE's configuration file contains all the required information to run a given compiler and because all of the other files are common, several cross-compilers can share the same AIDE directory and configuration.

6 Generating a target Forth kernel

This chapter describes how to generate a target ANS Forth for your target board. Generating a stand-alone Forth and generating an Umbilical Forth are essentially the the same process, so the differences for Umbilical Forth are noted at the end of the chapter.

This chapter guides you through:

- setting up your hardware and software
- writing the serial line drivers
- modifying the memory map for your board
- compiling and running a target Forth

6.1 Is your target already supported?

Supplied with the cross-compiler are configurations for a number of boards and terminals. If one of the supplied control files matches your hardware, use it. By using these files, the installation of a target Forth for your board will be greatly simplified. If you do not have one of the supported targets you will have to modify a control file and write serial line drivers for your board. If you are doing this for the first time, take it slowly and test everything at each stage. We strongly recommend the investment in a board we already support. These boards are much cheaper than your wasted time and frustration when becoming familiar with a new package.

6.2 The control file

A control file is the master file for a specific project. It contains the target description, including the memory map, crystal speeds and UART details. These details include:

- the memory map of your board
- whether you wish a log to be displayed
- the clock rate of your board
- the serial port speed.

As well as containing configuration information, the control file contains compiler directives and a list of files that are to be cross-compiled. Once the cross-compiler knows these items, it can generate a correct binary image from your source code. An example control file is shown in the chapter on Controlling compilation. To create a new control file, copy an existing one and then modify it to match your target. This is normally easier than generating one from scratch. Example control files are in the directory <*CPU*>*CONFIGS* and/or <*CPU*>*Hardware*\<*dev/board*>.

6.3 Memory map

The memory map describes the addresses where the ROM and RAM areas start and end in your target system.

6.3.1 Setting the memory map

The memory map is described in your control file, so once the file has been created, you can change the memory map definition to match your target. The memory map is described in three parts:

- the start and end of Flash - where the code is.
- the start and end of initialised RAM
- the start and end of uninitialised RAM

6.3.2 Start and end of Flash

The start and end of ROM (and any other memory area) is defined by using the compiler directive `SECTION` in the form:

```
rom-start rom-end CDATA SECTION <name>
```

where *rom-start* is the address of the start of Flash used for code, *rom-end* is the address of the end of Flash used for code, and `<name>` is the name of the output file. The compiler automatically gives the filename <*name*> an extension *.IMG* so `<name>` must be just a name without an extension. The numbers *rom-start* and *rom-end* are, by default, in decimal, but can be entered in hex by preceding them with a $ character, e.g

```
$0100
```

This area also contains any data defined by `CDATA` during the cross-compilation. This directive is discussed in detail elsewhere in the manual. In practice, it just means that what follows is code.

The first `CDATA` section defined must be the one entered when the system powers up.

6.3.3 Start and end of initialised RAM

The start and end of the initialised RAM area is defined by using the compiler directive `IDATA SECTION`, i.e.

```
ram-start ram-end IDATA SECTION <name>
```

where *ram-start* is the address of the start of RAM, *ram-end* is the address of the end of RAM and `<name>` is the name for this area of memory. The numbers are, by default, in decimal, but can be entered in hex by preceding them with a '$' character.

The initialised RAM area contains any data defined by `VARIABLE`, `VALUE` or `IDATA` during the cross-compilation. These directives are discussed elsewhere in this manual. If an interactive Forth is compiled for the target then definitions entered interactively are placed in this section. The data in `IDATA` sections is appended to the first `CDATA` section before the file is saved.

6.3.4 Start and end of uninitialised RAM

The start and end of the uninitialised RAM area are defined by using the compiler directive `UDATA SECTION`, used in the form:

```
ram-start ram-end UDATA SECTION <name>
```

where *ram-start* is the address of the start of uninitialised RAM, *ram-end* is the address of the end of RAM and `<name>` is the name for this area of memory. The numbers *ram-start* and *ram-end* are, by default, in decimal, but can be entered in hex by preceding them with a '$' character.

The uninitialised RAM area contains data areas allocated by `BUFFER:` or `UDATA` during cross-compilation.

6.3.5 Setting the compilation areas

There must be at least one section of each type `CDATA`, `IDATA` and `UDATA`. Because defining a section also selects it, it is good practice to name one of each section type and then select the current `CDATA` section as the current section type, e.g.

```
$00000 $07FFF CDATA SECTION Kern
$08000 $0FFFF IDATA SECTION KernI
$10000 $1FFFF UDATA SECTION KernU
Kern KernI KernU
CDATA
```

This indicates three areas of memory with names `Kern`, `KernI` and `KernU`. With this setup, your kernel will have 32k of ROM and 32K for variables and interactive development, plus 64k of uninitialised RAM that is not affected at power up.

6.4 Modifying the serial line drivers

Your target board initially communicates with the external world via a UART. Drivers are supplied for the supported targets. If you are using one of these, the appropriate supplied serial driver code can be used. This is located in the directory *<cpu\>Drivers*. Look here first, as new drivers may have been added since the manual was written.

If you are using a UART for which driver code is not supplied, you will have to write all the words required to:

- initialise the UART(s) with a word named `INIT-SER`,
- send a character,
- receive a character,
- test if a character has been received.

All four words are usually Forth CODE definitions if the VFX code generator is not available. This is required so that the send and receive words are as fast as possible. Example serial line drivers in the directory *<CPU>\Drivers* can be used as a template. As with the control file it is normally easier to modify an existing serial driver file rather than creating your own from scratch. The four words are then used to create the serial device words used by the device driver.

Two types of serial handler can be written:

- interrupt driven
- polled

6.4.1 Interrupt driven

An interrupt driven serial line can only be used if the UART generates interrupt signals when characters are received. An interrupt driven driver will allow buffered serial communications to

be implemented with least processor overhead. Interrupt-driven drivers are a little more difficult to write than polled drivers.

6.4.2 Polled

A polled driver will continuously poll a status bit in the UART to detect when the UART has either transmitted or received a character.

6.4.3 Initialising the serial line

The word `INIT-SER` performs all the UART initialisation. This includes setting:

- the baud rate
- any handshaking required
- the number of data bits
- the number of stop bits
- the parity to be used

By default all MPE code assumes that the serial line uses 8 data bits, no parity, 1 stop bit. A three wire link (TX, RX, GND) is all that is required.

It is recommended that the baud rate is initially set to 9600 baud until the target board is working. It can then be raised to make a more responsive target.

6.4.4 Sending a character to the host

The target code needs to be able to send a character to the host for display on the terminal. Therefore, you need to write a word which:

- waits for the transmit line to become available
- transmits a character to the host

The method used can be either a polled or interrupt driven driver. The stack effect of the word is:

```
serEMIT    \ char -- ; send char to host
```

6.4.5 Receiving a character from the host

The target code needs to receive a character from the host. To do this it needs to:

- wait for a character to be received - `serKEY?`
- place the character on the Forth stack - `serKEY`

`SerKEY?` should return true (-1) on the data stack if a character is available, or return false (0) if a character is not available. The stack effect of `serKEY?` is:

```
serKEY?     \ -- t/f ; true if character received
```

The word that receives a character is `KEY`, and the primitive for a serial line may be called `serKEY`. serKEY \ – char ; wait for char to be received

6.4.6 Generic I/O device table

MPE targets developed in the last ten years or so use what we call *generic I/O*, which allows KEY, EMIT and friends to be directed at will to any I/O device that follows the rules of generic I/O. The device can be a UART, a file, an LCD controller, a memory buffer, or a Telnet session running over TCP/IP. Generic I/O is discussed in detail in a separate chapter.

The primitive words are used to generate equivalents of the words KEY, KEY?, EMIT, TYPE and CR. Harvard targets have one more word, TYPEC. The words for EMIT, TYPE and CR must not manipulate the counter OUT as this is taken care of in the Forth kernel.

The first example is taken from a driver for a single-chip ARM.

```
Cdata
create Console0   \ -- addr ; OUT managed by upper driver
  ' serkey0 ,              \ -- char ; receive char
  ' serkey?0 ,             \ -- flag ; check receive char
  ' seremit0 ,             \ -- char ; display char
  ' sertype0 ,             \ caddr len -- ; display string
  ' sercr0 ,               \ -- ; display new line
```

The next example is taken from an 8051 implementation, showing the table for a Harvard architecture device.

```
Cdata
create SerConsole       \ -- addr ; OUT managed by upper driver
tasking? [if]
  ' (mserkey) ,         \ -- char ; shedule and receive character
[else]
  ' (serkey) ,          \ -- char ; receive char
[then]
  ' (serkey?) ,         \ -- flag ; check receive char
  ' (seremit) ,         \ -- char ; display char
  ' (sertype) ,         \ caddr len -- ; display string
  ' (sercr) ,           \ -- ; display new line
  ' (sertypec) ,        \ caddr len -- ; display string from CDATA space
```

6.5 Setting up the system

Setting up the system involves both hardware and software. The target hardware, PC, Flash/EPROM emulator/programmer and serial line have to be connected as well as configuring a terminal program to run the cross-compiler.

6.5.1 Setting up the hardware

- A PC,
- A serial cable,
- A target board,
- A Flash/EPROM programmer, emulator or downloader.

Your PC needs to have at least one serial port for connecting to the target, so making the Forth interactive. The default serial port for Umbilical Forth is set in the umbilical control file. For standalone targets, AIDE's PowerTerm terminal emulator defaults to COM1.

MPE Forth systems only require the serial line to use transmit, receive and ground connections. The serial drivers in AIDE and the cross compiler use no trickery. They will also work with the vast majority of USB to RS232 coverters.

6.5.2 Setting up the software

To compile source code that generates a standalone Forth target, configure the cross-compiler to use the control file you have just selected or created. The easiest way to do this is to modify the AIDE configuration to add a new tool for your project.

6.6 Cross-compiling

Now that the hardware and software are set up, you can cross-compile the source code to generate an executable image.

6.6.1 Creating an image

To cross-compile the source, ensure that the cross-compiler macros are set up correctly and point to your control file. If you cannot be bothered with macros, just use absolute path names in AIDE. Press the toolbar button to begin compilation. The compiler displays its sign-on message and compiles the source code.

6.6.2 Log display

Following the compiler sign-on, depending on the compiler settings, you should see the cross-compile-log. As each word is compiled the compiler displays the word's address, its type and its shortened name. The type of item is coded as two characters as in the following table.

Code	**Compiled type**	**Code**	**Compiled type**
VR	Variable	FV	FP variable
CN	Constant	FC	FP constant
LB	Label	FA	FP array
:	Colon definition	EQ	Equate
CD	CODE definition	CR	CREATEd word
DF	DEFERred word	US	User variable
VC	Vocabulary		

Table 6.1: Log display indicators

The output can be sent to a file or to the printer. Note that having the log on the screen slows down the compiler, but is useful when you have a compilation errors or debug information to display. The scroll bars allow the log to be reviewed before the compiler finishes, and portions of the text can be sent to the printer using the File menu or AIDE's right-click menu.

6.6.3 Turning the log on and off

Instead of having the data displayed for each compiled item, the log can be turned off. The advantage of this is that the compiler spends less time displaying data and so cross-compilation is quicker. To do this, change the compiler directive in the control file from `LOG` to `NO-LOG`. The log can be turned on again by replacing `NO-LOG` with `LOG` in the control file.

6.6.4 Log to file or printer

The cross-compiler can redirect the log to a file instead of the display. To do this, use:

```
FILE: <name>
```

Under Windows, to send the log to a printer, use:

```
PRN:
```

6.6.5 Compilation summary

Once the cross-compiler has finished cross-compiling source code, it displays information about the compilation. This includes:

- any unresolved references
- the number of forward references made and the number of unresolved
- (outstanding) forward references
- the size of the compiled image
- the initialised RAM table address and length
- section information
- the compilation time

Unresolved references are words that are referenced in the source code but are not defined. These can be due to spelling mistakes or not compiling some of your code.

If there are any unresolved forward references, your target may not work, and the compiler tells you so.

The size of the compiled image is the amount of actual code output into the file. The actual file size will be the size of the ROM indicated by the memory map.

The RAM table is the place in ROM where initial data for the initialised RAM section is stored. When the target board is reset, initialisation code copies this table into the initialised RAM areas.

6.6.6 The created image

The cross-compiler always creates a straight binary image file with a *.IMG* extension. It can be

downloaded to a Flash/EPROM emulator or programmer. The file has the name given when defining the memory map using the `SECTION` directive. It has the extension `.IMG` by default, which can be changed using the directive `setBinExt`, e.g.

```
setBinExt .bin
```

A range of alternate file formats is also supported, but the required one has to be selected by a compiler directive.

```
HEX-I16    \ -- ; Intel Hex used for 8-bit CPUs
HEX-I32    \ -- ; Intel Hex for 32 bit addresses, e.g. ARM
HEX-S19    \ -- ; Motorola S19 - 16 bit addresses
HEX-S28    \ -- ; Motorola S28 - 24 bit address, e.g. 9S12
HEX-S37    \ -- ; Motorola S37 - 32 bit address, e.g. Coldfire
ELF-ARM    \ -- ; ELF file for ARM or Cortex
ELF-386    \ -- ; ELF file for 386
ELF-FORMAT \ machine flags -- ; generic ELF file
```

When programming paged Flash, e.g. for a 68HC12/9S12 CPU, programming tools often require a physical base address in the Flash, rather than the 64k addresses used in the `SECTION` and `BANK` definitions. When a hex file is output, the base address of a section can be overridden using:

```
physaddr SetFlashBase
```

This situation can also arise in CPUs, e.g. some ARMs, for which the Flash address used by programming tools or in-system loaders does not match the normal run-time address of the code.

The initial execution address can be set for S28, S37 and ELF formats by:

```
<addr> SetBoot
```

6.6.7 Problems, problems ...

If an error occurs during compilation, the compiler will stop and display the line on which the error occurred. The cross-compiler shows the line number and the file name where the error occurred as well as the type of error that has occurred.

If you are working with AIDE, you can use the *IDE -> Configure* option to define your editor and the editor will then display the offending line after an error.

If you are using the compiler in stand-alone mode, you can set it to call the editor on error. The Windows and Linux versions have different configuration mechanisms.

6.7 Downloading the compiled image

Once the source code has been compiled the image needs to be downloaded to Flash or EPROM using a chip-specific utility, Flash/EPROM emulator or programmer.

If the board already has a Forth running on it, the Forth may include the MPE `REFLASH` utility. This utility erases the on-chip Flash, tells AIDE's PowerTerm that a new image file is needed, and

downloads and programs the selected image file using an Xmodem protocol. PowerTerm's file server must be enabled. To use the `REFLASH` utility, just connect to the board using PowerTerm, and type:

```
reflash
```

The source code for `REFLASH` is provided with the target source code. Consult the target Forth manual and target source code. `REFLASH` is present for most ARM, Cortex-M3 and Coldfire CPU targets, and may be present for other CPUs.

For Coldfire and some other targets, downloading to Flash through a BDM or JTAG unit is supported. Consult the target-specific manual and target sources.

6.7.1 Downloading to Flash

The *Tools* directory will contain CPU or chip-specific download tools whenever we have written them or the owner permits free distribution of them. You can add a short cut on your desktop or by adding an external tool to AIDE.

6.7.2 Downloading to an emulator or programmer

The binary image can be downloaded to any Flash/EPROM emulator as long as the emulator's software supports binary image files or one of the available alternate file formats.

6.8 Running the target Forth

Once the image generated by the compiler has been downloaded to the target, it is ready to be reset and the Forth tested.

6.8.1 Switching to target mode

To receive characters from the target, run and configure your terminal program. All versions of Windows are supplied with terminal emulation programs. The cross-compiler IDE also comes supplied with its own terminal emulator *\zi{PowerTerm}.

6.8.2 Resetting the target board

Once the image has been downloaded, you can reset the target board. You can either use the reset supplied on the board or power off and on again.

6.8.3 The sign-on

- the serial line drivers
- the memory map definition
- your target board
- your EPROM emulator/programmer
- Direct port access

Each of these should be checked.

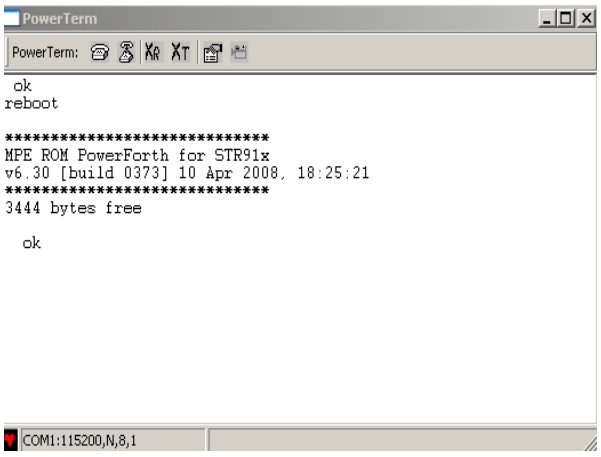


Figure 6.1: Target sign-on

Serial line drivers

If you do not get the sign on message, your transmit word might not be working correctly. You can check that you can transmit a character up the serial line, by appending code for emitting a character up the serial line, onto the end of the initialisation word `INIT-SER`. Therefore a character can be transmitted and seen early in the initialisation sequence. By default all MPE code assumes that the serial line uses 8 data bits, no parity, 1 stop bit. A three wire link (TX, RX, GND) is all that is required.

Memory map definition

If the memory map for the ROM definition is wrong. The target may not sign-on at all. If the definition of the RAM memory map is wrong, the target may sign-on but may display 'garbage'.

Target board

- Is the serial line connected?
- Has your target board got power?
- Flash/RAM plugged in correctly?
- Are jumpers set correctly?
- Is it still in download mode?

EPROM/Flash emulator/programmer

Check to see if your emulator is emulating an EPROM/Flash that your target board is expecting. If you have the wrong type set, your target will not sign on.

Testing the Forth - an example

Once the Forth has signed-on, you need to test that it iss working properly. Type `WORDS`, this will display all the Forth words available. If this works then type in:

```
: FORTH-TEST   \ -- ; A quick test for forth
  ." HELLO"
;
FORTH-TEST
```

This should display:

```
HELLO
```

followed by the ok prompt.

6.9 Cross-compiling an application

Once your Forth is working on your target board, you will now want to compile your application code.

6.9.1 Modifying the control file

Once new code has been written, you can add it to the control file. Near the bottom of the control file, there is a list of commands in the form:

`INCLUDE <name>`

To compile your application files you add them to the end of the list, although normally before the line that reads similar to:

`INCLUDE ...\LIBRARY`

6.9.2 Running your application

To compile the application you need to:

- run the cross-compiler
- download to the Flash/EPROM emulator/programmer
- apply power and reset the target

The target board signs-on. You can now test your application.

6.10 Generating a turnkey application

Once you have written your application, you will want to make it start when the target board is reset. This is known as a turnkey or autostarting application.

6.10.1 Using MAKE-TURNKEY

To make an application turnkey, use the directive `MAKE-TURNKEY` in the form:

```
MAKE-TURNKEY <name>
```

where `<name>` is the name of the word to run at startup. The word `<name>` must be defined before using this directive. The example generates a simple turnkey application when cross-compiled. If you require the use of serial communications, the multitasker, the heap, or leds, you must initialise them in your application. To initialise the serial communications use the word `INIT-SER`. To initialise the multitasker use `INIT-MULTI`. Note that `(INIT)` must be called so that initialised data can be copied into RAM etc.

```
: RUN
  (INIT)          \ Init. system (Mandatory!)
  INIT-SER        \ Init. the serial line
  INIT-MULTI      \ If multitasking
  INIT-HEAP       \ If using the heap
  0               \ counter
  BEGIN
    CR " Hello world!" dup .
```

Figure 6.2: Example turnkey application

The word `COLD` in *kernel62.fth* is the default action.

```
: COLD          \ --
  (init)                \ start Forth
  init-ser              \ initialise serial line
  console opvec !       \ default i/o channels
  console ipvec !
[ SIZEOFHEAP ] [if]
  init-heap             \ initialise memory heap manager
[then]
[ paged? ] [if]
  init-pages            \ initialise paging
[then]
[ tasking? ] [if]
  init-multi            \ initialise multi-tasker
[then]
[ romforth? ] [if]
  relink                \ initialise ROM PowerForth
[then]
[defined] WalkColdChain [if]
  WalkColdChain         \ execute user specified initialisation
[then]

  CR .cpu               \ sign on
  cr .free              \ display free space
  cr cr ."   ok"        \ display prompt
  s0 @ sp!              \ reset data stack
[ 32bit? ] [if]  FlushKeys  [then]      \ flush UART input
  quit                  \ start text interpreter
;
make-turnkey cold       \ Default start-up word.
```

A few items such as UARTs and the multi-tasker are initialised, and then the bulk of the initialisation is performed in the cold chain. Words with a null stack effect `( -- )` are added to the cold chain using `AtCold`. After that, the Forth interpreter is started.

If your application does not use the Forth interpreter, you can write your own version of `COLD` based on the original, but containing the endless loop that is your application. If you have enough Flash and RAM space, we strongly recommend that you keep the Forth interpreter. With a suitable comms link such as TCP/IP, you can talk to the running Forth from anywhere in world. This is much cheaper than going on-site to fix a simple configuration problem.

If your application is not interactive, the compiler directive `NO-HEADS` can be used to reduce the size of the application. `NO-HEADS` removes all word headers, whereas `INTERNAL` and `EXTERNAL` can be sprinkled through your code.

6.10.2 Using ATCOLD

`AtCold` is usually used in the form:

```
  ' foo AtCold
```

so that `foo ( -- )` is executed at start up. The word must have a null stack effect. See the target manual for details of the implementation. The cold chain is executed by `WalkColdChain`.

Although `AtCold` is mostly used to run initialisation routines, it can be used to run an application word. This is particularly useful if a back-door is provided so that the application word can drop out into the Forth interpreter. If you have enough RAM and CPU performance, you may prefer to write the application as a separate task started by by a word run in the cold chain.

```
' StartApp AtCold
```

This way, the Forth interpreter is run in one task, and the application is run in another.

6.11 Umbilical Forth

An Umbilical Forth system has no interpreter on the target, which saves code space. To provide the usual Forth interactivity during development, the cross compiler provides the text interpreter and passes execution addresses to a small message handler in the target.

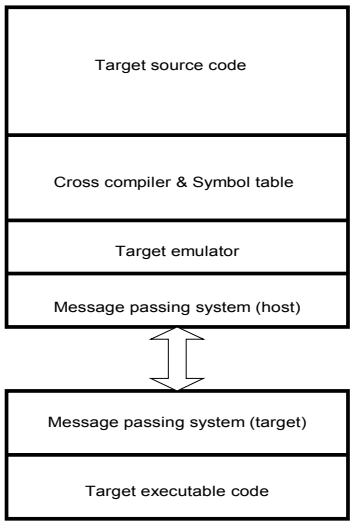


Figure 6.3: Umbilical Forth structure

Generating an Umbilical Forth system is very much the same as generating a stand-alone Forth, but is different in terms of the kernel files, communication link, and what happens when the cross-compiler has finished compilation.

An advantage of Umbilical Forth, especially for 8 bit CPUs, is that all the host tools, including compiler, assember and disassembler, are available during interactive debugging without consuming target resources. The disadvantage is that interactive debugging requires the cross compiler, target source code and the Umbilical link.

See one of the example control files for details of the files that are compiled.

6.11.1 Comms links

Most Umbilical Forth systems use a UART in the same way as a standalone Forth. However, in an Umbilical Forth some characters are used as triggers. This link is sometimes called the "cross target link" or XTL. After compiling the serial driver, you just need to tell the Umbilical Forth message passer in *Common/targend.fth* which driver words to use. The example below is for an LPC932 (an 8051 derivative).

The cross compiler needs 8 data bits, no parity, 1 stop bit. A three wire link (TX, RX, GND) is all that is required.

```
include drivers\SerLPC932ui    \ serial i/o
Synonym wait-byte  (serkey)    \ Say which XTL drivers to use
Synonym send-byte  (seremit)
Synonym Wait-Byte? (serkey?)
Synonym Init-XTL   Init-Ser
include Hardware\LPC932\IAP932 \ IAP Flash routines
include %DIRROMCOM%\targend    \ message driver
```

Umbilical Forth is not restricted to a serial line. We have used I2C, SPI and others for various CPUs. However, whereas in the past these protocols used to be implemented by bit-banging the PC parallel port, the parallel port is now disappearing from PCs and is being replaced by USB. In consequence, many silicon manufacturers provide USB widgets for communicating with and debugging the silicon.

If the serial link is being seriously stubborn, you can display the serial traffic. When serial debugging is enabled, characters are displayed as hex bytes. Characters transmitted by the PC are in the form `<xy>`, and characters received by the PC are shown in the form `[ab]`.

```
+SERIAL-DEBUG   \ -- ; enable serial debugging
-SERIAL-DEBUG   \ -- ; disable serial debugging
SERIAL-DEBUG?   \ -- flag ; true if debugging
```

Some Umbilical Forth link drivers are specific to various CPU types and families, and are described in the target specific manuals. Note that there are two parts to the Umbilical system, the link driver which handles communications during debugging, and the memory driver (see below) which handles programming of the CPU code space. New drivers can be installed at any time, and users wishing to write a new driver can contact MPE for further details. MPE is also available to develop new drivers for you.

- Atmel 89S8252 SPI link Umbilical link and programming
- 8051 SPI access Umbilical link only
- BDM for 9S12 and CPU32 cores such as the 68332 Umbilical link plus RAM and limited EPROM/Flash drivers
- JTAG and SWD for ARM and Cortex cores Umbilical link plus RAM and Flash drivers. The Segger J-Link is fully supported by the ARM/Cortex compilers.
- JTAG for MSP430 cores Umbilical link plus RAM and Flash drivers.

Note that for Windows NT upwards or Linux, direct port I/O is a privileged operation. Under Windows you must install the file *NTPORT.EXE* file from the *COMPILER\XTRA* directory as described in the installation section of the manual and modify your control file to include the *NT-ACCESS-PORTS* directive.

6.11.2 Serial line configuration

The cross compiler needs 8 data bits, no parity, 1 stop bit. A three wire link (TX, RX, GND) is all that is required. The serial line is usually defined near the end of the control file.

```
umbilical? [if]                    \ in devlopment mode
  make-turnkey run-umbilical       \ cold start to Umbilical Forth
  C" COM1:" console-speed serial   \ define link driver
  c" dtr=off rts=off" set-control      \ define link state
  umbilical-forth                  \ switch to interactive mode
[else]
  make-turnkey cold                \ cold start to application
  finis                            \ done with cross compiler
[then]
```

The words `serial` and `set-control` merely save data for when the umbilical link is actually opened.

```
: serial        \ port$ baud --
```

Use in the form

```
 c" COM1" #9600 SERIAL
```

to define the port being used and the baud rate.

```
: ser-control   \ control$ --
```

Use in the form:

```
  c" dtr=off rts=off" ser-control
```

to define any additional configuration.

The formats of the device and control strings are operating system specific.

Windows

Under Windows, the control string has the same form as in the MODE command. Type *mode /?* in a DOS box for the gory details.

```
MODE COMx
BAUD=b PARITY=p DATA=d STOP=s
to=on|off xon=on|off odsr=on|off
octs=on|off dtr=on|off|hs
rts=on|off|hs|tg idsr=on|off
```

COMx and BAUD are set by the configuration of `SERIAL`. If you need to use a COM port whose number is greater than 9, as is common for USB ports, use the form:

```
 c" \\.\COM14" #115200 SERIAL
```

By default, the control string is set to:

```
  parity=n data=8 stop=1  dtr=on rts=on
```

Mac OS X

The serial ports used to be of the form */dev/ttyS0*, but USB serial ports will be seen with two options, e.g.

```
/dev/cu.usbserial-FTABCDEF
/dev/tty.usbserial-FTABCDEF
```

Use the */dev/cu.xxxx* form. The "cu" stands for "calling unit" when we initiate the call. The */dev/ttyxxx* devices are for modems that require the DCD line. You can see the available devices by opening a Terminal and typing:

```
ls /dev/cu*
```

The control string defaults to:

```
no parity  8 data  1 stop  1 dtr  1 rts
```

The full set is

```
no|even|odd parity  7|8 data  1|2 stop
1|0 dtr  1|0 rts
Unix|DOS
```

Linux

The (real) serial ports used to be of the form */dev/ttyS0*. When using USB serial devices, the name used varies according to your distribution. The most common names appear to be:

```
/dev/ttyUSBx
/dev/ttyACMx
```

USB serial ports are discussed in more detail at the end of this chapter.

The control string defaults to:

```
no parity  8 data  1 stop  1 dtr  1 rts
```

The full set is

```
no|even|odd parity  7|8 data  1|2 stop
1|0 dtr  1|0 rts
Unix|DOS
```

6.11.3 Memory drivers

The target Flash has to be programmed with the message-passing Umbilical Forth kernel. For many CPUs this can be done using a serial bootloader, most of which accept Intel Hex or Motorola S-record files. For full interactive development, the compiler needs a way to program new code into the target. Increasingly this is done using In-System Programming (ISP) facilities provided by the chip, or by using the manufacturer's debug hardware.

The memory driver architecture of Forth 7 is open. You can use existing drivers as a model. If you need custom memory drivers, MPE can help you.

Once the Umbilical Forth kernel has been programmed and is running, interactive development can start.

6.11.4 Downloading to Flash

Once the kernel and application have been compiled into the target, you can start interactive debugging. There are a few lines at the end of the control file that configure this.

```
umbilical? [if]                    \ in devlopment mode
  make-turnkey run-umbilical       \ cold start to Umbilical Forth
  C" COM1:" console-speed serial   \ define link driver
  c" dtr=off rts=off" set-control      \ define link state
  umbilical-forth                  \ switch to interactive mode
[else]
  make-turnkey cold                \ cold start to application
  finis                            \ done with cross compiler
[then]
```

The words `serial` and `set-control` merely save data for when the umbilical link is actually opened.

When `umbilical-forth` is run, the compiler will prompt you to power up and reset the target. Until you respond to this prompt, the link is not active. The image files have been saved to disk, so you can use an external download tool, e.g. one from the silicon manufacturer, to program the Flash.

The compiler will ask you if you want to download memory sections to the target. The only one you usually need to download will be the `CDATA` section(s).

When the compiler issues the download prompt, the link is not active. At this stage, the image files have been saved to disk, so you can use a separate download tool, e.g. one from the silicon manufacturer, to perform the download. If you do this, just answer 'N' to the download questions **after** you have performed the download. Some compilers contain integration with debug tools so that you can answer 'Y' to the download question. See the target-specific manual for details.

The *Tools* directory will contain CPU or chip-specific download tools whenever we have written them or the owner permits free distribution of them. You can add a short cut by adding an external tool to AIDE.

Once the binary has been downloaded to the target you may have to reset the target board again.

6.11.5 Using In-Application-Programming (IAP)

Some processors allow you to program the internal Flash yourself. Examples of these are the

NXP LPC9xx (8051 core) and LPC2xxx (ARM7 core) families. Umbilical Forth can use these facilities for updating the flash.

To use these, you must provide the words `C!F`, `W!F` and `L!F` (32 bit targets only). These behave like the normal Forth store words, but use the IAP Flash routines if the target address is in the Flash. Your code is responsible for managing sector erase and any required buffering.

When coding the fash routines, you must consider the impact of Flash sector sizes and how much RAM is needed for sector buffering.

6.11.6 Interactive debugging

Once the source code has been compiled and downloaded to the target you can reset the target board. Follow the instructions given by the cross-compiler.

After resetting the target, you will see a message displaying information such as the version number, copyright details etc. The cross-compiler itself displays this message, so the target is not necessarily up and working. To test the target board, you need to execute a target definition. If there is not already a target definition, type:

```
: FORTH-TEST    \ -- ; A quick test
   ." HELLO"
;
FORTH-TEST
```

This should display:

```
HELLO
```

followed by the `T-OK` prompt.

If you have not written the Flash drivers yet and your CPU supports execution from RAM, you should find a section of unused RAM and set the dictionary pointer to that location.

```
<addr> CORG
```

6.11.7 Problems, problems

Most of the problems involved in getting an Umbilical Forth to work come from initialisation problems. By default, the word executed at power up is `RUN-UMBILICAL`, which only executes `INIT-SER`. CPUs with complex peripherals, e.g. ARMs often require more to be done than just this. The cross compiler sets the port to raw mode, 8 data bits, no parity, 1 stop bit, DTR and CTS set. A three wire link (TX, RX, GND) is all that is required.

Proper testing of the serial/XTL link saves time. Find the word `run-umbilical` in *Common\Targend.fth*. Just before the word `message-passer`, insert the following code:

```
begin
  [char] A send-byte   \ can use your word directly
again
```

Download this to the target, and use a serial terminal rather than the interactive Forth. When the target is reset, there should be stream of 'A's on the terminal. Your `EMIT` word is working. To test your version of `KEY`, replace the code above:

```
begin
  wait-byte send-byte
again
```

Every time you press a key in the terminal, you should see its echo on the terminal. Your version of `KEY` is working.

Especially if you are reusing code from the standalone model, you may find that the code relies on variables being initialised at power up. In this case, your control file must set the equate `init-idata?` to non-zero.

```
1 equ init-idata?    \ true if IDATA to be initialised
```

At target runtime, you must execute `INIT-IDATA` to perform the copy from Flash to RAM.

Again when reusing code from the standalone model, you may find that the word `AtCold` is used to add a word to the start up chain. You can either add the `ColdChain` mechanism to your Umbilical target system, or you can explicitly add these words to your start up code. If you take the second option, you will find that the compiler stops and warns you every time. If you provide an `INTERPRETER` version of `AtCold` (see example below) the warnings will be suppressed.

The following example is taken from a control file for an LPC2106 ARM implementation.

```
1 equ init-idata?    \ true if IDATA to be initialised
...
interpreter
: AtCold    \ xt --
  drop
;
target
...
: runUmb        \ --
\ *G Starts Umbilical Forth with additional initialisation.
  init-idata initVIC run-umbilical
;
...
make-turnkey runUmb  \ cold start to Umbilical Forth
```

6.12 Serial port problems

After many decades of using serial devices, one would expect operating systems to deal with them easily. If only it was true. USB serial devices can be problematic.

6.12.1 Windows USB serial devices

When you plug in a USB serial adapter, Windows often fails to tell you which COM port it has become. You find this out using the Device Manager, usually from Control Panel -> System -> Device Manager, then Ports. From a console you can use:

```
start devmgmt.msc
```

When you plug in or remove an adapter, the display will change. If installation fails, you can use View -> Show hidden devices to show unconnected devices and then update or remove the drivers. From a console use:

```
set devmgr_show_nonpresent_devices=1
start devmgmt.msc
```

6.12.2 Windows terminal emulators

HyperTerm is much derided by geeks. It's very old, and does not appear to have been updated for use with USB devices. Commonly recommended free alternatives are TeraTerm and PuTTY. If we only need a single terminal, we use PowerTerm within AIDE.

6.12.3 Mac OS X USB serial devices

When using USB serial devices, the name used varies according to the function. The names are:

```
/dev/tty.usb??????
/dev/cu.usb?????
```

You can list them at a Terminal prompt with

```
ls /dev/tty*
ls /dev/cu*
```

The */dev/tty** devices are for modems waiting for a call into the OS X machine. It is assumed that the DCD line is active. Hence these are of little use for the three wire connections (RX, TX, Gnd) typically used to connect to embedded systems.

The */dev/cu** devices are much better suited for connecting out (calling up) to other systems.

Mac OS X terminal emulators

The ones listed here are just ones recommended by others.

screen - on every Mac. For hardcore Unix buffs.

```
screen /dev/cu.usbserial 19200
http://hints.macworld.com/article.php?story=20061109133825654
```

Coolterm - GUI app.

```
http://freeware.the-meiers.org
```

goSerial - GUI app.

```
http://www.furrysoft.de/?page=goserial
```

The most widely used equivalent to Windows' HyperTerm appears to be *minicom*. It isn't pretty, but it works and is easy to use.

```
http://pbxbook.com/other/mac-tty.html#minicom
```

6.12.4 Linux USB serial devices

When using USB serial devices, the name used varies according to your distribution. The most common names appear to be:

```
/dev/ttyUSBx
/dev/ttyACMx
```

There are several methods of finding USB serial ports. The simplest seems to be to unplug the device, then reconnect it, then type the following incantation:

```
dmesg | grep tty
```

where you must have root access. On many systems, e.g. Ubuntu

```
sudo dmesg | grep tty
```

is required. The last few lines should then tell you which USB serial port, e.g. `/dev/ttyUSB0` was selected for your device. If the last tells you that the device is now disconnected, it is probably because of the "brltty bug". Unless you need the Braille TTY access, remove the package *brltty*. Repeat:

```
sudo dmesg | grep tty
```

to check that device remains connected. Some forums suggest that you may also need to create the */dev/ttyUSBx* entries. Do this with:

```
sudo mknod /dev/ttyUSB0 c 188 0
sudo mknod /dev/ttyUSB1 c 188 1
sudo mknod /dev/ttyUSB2 c 188 2
```

Linux serial terminal emulators

The most widely used Linux equivalent to Windows' HyperTerm appears to be *minicom*. It isn't pretty, but it works and is easy to use. There are plenty of others, including GUI ones, but *minicom* is the one we come back to as it is available for nearly all distributions.

7 Optimising the target Forth

Once you have a target Forth running, you may want to either reduce the size of your image or increase the execution speed of the code. This chapter describes those features of Forth 7 that help you with this aim.

7.1 Reducing the image size

During development you may need to reduce the size of your target image. For example, your application may have grown too large for your Flash space. Reducing Flash requirements is usually done by:

- removing headers
- factoring your code
- removing excess code
- using equates instead of constants
- removing forward references
- using Umbilical Forth

7.2 Removing headers

If you have already been using Umbilical Forth, the compiler will not have generated any heads, so this discussion only applies to a standalone target.

To reduce the size of the compiled image, you can instruct the compiler to compile all or some of the code without heads. For each word defined, the cross-compiler generates a header in the target image. A header is the name of the word stored as a counted string and is used when the target is used interactively. Therefore, by removing the heads of words you reduce the interactivity of your system.

7.2.1 Removing all headers

To remove the heads from all the code, use `NO-HEADS`. The compiler will produce code that will be greatly reduced in size, but cannot be used interactively.

7.2.2 Selectively removing headers

To select a number of words to be made headerless, use `INTERNAL` and `EXTERNAL`. `INTERNAL` instructs the compiler to stop generating headers, and `EXTERNAL` instructs it to generate headers again.

7.3 Factoring your code

Procedures calls in Forth are very cheap, so code reuse of small fragments of code does not have a great performance penalty. By reusing code, your target image size can be greatly reduced. The smaller are the procedures you use, the more easily they can be reused. In addition, small procedures are easy to test. Consequently code written with small procedures is normally more reliable.

Factoring code is something of an art form, but is well worth the effort. A client reports

that MPE's PowerNet TCP/IP stack is half the size of other commercial offerings. When that translates into one million dollars, the savings are apparent. Note also that having to maintain half the number of lines of code is a long-term saving for any product, regardless of volume.

7.4 Removing excess code

During development, debug and test code is often inserted into the sources. This code is easily left in and forgotten about. By stripping out this excess code you can gain more space in the Flash. The easiest way to do this is to use the `XREF` system (not available in the Forth Stamp versions).

The `XREF` system is turned on by using the word `+XREFS` in the control file. All code after `+XREFS` will be cross referenced. Use `XREFS` to turn cross referencing off. Use `XREF-UNUSED` to find which words are unused. The `XREF` words:

```
XREF <name>
XREF-UNUSED
XREF-ALL
```

are always available in Umbilical Forth. For standalone Forths, you can put the compiler into interactive mode by including `INTERACTIVE` before `FINIS` in your control file, or you can include the `XREF` words in your source code.

You can also reduce the size of the code by using the library file mechanism (see Controlling compilation) which enables the compiler to include only those words that have already been referenced.

7.5 Using equates instead of constants

An equate is a constant that just resides within the cross-compiler. It cannot be referenced when interactively debugging your target system. The actual value of the equate is compiled 'in-line' as a literal instead of referring to a constant. You can often save some space on the target board for each constant defined but sacrifice some interactivity. This works if you don't refer to an equate many times, as several instances of an equate compiled in-line may use more bytes than the memory required to store a constant and reference it.

The VFX code generators nearly all treat constants as literals. The trade-off between equates and constants is very architecture dependent.

An equate is defined in a similar way to a constant:

```
xxxx EQU <name>
```

where `xxxx` is the value of the equate and `<name>` is its name. An equate is used in the same way as a constant, by stating its name.

```
$0100 EQU ADDRESS
  ADDRESS CELL + EQU ADDRESS2
: SOME-WORD     \ --
  ... ADDRESS ...
  ... ADDRESS2 ...
;
```

7.6 Removing forward references

When a forward reference is compiled on a subroutine threaded target, the largest available target range branch has to be used. For most CPUs, shorter instructions are available if the destination address is already known. Removing forward references reduces the number of unknown destinations and reduces code size.

The compiler log tells you how many forward references were made. You can find out which words were forward referenced using the directive `.FORWARDS ( -- )`.

7.7 Using Umbilical Forth

If you require a compact target Forth but without the inconvenience of removing target headers, you can use Umbilical Forth. Umbilical Forth gives you a very compact interactive Forth. The Umbilical Forth kernel is about 2.5k bytes for 16 bit targets, and 4k bytes for 32 bit targets. The kernel does not contain all the words in the standalone target, so you may have to write a few words (or copy them from the standalone kernel) to get your code to compile.

7.8 Speeding up your code

The normal way to increase the speed of your code is to code strategic words in assembler. Good candidates for coding are inner loops and words containing a lot of stack manipulation (DUP, SWAP etc.). The VFX optimisers significantly reduce the need to code in assembler. However, some impact can be made by replacing very small definitions by compiler directives. Every time the VFX optimiser has to generate a call, it has to generate what we call a canonical Forth stack. If you replace a short definition by a compiler directive, the optimiser does not call it, but compiles it as if from source code. Thus:

```
:  foo  \ addr -- addr
  3 cells + @
;
```

can be replaced by

```
compiler
:  foo  \ addr -- addr
  3 cells + @
;
target
```

On many target CPUs, especially those with good indexed addressing modes, the resulting code is shorter. Compiler directives allow you to retain the code modularity of short Forth definitions without the calling overhead. In a standalone Forth, a `COMPILER` word also has no head.

8 Generic I/O

8.1 About Generic I/O

Generic I/O allows the Forth words KEY, KEY?, EMIT, TYPE and CR to use any I/O device. The user variables IPVEC and OPVEC each contain the address of a structure for a device. This structure contains a list of Forth words used for the words above.

By using different devices for input and output, input can be from a serial channel and output can be to an LCD screen. The selection can be changed at any time by the application. Because IPVEC and OPVEC are USER variables, i.e. are specific to each task, different tasks may have different I/O devices.

The generic I/O structure consists of any array of five (six for Harvard targets) XTs. The XTs are for the words that perform the following basic functions.

```
cell  KEY action
cell  KEY? Action
cell  EMIT action
cell  TYPE action
cell  CR action
cell  TYPEC action; Harvard CPUs (e.g. 8051) only
```

The CR and TYPE actions are provided to ease implementations of devices such as LCD output in which CR does not naturally correspond to 13 EMIT 10 EMIT, and for which TYPE will be much faster than repeated EMITs. The output functions update the USER variable OUT before calling the action.

8.2 Creating a new device

```
cdata    \ this table goes in CODE space
create SerConsole  \ -- addr ; OUT managed by upper driver
  ' (serkey) ,         \ -- char ; schedule, receive char
  ' (serkey?) ,        \ -- flag ; check receive char
  ' (seremit) ,        \ -- char ; display char
  ' (sertype) ,        \ caddr len -- ; display string
  ' (sercr) ,          \ -- ; display new line
  ' (sertypec) ,       \ caddr len -- ; display CDATA
                       \   for Harvard targets only
```

Generic I/O handles all use of OUT for the output functions. OUT is manipulated before the action is performed so that special cases can update OUT themselves.

When the multi-tasker is used, a multi-tasking version of (SERKEY) must be used. Conditional compilation can be used in the primitive words. The equate Tasking? is set non-zero when multi-tasking is in use.

8.3 Selecting a device

To select serial input, the phrase

```
SerConsole IpVec !
```

is all that is needed. Similarly, to select serial output

```
SerConsole OpVec !
```

is all that is needed.

9 Multitasker

The multitasker supplied with Forth 7 greatly simplifies complex tasks by allowing you to break them down into manageable chunks. This chapter leads you through:

- initialising the multitasker
- writing a task
- communicating between tasks
- handling events

The multitasker is in the file *MULTIxx.FTH* in the CPU directory, where the 'xx' denotes the processor type. Where the CPU (e.g. 8051) uses a different code base for single chip and expanded operation, the files will be called *MULTIxxINT.FTH* and *MULTIxxEXT.FTH*. There are minor differences between the implementations on different CPU cores. A full glossary can be found in the CPU specific target code manual.

To compile the multitasker, most control files need the equate `Tasking?` to be set non-zero, e.g.

```
1 equ Tasking?
```

9.1 Initialising the multitasker

The multitasker needs to be initialised before use.

9.1.1 Selecting the multi-tasker

When set non-zero, the equate `TASKING?` in the control file causes the multitasker to be loaded. Note that `TASKING?` also affects other words such as `KEY` and `MS` so that calls to the scheduler are included by words that can block for a significant amount of time, for example when waiting for human input.

```
xxxx EQU TASKING?
```

The configuration of the multitasker is controlled by other equates which control what facilities are compiled.

```
6 cells equ tcb-size     \ for internal consistency check
  0 equ event-handler?   \ true for event handler
  0 equ message-handler? \ true for message handler
  0 equ semaphores?      \ true for semaphores
```

9.1.2 Starting the multitasker

Before use the multitasker must be initialised by the word `INIT-MULTI`, which initialises the initial task `MAIN`, and enables the multi-tasker. To start the multitasker, use `MULTI`, which starts the scheduler so new tasks can be added.

9.1.3 Stopping the multitasker

To stop the multitasker, use `SINGLE`.

9.2 Writing a task

Tasks are very straightforward to write, but the way tasks are scheduled needs to be understood.

9.2.1 Using the scheduler

The multitasker is software scheduled, sometimes called cooperative. This means that each task relinquishes control back to the scheduler when it is ready to do so. This is different from a pre-emptive scheduler where the scheduler interrupts a task. The word `PAUSE ( -- )` is supplied so that a task can relinquish control to the scheduler. `PAUSE` passes control back to the scheduler, which executes all the other tasks once, and then returns back to the calling task.

9.2.2 An example task

An example task is shown below. The task is an endless loop with the word `WAIT ( u -- )` embedded in it. When `WAIT` is executed, the scheduler reschedules to the next task. The scheduler will not run this task until it has run all other tasks 5000 times. Each time the task is executed, it will emit a beep. Most implementations also include the word `MS ( ms -- )` which waits for the given number of milliseconds, and gives a more repeatable delay time.

```
: WAIT   \ n -- ; wait for n iterations
  0 ?DO  PAUSE  LOOP
;
5000 value RATE1   \ -- ; delay counter
: ACT1ON1          \ -- ; An example task
  decimal          \ might need it somewhere
  console opvec !  \ output device
  console ipvec !  \ input device
  BEGIN            \ Start an endless loop
    7 EMIT         \ Produce a beep
    RATE1 WAIT     \ Reschedule so many times
  AGAIN            \ Go round again
;
TASK TASK1         \ name task, get space for it
```

The task name created by `TASK` is used as the task identifier by all words that control tasks.

9.2.3 Task dependent variables

An area of RAM is set aside for each task. This memory contains `USER` variables which contain task specific data. For example, `BASE` (holds the current number base) is normally a `USER` variable as it can vary from task to task.

A `USER` variable is defined in the form:

```
n USER <name>
```

where n is the nth byte in the user area. From version 6.1 onwards, the word `+USER` can be used to add a `USER` variable of a given size:

```
<size> +USER <name>
```

The use of +USER avoids any need to know the offset at which the variable starts. The kernel code relies on +USER and new application code should use +USER in preference to USER.

A USER variable is used in the same way as a normal variable. By stating its name, its address is placed on the stack. Data can then be fetched using @ and stored by ! in the usual way.

Local variables are held on the return stack, and so are intrinsically task safe. If heavy use of local variables is made, the required return stack depth can be large. If you suspect this of causing problems such as random crashes, increase the value of the EQUate for the return stack size in the control file.

9.2.4 Controlling tasks

Tasks can be controlled in the following ways:

- activated
- suspended for a number of schedules
- halted
- restarted after its been halted

You can also stop the current task.

To start a task, use the word INITIATE:

```
' <action> <task> INITIATE
```

where ' <action> gives the xt of the word to be run and <task> is the task identifier.

To temporarily stop a task, use HALT, which is used in the form:

```
<task> HALT
```

where <task> is the task to be stopped. To restart a stopped task, use RESTART, used in the form:

```
<task> RESTART
```

where <task> is the task to restart.

To stop the current task (i.e. stop itself) use STOP (--), used in the form:

```
STOP
```

9.3 Message handling

A useful feature of the multitasker is the ability to send and receive messages between tasks. We use a simple mailbox approach in which each message is a single cell of data whose meaning is entirely up to you. We have seen all the following uses of messages:

- Message numbers
- Pointer to a complex structure
- Xt of Forth word to execute

To send a message to another task, use `SEND-MESSAGE`, used in the form:

```
message task SEND-MESSAGE
```

where *message* is a single-cell message and *task* is the identifier of the task to send the message to. The message can be data, an address or any other type of information but its meaning must be known to the receiving task.

To receive a message, use `GET-MESSAGE`, which suspends the task until a message arrives. When a message is received the recieving task is restarted and the data is returned.

9.4 Event handling

Events are analogous to interrupts. Whereas interrupts happen on hardware signals, events happen under software control. Events are used to separate fast real-time processing from slower handling. For example an analogue-to-digital converter (ADC) may be run from a timer interrupt. To minimise time spent in the interrupt service routine (ISR), the ISR just puts the data in a buffer or queue, and sets the event flag in a task that processes the data. The next time round the scheduler loop in `PAUSE`, the task will execute the event handler before resuming its previous action.

An event handler is a normal Forth word with no overall stack effect `( -- )`. An event handler is associated to a task so that when the event is triggered, the task is activated. Therefore, an event cab used as a trigger for a task.

9.4.1 Initialising an event

Events are initialised in a similar way to tasks. They are assigned in the form,

```
ASSIGN EVENT1 task TO-EVENT
```

or

```
' EVENT1 task TO-EVENT
```

where `EVENT1` is your event handler and `task` is the associated task identifier.

9.4.2 Triggering an event

There are two ways of triggering an event.

`SET-EVENT ( task -- )` sets an event flag in a task. Once the event flag is set, the tasker will execute the event before it switches to the task. The task is also activated.

A bit can be set in a task's status word that indicates to the multitasker that an event has taken place. This method can be used to trigger an event from a hardware interrupt. Refer to 'The multitasker internals' later in the chapter for details of the status byte.

This mechanism is convenient in interrupt code written in assembler to signal that an interrupt has taken place, and that consequent processing should start.

9.4.3 Clearing an event

To acknowledge that an event handler has been run, use `CLEAR-EVENT`.

9.5 Critical sections and interrupts

Sometimes the multitasker has to be inhibited so that other tasks are not run during critical operations. These would otherwise cause the scheduler to operate, e.g. `KEY`. This achieved using the words `SINGLE` and `MULTI`.

```
SINGLE  -- ; inhibit tasker
MULTI   -- ; restart tasker
```

When communication between a task and an interrupt routine is required, or if the scheduler has been converted to be pre-emptive rather than cooperative, great care must be taken. Flags may be tested by the main task, interrupted and modified by the interrupt routine, and then written back by the main routine, causing the last interrupt change to be ignored. Six words are provided for interrupt management, and these are also documented in the interrupt chapter. There is considerable variation in CPU architectures, and if the words described here are not present, alternatives will be documented in the CPU specific code manual.

`code DI  \ --`
Globally disable interrupts.

`code EI  \ --`
Globally enable interrupts.

`code [I  \ -- ; R: -- x`
Save the current interrupt status on the return stack and globally disable interrupts. This word can only be used inside a colon definition and `[I` and `I]` must be used in matching pairs in the same word ... unless you **really** know what you are doing.
Restore the interrupt status from the return stack. This word can only be used inside a colon definition and `[I` and `I]` must be used in matching pairs.

If the target CPU has a maskable high priority interrupt, e.g. An NMI or ARM's FIQ, there may be additional words. See the CPU specific target code manual for the details.

`code SAVE-INT  \ -- x`
OBSOLETE: Return current interrupt state, and disable interrupts. This word is provided for compatibility with previous versions of the compiler and target code, but shorter and faster code is likely to be produced using `[I` and `I]`. This word will disappear in a future Forth 7.x release.

`code RESTORE-INT  \ x --`
OBSOLETE: Restore the interrupt state returned by `SAVE-INT`. This word is provided for compatibility with previous versions of the compiler and target code, but shorter and faster code is likely to be produced using `[I` and `I]`. This word will disappear in a future Forth 7.x release.

9.6 Semaphores

A `SEMAPHORE` is a structure used for signalling between tasks and for resource allocation. It has two fields, a counter (cell) and an owner (taskid, cell). The counter field is used as a count of the number of times the resource may be used, and the owner field contains the task identifier

of the task that last gained access. This field can be used for priority arbitration and deadlock detection/arbitration. An example compiler definition of SEMAPHORE is below.

```
Interpreter
: semaphore    \ -- ; -- addr [child]
  idata create
    1 ,  0 ,               \ count and arbiter fields
;
target
```

This design of a semaphore can be used either to lock a resource such as a comms channel or disc drive during access by one task, or as a counted semaphore controlling access to a buffer. In the second case the counter field contains the number of times the resource can be used.

Semaphores are accessed using SIGNAL and REQUEST. 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.

```
: signal        \ sem --
\ increment counter, so making it available
  [i                       \ must be interrupt safe
  1 over +!  cell+ off     \ inc. counter, release
  i]
;
```

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.

```
: request       \ sem --
\ Get access to semaphore
  begin
    [i  dup @ 0=           \ n.b test and set
   while
    i]  pause              \ operations must be
  repeat                   \ non-interruptible
  -1 over +!               \ got it, decrement counter
  self swap cell+ !        \ mark resource as mine
  i]                       \ re-enable interrupts
;
```

9.7 Multitasker internals

A multitasker tries to simulate many processors with just one. The multitasker works by rapidly

switching between tasks. On each task switch it saves the current state of the processor, and restores the state that the next task needs.

The Forth multitasker is software scheduled. This means that each task relinquishes control to the scheduler, which then switches to the next task. In this way less processor state information needs to be saved than for a preemptive scheduler.

9.7.1 Scheduler data structure

The Forth multitasker creates a task control block for each task. The task control block (TCB) is a data structure that contains information relevant to a task (see below). The status cell, `TCBST`, contains information on the execution of the task and its event (see below). The control block occupies the start of the `USER` area.

Field	Contents	Size	Offset
`TCB.LINK`	Pointer to next nexts TCB	Cell	0
`TCB.SSP`	Saved task stack pointer	Cell	2/4
`TCB.STATUS`	Task status	Cell	4/8
`TCB.MSRC`	Task ID of last message sent to this task	Cell	6/12
`TCB.MESG`	Message data	Cell	8/16
`TCB.EVENT`	XT of word run by tasks event handler	Cell	10/20

Table 9.1: Task control block

Bit	When set	When Reset
0	Task is running	Task is halted
1	Message pending but not read	No messages
2	Event triggered	No events
3	Event handler has been run	No events (reset by user)
4..	User defined	User defined

Table 9.2: Task status cell

9.8 Example Task

The following example is a simple demonstration of the multitasker. Its role is to display a hash (#) every so often, leaving the foreground Forth interpreter running. To use the multitasker you must cross-compile the file *MULTI*.FTH* into your target. The sample control files have an `EQU`ate `Tasking?` which, when non-zero, will compile the multitasker.

9.8.1 Defining the task

The following code defines a simple task called `TASK1`. It displays a # every 1000 schedules.

```
VARIABLE DELAY     \ time delay between #'s
  1000 DELAY !          \ initialise time delay
: ACTION1          \ -- ; task to display #'s
  CONSOLE OPVEC !          \ select output device
  [CHAR] $ EMIT            \ Display a dollar ($)
  BEGIN                    \ Start continuous loop
    [CHAR] # EMIT          \ Display a hash (#)
    DELAY @ 0              \ Reschedule Delay times
    ?DO  PAUSE  LOOP
  AGAIN                    \ Back to the start ...
;
```

9.8.2 Initialising the multitasker

Before any tasks can be activated, the multitasker must be initialised. This is done with the following code:

```
INIT-MULTI
```

The word `INIT-MULTI` initialises all the multitasker's data structures and starts multitasking. This word need only be executed once in a multitasking system.

9.8.3 Activating the task

To activate (run) the example task, type:

```
TASK TASK1
ASSIGN ACTION1 TASK1 INITIATE
```

This will set `ACTION1` as the action of task `TASK1`. Immediately you will see a dollar and a hash ($#) displayed. If you press <return> a few times, you see that the Forth interpreter is still running. After a few seconds another hash will appear. This is the example task working in the background. The repetition rate of the has symbol will depend on the performance of your CPU.

9.8.4 Controlling the task

The example task can be controlled in several ways:

- the rate of generation of ashes can be changed
- it can be halted
- once halted it can be restarted
- it can be started from scratch

Changing the variable `DELAY` will change the rate of production of hashes. Try:

```
2000 DELAY !
```

This changes the number of schedules that the example tasks makes between displaying hashes to 2000. The rate of displaying hashes halves.

Typing the task name followed by `HALT` halts the task:

```
TASK1 HALT
```

You notice that the hashes are not displayed any more.

The task is restarted by the word `RESTART`. Type the task name followed by `RESTART`:

```
TASK1 RESTART
```

The hashes are displayed again.

To restart the task from scratch, just kill it and start it again:

```
TASK1 TERMINATE
ASSIGN ACTION1 TASK1 INITIATE
```

The dollar and the hash ($#) are displayed, followed by hashes (#).

9.9 Troubleshooting tasks

The most common fault is a stack fault. Since a task is an endless loop it is simple to put stack depth checks in the main loop. A simple task with checking is shown below.

```
: TASK-ACTION
  sp@ s0 !                        \ store stack base
  <initialisation>
  BEGIN
    <body of task>
    depth                         \ non-zero if anything there
    IF
      s0 @ sp!
      <warn programmer!>
    ENDIF
  AGAIN
;
```

When using Umbilical Forth, the multitasker may need to be disabled by `SINGLE` before compiling new definitions interactively. If the multitasker is not disabled, the CPU is never put to sleep, and the act of compiling code through debug hardware may/will crash the running target.

9.10 Single chip tasking

Some of the smaller 8 bit CPUs, e.g. 8051, have a different memory model when used in single chip mode rather than with external RAM. For these and for CPUs with very limited internal RAM, there is a small version of the multi-tasker. event handling, messages, or semaphores. Details of this multitasker are provided in the CPU specific compiler manual.

9.11 Glossary

This glossary contains details of the major words in the multi-tasking system. Other words exist, but are only used as fractions of the words below.

`: CLR-EVENT-RUN \ --`
Clears the event run flag for the current task. This is bit 4 in the task status byte.

`code DI \ --`
Globally disable interrupts.

`code EI \ --`
Globally enable interrupts

`: EVENT? \ -- t/f`
Returns true if the event-triggered bit has been set in the current task's status byte.

`: GET-MESSAGE \ -- message task`
Waits for a message and returns the message and the sending task.

`: HALT \ task --`
Halts the task whose number is given. Do not halt task `MAIN`. Halting a task prevents it responding to messages or events.

`: INIT-MULTI \ --`
Initialises the multi-tasker and starts the multi-tasker. Just include this word in `COLD` to kick the multi-tasker into action.

`: INITIATE \ xt task --`
Initialises and starts the given task . Task `MAIN` is Forth itself and was activated when Forth started. Note that `INITIATE` causes the task to start from the very beginning. If the task was halted, and execution should resume where it left off, use `RESTART` instead.

`: MS \ ms -`
Waits for at least ms milliseconds, the exact time depending on the granularity of the timer.

`: MSG? \ task -- t/f`
Returns true if the task is holding a message, and is therefore not free to receive another one.

`: MULTI \ --`
Turns the multi-tasker on,.

`code PAUSE \ --`
Waits for one iteration of the scheduler.

`: RESTART \ task --`
Restarts a task that was halted by `HALT`. Unlike `INITATE`, the task resumes where it left off.

`code RESTORE-INT \ sr --`
Obsolete: Restore the interrupt enable state previously saved by `SAVE-INT`.

`code SAVE-INT \ -- sr`
Obsolete: Saves the current state of the interrupt enable, and disables interrupts. See `RESTORE-INT`.

`: SELF \ -- task`
Returns the task identifier for the current task. Useful with `MSG?` in particular to determine whether or not a message has been received by the task.

`: SEND-MESSAGE \ message task --`
Sends a message to the given task. The message address can be used on its own, or as a pointer to an extended message.

`: SINGLE \ --`
Turns off the multi-tasker.

`: STATUS \ -- n`
Returns the task status cell of the current task but with the running bit (bit 0) masked off. If this value is non-zero, the task has been awakened for a reason other than for normal running.

`: STOP \ --`
Halt the current task until it is `RESTART`ed or `TERMINATE`d.

`: TERMINATE \ task`
Remove a task from the list of active tasks and reschedule.

`: TO-EVENT \ cfa task --`
Sets the XT of a Forth word as the action to run when the task's event trigger is set.

```
ASSIGN <word> <task> TO-EVENT
```

`: WAIT-EVENT/MSG \ --`
The current task is suspended until it receives a message or an event trigger. The words `MSG?` and `EVENT?` can be used to determine whether a message or an event trigger terminated the wait. Note that if an event trigger is received, the event handler will have been called, and the event run flag (bit 4 in the status byte) will be set.

`code [I \ R: -- x`
Save the current interrupt status on the return stack and disable interrupts. This word can only be used inside a colon definition and `[I` and `I]` must be used in matching pairs in the same word.

`code I] \ R: ccr --`
Restore the interrupt status from the return stack. This word can only be used inside a colon definition and `[I` and `I]` must be used in matching pairs in the same word..

9.12 Converting to the v6.x multitasker

If your application was written before the new multitasker was released in the Forth 6 series, you are recommended to change to the new version.

9.12.1 Configuration

The new multitasker is configured by a different set of equates. The equate `#TASKS` was used to build a table of TCBs at compile time. This equate is replaced by `TASKING?` which only indicates that the multitasker is required. 1 equ tasking? \ true if multitasker needed 6 cells equ tcb-size \ internal consistency check 0 equ event-handler? \ true for event handler 0 equ message-handler? \ true for message handler 0 equ semaphores? \ true for semaphores

9.12.2 Task identifiers and TASK

The v6.x multitasker uses a linked list of tasks. Tasks are created by the defining word `TASK <name>` which allocates the resources needed. Execution of `<name>` returns the base address of the tasks `USER` area, and the task control information occupies the start of the user area. This address is referred to as a task identifier.

9.12.3 WAIT and MS

The word `WAIT` is not present in the v6.1 multitasker. It was mostly used to produce timed waits, and this function is now provided by the new word `MS`, which is supplied by the code in *DELAYS.FTH* and *TIMEBASE.FTH* or another timing system. `MS` waits for the specified number of milliseconds.

```
MS              \ ms --
```

9.12.4 INITIATE and ACTIVATE

The word `ACTIVATE` has been replaced by `INITIATE`, and `DEACTIVATE` has been replaced by `TERMINATE`.

```
INITIATE        \ xt task --
TERMINATE       \ task --
```

9.12.5 ?EVENT

The word `?EVENT` was hardly ever used in application code, and its action is now built into `PAUSE`.

10 Periodic Timers

This code provides a timer system that allows many timers 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. 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 10..100 ms. The source code is in the the file *TIMEBASE.FTH*.

The file *DELAYS.FTH* should be compiled after *TIMEBASE.FTH*. The code to start and stop the timebase system is part of the ticker interrupt system, which is compiled after *DELAYS.FTH*. If you need to write a new ticker interrupt handler, there will be examples to start from in the <*CPU*>*DRIVERS* folder. The required compilation order is this:

```
multitasker (optional)
TIMEBASE.FTH (optional)
DELAYS.FTH
Ticker driver
```

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, 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.

10.1 The basics of timers

These basic words are defined for applications to use the timer system. Other words are detailed elswhere 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 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.

10.2 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 use, either directly or indirectly.

The interrupt that handles all the timers does not set IPVEC and OPVEC to a default value. If you use 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.

10.3 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 one of the timer routines takes 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.

10.4 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.

```
: do-timers     \ --
```
Process all the timers in the chain

```
: 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 ; 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.

11 Time Delays

The code in *Common\Delays.fth* allows you to handle time delays specified in milliseconds. If you use the multitasker or *Common\Timebase.fth*, *Common\Delays.fth* should be compiled after them.

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

```
0 value ticks    \ -- n
```
Return current clock value in milliseconds. This value can treated as a 32 bit unsigned value that will wrap when it overflows.

```
: later          \ n -- n'
```
Generates the timebase value for termination in n millseconds 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. `TIMEDOUT?` does not call `PAUSE`, so it can be used in interrupt handlers. 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`.

12 Heap Memory Allocation

12.1 Heap definition

The heap is allocated from a predefined section of memory. Facilities are provided for user expansion of the heap to mass storage, although the current code makes no provision for page management. When the heap is initialised, a free block and an end block are created. The end block is of zero size, and is used only as a marker. The address returned by ALLOCATE and RESIZE is the address of the first data byte, as is the address consumed by FREE.

The heap MUST be initialised before use by calling INIT-HEAP. Heap access words return status=0 for success, and status<>0 for error.

Two equates are required during compilation to allocate a contiguous block of RAM for the heap.

```
STARTOFHEAP is the start address of the heap
SIZEOFHEAP is the size of the RAM for the heap
```

There are two versions of this code provided. HEAP32.FTH is provided for 32 bit targets and is optimised for the VFX code generator. HEAP16.FTH is for 16 bit targets, and is optimised for code density.

12.1.1 16 bit targets - HEAP16.FTH

The heap is controlled using two cells per block. This information is used in three parts:

```
cell = #bytes, number of bytes in this block
cell = flag, split between a four bit and a 12 bit field
```

The top four bits of the flag are used to indicate the block type, where $E = End, $F = Free, $A = Allocated. Others may be added later for type management.

The bottom 12 bits of the flag are currently unused, and should be set to zero.

12.1.2 32 bit targets - HEAP32.FTH

The heap is controlled using a single cell per block. This information is used in two parts:

```
bits 31..24: $EE - End, $FF - Free, $AA - Allocated
bits 23..0: 24 bits for number of data bytes in block.
```

A consequence of this is that the maximum block size that can be allocated is 16Mb-1 bytes.

If you use a pre-emptive scheduler or need to use the heap routines inside interrupt routines, you must define suitable heap lock and unlock routines and set the equate LOCKHEAP? to non-zero.

`LockHeap=0`
: no heap locking

`LockHeap=1`
: heap locking by turning off interrupts

`LockHeap=2`
: heap locking by semaphore.

12.2 Gotchas

The heap routines must be protected if they are to be used both in normal code and in interrupts. In this case the code must be modified to be interrupt safe, but this may have a significant impact on interrupt latency. Examples may be found in HEAP32.FTH.

12.3 Glossary

The glossary does not include all the factors used in the code. If you are interested in the implementation, please read the sources.

`: allocate \ #bytes -- addr status`
Attempt to allocate some memory from the heap. Walk the heap looking for a single big enough block. If the block is larger than than required split it into two blocks. Allocate part or all of the free block. Status=0 for success.

`: free \ address -- status`
Attempt to free a heap block. Status=0 for success. If *addr* is zero, no action is taken and zero is returned.

`: resize \ addr1 size -- addr2 ior`
Try to resize an allocated block to a new size, allowing for alignment. If the existing memory block is not big enough, the data will be copied to a new block, and the returned addr2 will not be the same as addr1. Status=0 for success.

`: init-heap \ -- ; initialise the heap structures`
The heap is initialised by creating 2 blocks. Block 1 starts at the beginning and is marked as a free block. Block 2 Is a null marker at the end of heap space.

12.4 Diagnostics

`: size \ addr -- currsize | -1`
Return the size of an allocated block or -1 if there's an error.

`: .heap \ -- ; display heap info`
Walk the heap displaying block information.

`: heapok? \ -- t/f ; check heap`
Walk the heap and return TRUE if the heap is "well".

13 Software Floating Point

13.1 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 double 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. Some words are avaliable during compilation of colon definitions, but not while interpreting.

13.2 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.

13.3 Entering floating-point numbers

Floating point number entry is enabled by `REALS` and disabled by `INTEGERS`.

Floating-point numbers of the form 0.1234e1 are required (see `FNUMBER?`) during interpretation and compilation of source code. Floating-point numbers are compiled as literal numbers when in a colon definition (compiling) and placed on the stack when outside a definition (interpreting).

The more flexible word `>FLOAT` accepts numbers in two forms, 1.234 and 0.1234e1. Both words are documented later in this chapter. See also the section on *Gotchas* later in this chapter.

Note also that MPE Forths use ',' as the double number indicator - it makes life much easier for Europeans.

13.4 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.

13.5 Creating and using variables

To create a variable, use `FVARIABLE`. `FVARIABLE` works in the same way as `VARIABLE`. For example, to create a floating-point variable called `VAR1` you code:

```
FVARIABLE VAR1
```

When `VAR1` is used, it returns the address of the floating-point number.

Two words are used to access floating-point variables, `F@` and `F!`. These are analogous to `@` and `!`.

13.6 Creating constants

To create a floating-point constant, use `FCONSTANT`, which is analogous to `CONSTANT`. For example, to generate a floating-point constant called `CON1` with a value of 1.234, you enter:

```
1.234e0 FCONSTANT CON1
```

When `CON1` is executed, it returns 1.234 on the Forth stack.

13.7 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.

13.7.1 Calculating sines, cosines and tangents

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

13.7.2 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.

13.7.3 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.

13.7.4 Calculating powers

Three power functions are supplied:

```
FE^X F10^X X^Y
```

13.8 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.

13.9 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.

13.10 Number formats, ANS and Forth200x

The ANS Forth standard specifies that floating point numbers must be entered in the form 1.234e5 and must contain a point '.' and 'e' or 'E', and that double integers are terminated by a point '.'.

This situation prevents the use of the standard conversion words in international applications because of the interchangable use of the '.' and ',' characters in numbers. Because of this, the cross-compiler's host VFX Forth uses two four-byte arrays, `FP-CHAR` and `DP-CHAR`, to hold the characters used as the floating point and double integer indicator characters. By default, `FP-CHAR` is initialised to '.' and `DP-CHAR` is initialised to to ',' and '.'. For strict ANS compliance, you should set them as follows **before** `CROSS-COMPILE` is run.

```
\ ANS standard setting
  char . dp-char !
  char . fp-char !
: ans-floats    \ -- ; for strict ANS compliance
  [char] . dp-char !
  [char] . fp-char !
;
\ MPE defaults
  char , dp-char !
  char . dp-char 1+ c!
  char . fp-char !
: mpe-floats    \ -- ; for existing and most legacy code
  [char] , dp-char !
  [char] . dp-char 1+ c!
  [char] . fp-char !
;
\ Legacy defaults, including ProForth
  char , dp-char !
  char . fp-char !
: legacy-floats \ -- ; for legacy code
  [char] , dp-char !
  [char] . fp-char !
;
```

You can of course set these arrays to hold any values which suit your application's language and locale. Note that integer conversion is always attempted before floating point conversion. This means that if the `FP-CHAR` and `DP-CHAR` arrays contain the same character, floating point numbers must contain 'e' or 'E'. If the arrays are all different, a number containing the `FP-CHAR` will be successfully converted as a floating point number, even if it does not contain 'e' or 'E'.

13.11 Glossary

13.11.1 Error Strings/Codes

These strings describe the various FP maths errors. The string address is

```
CREATE FP_FLN_ERR       \ -- addr
  ," Invalid argument to FLN/FLOG"
CREATE FP_FSQR_ERR      \ -- addr
  ," Square root of negative no.!"
CREATE FP_FE^X_ERR      \ -- addr
  ," Overflow in FE^X"
CREATE FP_F10^X_ERR     \ -- addr
  ," Overflow in f10^x"
CREATE FP_EX^Y_ERR      \ -- addr
  ," Result of FX^Y is complex"
CREATE FP_TRIG_ERR      \ -- addr
  ," Overflow in trig. function"
```

13.11.2 Separators

Before July 2010, the floating point separator, '.', was fixed. To ease internationalisation, it is now variable.

`variable fp-char \ -- addr`
Holds up to four character(s) to be treated as floating point indicators. Set to '.' for ANS compatibility. Note that this should be accessed as a one to four byte array. The first character is used as the point character for output.

`0 equ SepArray? \ -- flag`
If the equate is non-zero, `fp-char` is treated as a four byte array, otherwise as a one byte array. This is a flag for future expansion.

`: isSep? \ char addr -- flag`
Return true if *char* is one of the four bytes at *addr*. If less than than four bytes are needed, a zero byte acts as a terminator. Used when `SepArray?` is true.

`: isSep? c@ = ;`
A compiler macro used when `SepArray?` is false.

13.11.3 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`.

13.11.4 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 returns a floating point number.

`: FARRAY \ "<spaces>name" fn-1..f0 n -- ; Run: i -- ri`
Create an initialised array of floating point numbers. Use in the form:

```
fn-1 .. f1 f0 n FARRAY <name>
```

to create an array of n floating point numbers. When the array `name` is executed, the index i is used to return the address of the i'th 0 zero-based element in the array. For example:

```
4e0 3e0 2e0 1e0 0e0 5 FARRAY TEST
```

will set up an array of five elements. Note that the rightmost float (0e0) is element 0. Then `i TEST` will return the *\{i}th element. If you create this array in `IDATA`, restore `CDATA` afterwards.

`: FBUFF \ u "name" -- ; i -- addr`
Creates a buffer of u floats in the current memory section. The child action is to return the address of the ith element (zero-based).

```
10 fbuff foo
```

Creates an buffer for ten float elements in.

```
3 foo
```

Returns the address of element 3 in the buffer.

The default section is `CDATA`, and we recommend that you leave it that way! To create a ten element array in `UDATA` space, you can use:

```
udata
10 fbuff MyFloats
cdata
```

13.11.5 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.

`: S>F \ n -- fn`
Converts a single integer to a float.

`: 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.

`: D>F \ d -- fn`

Converts a double integer to a float.

```
: F>D           \ fn -- d
```
Converts a float to a double 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.

13.11.6 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.

13.11.7 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.

13.11.8 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.

13.11.9 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.

13.11.10 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.

```
: .FPsign       \ flag --
```
If flag is non-zero, generate a '-' otherwise a space. INTERNAL.

```
: .FPsep        \ --
```
Issue the FP separator, usually '.'. 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.xxxxEyy. Place the significand xxxxx at c-addr with a maximum of u digits. Return n the signed integer version of yy. Return flag1 true if f is negative, and return flag2 true if the results are valid. In this implementation all errors are handled by exceptions, and so flag2 is always true.

```
: F.            \ f --
```

Print the f.p. number in free format, xxxx.yyyy, if possible. Otherwise display using the x.xxxxEyy format.

13.11.11 Floating point input

Note that number conversion takes place in `PAD`.

`: 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 exponent 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. This word only accepts words with an 'E' as a floating point indicator, e.g, 1.2345e0. If BASE is not decimal all numbers are treated as integers. The integer prefixes '#','$','0x' etc. are recognised and cause integer conversion to be used.

`: >FLOAT \ c-addr u -- 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. This word accepts several forms, e.g. 1.2345e0, 1.2345, 12345 and converts them to a float. Note that double numbers (containing a ',') cannot be converted. Number conversion is decimal only, regardless of the current `BASE`.

`: (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.

13.11.12 Trigonmetric functions

N.B. All angles are in radians.

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

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

```
: FSQR          \ f1 -- f2 ; FSQR by Heron's formula
```
F2=sqrt(f1) by Heron's formula.

```
: 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).

13.11.13 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 Y and Y are both floats.

13.11.14 IEEE format conversion

`: FP>IEEE \ fp -- ieee32`
Convert native FP value to IEEE 32 bit format.

`: IEEE>FP \ ieee32 -- fp`
Convert IEEE 32 bit float to native format.

13.12 Gotchas

The ANS and Forth200x specifications define the format of floating point numbers during text interpretation as:

```
Convertible string := <significand><exponent>

<significand> := [<sign>]<digits>[.<digits0>]
<exponent>    := E[<sign>]<digits0>
<sign>        := { + | - }
<digits>      := <digit><digits0>
<digits0>     := <digit>*
<digit>       := { 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 }
```

This format is handled by the word `FNUMBER?`. The word `>FLOAT` accepts a more relaxed format.

```
Convertible string := <significand>[<exponent>]

<significand> := [<sign>]{<digits>[.<digits0>] | .<digits> }
<exponent>    := <marker><digits0>
<marker>      := {<e-form> | <sign-form>}
<e-form>      := <e-char>[<sign-form>]
<sign-form>   := { + | - }
<e-char>      := { D | d | E | e }
```

This restriction makes it difficult to use the text interpreter during program execution as it requires floating point numbers to contain 'D' or 'E' indicators, which is not profane practice. A quick kluge to fix this is to change `FNUMBER?` as below.

```
Replace:
  fcheck drop if                        \ valid f.p. number?
with:
  fcheck or if                          \ valid f.p. number?
```

Note that this change can/will cause problems if number base is not `DECIMAL`.

13.13 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.

13.13.1 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.

13.13.2 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.

13.14 High Level 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.

```
: <<1           \ n -- n<<1
```
A compiler synonym for `2*` or `1 LSHIFT`.

```
: >>1           \ n -- n>>1
```
A compiler synonym for `u2/` or `1 RSHIFT`.

```
: 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.

```
: <-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.

```
: d<<1          \ xd -- xd<<1
```
One bit double left shift.

```
: d>>1          \ xd -- xd>>1
```
One bit double right logical shift.

```
: D>>N          \ d n -- d>>n
```
N bit double right logical shift.

14 ROM PowerForth utilities

Supplied as source in the *Common\ROMFORTH* directory are utilities to:

- compile source code on your target board from AIDE
- upload a binary image from your target to your PC
- download a binary image to your target from your PC

Note that the target source code supplied with cross compiler versions 6.02 onwards is incompatible with code supplied for previous versions of the cross compiler.

These utilities can be used to generate an image in Flash that has all the tools required to develop an application, or can be used during development to transfer modules to and from your PC. All the code is designed to be used with the MPE development environment, AIDE. The code will also work with other compatible terminal emulators.

Users who wish to distribute devices or memory images containing the ROM PowerForth utilities should contact MPE for details of the OEM licence, which includes documentation on disc of the Forth kernel and the ROM PowerForth utilities.

14.1 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 and download the entire source if a small modification is made. The utilities permit text file to be split into pages for better layout when printed.

14.1.1 The required files

To compile text files from your target board, cross-compile the files *IODEF.FTH* and *TEXTFILE.FTH*.

14.1.2 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. AIDE's internal file server must be enabled (in the console window configuration), and will be triggered.

14.2 Downloading a binary image

A binary image can be downloaded from the target to your host PC. Two utilities are provided:

- Intel hex download
- XMODEM download

For both utilities AIDE or a suitable communications package will be required.

14.2.1 XMODEM binary image download

Binary images can be downloaded to your PC using the XMODEM protocol. To use this utility you must cross-compile the file XmodemTxRx.fth (called *BIN-DOWN.FTH* in some targets), which provides Xmodem transmit and receive functionality in both 128 byte and 1024 byte block formats. Reduced versions are available in *XmodemTxRx128.fth* and *MinXmodem.fth*. For more details see the common target code manual.

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 down-load 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.

14.2.2 Intel hex download

Binary images can be downloaded to your PC using the Intel hex format. To use this utility you must cross-compile the file *INTELHEX.FTH*.

To download a binary image from the target system to your PC, use `HEX-DOWN` in the form:

```
addr #bytes HEX-DOWN
```

where *addr* is the start address and `#bytes` is the number of bytes to down-load starting from addr. For example,

```
1200 400 HEX-DOWN
```

sends the area of memory from 1200 to 1599 to your host PC. In AIDE, turn on console logging to receive the file. In other packages this may be referred to as file capture.

14.3 ROM PowerForth

ROM PowerForth can be used to generate a stand-alone Forth system. With these utilities, you can generate a memory area that contains an interactive Forth with the ability to develop an application.

Note: A licence is required to distribute open Forth systems. Contact MPE for more details.

With current single-chip processors, the use of ROM PowerForth to generate turnkey applications has become more difficult, as custom compilation and Flash routines are probably needed. We are always available for this sort of consultancy!

14.3.1 Hardware requirements

To develop an application using ROM PowerForth, your board requires three memory areas, one of which:

- Is always Flash or EPROM. This area contains the development kernel.
- Is always RAM. This area is used for variables and all changeable data.
- Is RAM for development and Flash/EPROM for application. This area is used to develop your application. Therefore, it must be RAM while developing. Once the application is developed, the application's image must be saved into battery-backed RAM, Flash or EPROM. Therefore, this area must have the ability to be alterable but also non-volatile.

14.3.2 Types of board

The type of board that can be used to develop using ROM PowerForth is restricted to:

- three site boards
- two site boards with battery backed RAM
- two site boards with socket converter

Three site boards

The three areas are provided by three memory sockets:

- Flash/EPROM holding development kernel
- RAM which holds the variables and changeable data
- Flash/EPROM or RAM which is selectable by a link on the board

Two site boards with battery backed RAM

The three areas are provided by two sockets:

- EPROM holding the development kernel
- battery-backed RAM which is split into two areas

Two site boards with socket converter

On many boards, there is unused space in the Flash/EPROM as ROM PowerForth occupies less than 32k bytes of memory. A header board can be made which converts one socket into two. For example, if the socket normally takes a 64kb Flash/EPROM, a board can be made which has a 32kb Flash/EPROM with the ROM PowerForth development kernel and 32k bytes of RAM. To access the RAM, the write line is attached to a suitable point on the main board with a fly lead. After the application has been developed, the two images are combined back into a single Flash/EPROM.

14.3.3 Making your application turnkey

Once your application has been developed, it needs to be made turnkey so that it is always available. The application can be made semi-permanent by compiling into battery-backed RAM in the RAM/Flash/EPROM area. Alternatively, it can be copied into a Flash/EPROM if the board allows.

The word `SETUP` takes the address of the word passed to it and marks this in the RAM/Flash/EPROM header as the address of the word to be run at power-up. If a value of zero is passed to `SETUP`, the interactive Forth kernel will be run at power-up.

For example, the word `JOB` is to be run at power-up. Therefore you type,

```
' JOB SETUP
```

The application can be discarded by typing:

```
0 ROM !
```

The constant `ROM` returns the start address of the application RAM area. If the address of this area is to be changed, the Flash/EPROM must be modified. To do this, the 32-bit value in `ROM` must be changed.

14.4 AIDE file server protocols

AIDE's file server must be enabled for automatic file handling.

Details of the protocols used should be obtained from the source code in the ROMFORTH directory.

14.5 Glossary

`: BIN-DOWN \ addr len --`
Transmits a target image in XMODEM format to the host. AIDE can receive this file if the file server facilities are enabled.

`: CLS \ --`
Clears the display by sending a trigger character (code 3) to the host.

`: HEX-DOWN \ addr len --`
Transmits a target image in Intel Hex format to the host. The host can receive this file by enabling logging/capture.

`: INCLUDE \ "<name>" -- ; INCLUDE <name>`
Compiles from a specified text file <name> on the host AIDE file server. File loading can be nested.

15 Controlling compilation

The cross compiler is an executable program written in Forth and hosted on VFX Forth for the operating system. When the compiler starts, it behaves as a normal interactive Forth system with a number of extensions.

At this point, you can compile additional extensions if you need to. These can include memory drivers, tools to output the compiled image in a special format, and tools to talk to special debug hardware. In the vast majority of cases, the compiler already includes what you need.

After adding any extensions, the cross-compiler needs to be told about your target.

The whole process is usually controlled from a master file for your project. We call this the *control file*. Control files from MPE usually have a .ctl extension, e.g. *project.ctl*.

You need to tell the cross-compiler:

- when to start cross-compiling
- which code and data pages to compile into
- whether to align code to even/odd bytes
- whether to enable floating-point
- whether to turn the compiler log on or off
- when to compile portions of code selectively
- when to stop cross compiling

These instructions are normally placed in the control file, before any instructions are compiled.

15.1 Start and Stop compilation

To start cross-compiling, use the word `CROSS-COMPILE ( -- )`. At this point, the compiler "pulls down the shutters" and enters cross-compilation mode. Apart from compiler directives that are interpreted, code after this will be compiled into the target image instead of compiled onto the cross-compiler.

To mark the end of the cross-compilation phase, use `FINIS` for a standalone application or `UMBILICAL-FORTH` to start debugging an Umbilical Forth system. `FINIS` is used to finish cross-compilation completely, whereas `UMBILICAL-FORTH` is used to finish the batch portion of the compilation and to start the cross target link ready for interactive testing of an Umbilical Forth target.

15.2 Defining memory sections and xDATA

Regions of memory, known as **sections**, are defined in the control file by the `SECTION` directive. The cross-compiler treats memory as fitting into one of three types of memory; code, initialised data, and uninitialised data, and maintains a current section for each type.

15.2.1 Defining sections

The directive SECTION is used in the form:

```
start end type SECTION <name>
```

where *start* is the start address of the section, *end* is the last address in the section and *type* is one of CDATA, IDATA or UDATA, and <name> is the name of the section. By default, the section will be saved to disc with the filename <*name*>.*IMG*. The compiler automatically gives the filename <*name*> an extension *.IMG* so <name> should not include an extension. <Name> will then become the current section of that type in use. When a section name is executed, it becomes the current section of its type.

CDATA is used to define areas of memory that contain code, and the usually end up in Flash/ROM. IDATA is used to define areas of memory that are initialised at start up. When the cross-compiler finishes (the FINIS or UMBILICAL-FORTH directives), the used portions of all the IDATA sections are added to the end of the current CDATA section so that the target startup code can copy them into RAM. UDATA is used to define areas of memory that will not be initialised.

CDATA sections contain code and any data defined by CDATA during the cross-compilation.

IDATA sections contain any data defined by VARIABLE, VALUE or IDATA during the cross-compilation.

UDATA sections contain data allocated by RESERVE, BUFFER: or UDATA during the cross-compilation.

CDATA, IDATA and UDATA control which section type the following words apply to:

```
, ALIGN ALIGNED ALLOT C, CREATE HERE ORG UNUSED W,
```

After executing CDATA, IDATA or UDATA, the current section of that type is referenced by these words. After executing a section name, that section becomes the current one of its type, and that type is applied.

After defining all the memory sections for your target hardware it is good practice to explicitly select one of each type of section and to set the default memory type, normally CDATA.

Each section has four pointers:

- Start address
- Last adress
- Data pointer (DP) - current location, grows up from the start.
- Buffer pointer (BP) - grows down from the end. See RESERVE.

You define the first two when you declare the section. The third one (DP) is used by all section types. The fourth (BP) is used by RESERVE for special cases.

15.2.2 Section charateristics

By default, SECTION creates a buffer that is saved to disc when the compiler finishes. Other

directives can be used to select a different behaviour. All these directives apply to the current section.

Unless otherwise specified by `ALL-SAVED` below, the saved image will be from the start of the section to the section's current value of `HERE`.

`: WRITE-IGNORE  \ --`
Causes writes to the current section to be ignored, and reads always return 0.

`: WRITE-INVALID \ --`
Causes writes to the current section to generate an error, and reads always return 0.
For example, in a section of EEPROM, stores during cross-compilation would be meaningless, and can be trapped by using `WRITE-INVALID`.

```
$20000 $27FFF UDATA SECTION EEPROM  WRITE-INVALID
```

`: ALL-SAVED     \ --`
Save the whole section, not just the used part from the start. This directive is usually used for the primary `CDATA` section of CPUs whose reset vectors are at the top of memory, e.g. FreeScale 9S12s.

`: VIA-LINK      \ --`
This is used by Umbilical Forth to redirect sections to be accessed over the Umbilical link during the interactive session. For example, a target system may contain three sections: `ROM`, `IRAM` and `URAM`. The `ROM` section will already be in Flash or EPROM. When the interactive session starts, the user can type:

```
IRAM VIA-LINK
URAM VIA-LINK
ROM
```

So that the RAM areas are accessed across the Umbilical link.

`: IN-EMULATOR   \ offset --`
Causes the section memory to be in a Flash or EPROM emulator. The offset value is the offset from the start of the Flash at which the section starts. If paged memory is being used, each page will be in the emulator at a different offset. This means that the target can be reset as soon as the compilation has finished, without any intervening download process. Umbilical Forth especially benefits from this.

```
$00000 $07FFF CDATA SECTION ROM  0 IN-EMULATOR
```

Although hardware EPROM emulators were common some time ago, the huge variation in Flash device pin-outs and the increase of on-chip Flash has made them unused, except for people with deep pockets and true in-circuit emulators (ICE).

15.2.3 An example

```
$00000 $07FFF CDATA SECTION ROM      \ Main ROM area
$08000 $0FFFF IDATA SECTION IRAM     \ Initialised data
$10000 $1FFFF UDATA SECTION BBRAM    \ battery backed RAM
$20000 $27FFF UDATA SECTION EEPROM   \ EEPROM
$80000 $803FF UDATA SECTION DPRAM    \ dual port RAM
ROM IRAM BBRAM  CDATA                \ defaults
```

This example defines five areas of memory. With this setup, your kernel will have 32kb of ROM and 32kb for variables and interactive development, 64kb of uninitialised RAM which is not affected at power up, an EEPROM area, and a dual port RAM.

15.2.4 Section tools

`: ORIGIN \ -- addr`

Returns start address of the current CDATA section.

`: SEC-BASE \ -- addr`

Returns start address of current section.

`: SEC-TOP \ -- addr`

Returns end address of current section.

`: SEC-LEN \ -- u`

Returns length (size) of current section.

`: SEC-END \ -- addr`

Returns BP of current section.

`: UNUSED \ -- n`

Used to find out how much space is left in a section. If `UNUSED` returns a negative value, it indicates that the upper location counter, BP, (see RESERVE) is now lower than the normal location pointer, DP, and that you have a problem.

`: RESERVE \ len - addr`

Allocates down from top of `UDATA` section.

`: UNUSED \ -- n`

Returns the remaining available space in the current section. If this value becomes negative, you have overrun the available space.

`: .SECTIONS \ --`

Show section status.

`: [SECTIONS \ -- x1 x2 x3 x4`

Preserve current, `CDATA`, `IDATA` and `UDATA` sections to be restored by `SECTIONS]` below. Use in the form:

```
[SECTIONS  <change section>
  ...
SECTIONS]
```

`: SECTIONS] \ x1 x2 x3 x4 --`

Restore current, `CDATA`, `IDATA` and `UDATA` sections.

`: ,C \ x --`

Lay (comma) cell data into the current `CDATA` section.

`: W,C \ x16 --`

Lay (comma) 16 bit data into the current `CDATA` section.

`: C,C \ x8 --`

Lay (comma) 8 bit data into the current `CDATA` section.

`: ,I \ x --`

Lay (comma) cell bit data into the current `IDATA` section.

```
: W,I           \ x16 --
```

Lay (comma) 16 bit data into the current `IDATA` section.

```
: C,I           \ x8 --
```

Lay (comma) 8 bit data into the current `IDATA` section.

15.3 Bank switched systems

Bank-switched memory is mostly the preserve of CPUs limited to a 16 bit (64k byte) address range. Such systems provide additional memory by playing games with I/O ports or memory mapping hardware which allows parts of the memory map to be replaced under software control.

Today, the only justification for bank-switched hardware is to access a block oriented device such as a silicon disc or other data store.

Unless you are faced with a stable high-volume product whose hardware is frozen, try to avoid bank-switched systems. After implementing and observing many such systems, MPE's conclusion is that software development for bank-switched systems is ugly, slow and error-prone. If you have any influence on the hardware, make strenuous efforts to replace the CPU with a 32 device having a linear address range. As of now (2008), there is no cost justification for bank-switched systems.

That having been said, sometimes you just have to face it and do it!

15.3.1 Defining banks and pages

In bank switched systems `BANK`s may be defined, to which are attached `PAGES`. A bank defines the address range and type of switched memory, and multiple pages are defined within the bank. There is no limit to the number of separate banks and pages. Each page behaves as a `SECTION`, except that only the last referenced page in each bank is active. This allows us to bank switch both ROM and RAM areas.

Each page must have a unique identifier, restricted only in that zero can not be used as an identifier by the compiler. Otherwise the selection of page identifiers is entirely free, and can be chosen to ease the writing of the page handling words (see below).

```
HEX
0 7FFF CDATA SECTION ROM           \ 32k common ROM
8000 BFFF CDATA BANK ROMBANK       \ 16k pages of ROM
  0001 PAGES BANK0
  0002 PAGES BANK1
  0003 PAGES BANK2
C000 DFFF IDATA BANK IRAMBANK      \ 8k IDATA pages
  0101 PAGES IBANK0
  0102 PAGES IBANK1
  0104 PAGES IBANK2
E000 FFFF UDATA BANK URAMBANK      \ 8k UDATA pages
  0201 PAGES UBANK0
  0202 PAGES UBANK1
  0204 PAGES UBANK2
F000 F7FF IDATA SECTION SYSTEMRAM  \ 2k non-banked IRAM
F800 FFFF UDATA SECTION STACKRAM   \ 2k non-banked URAM
```

A very common configuration is to have a fixed ROM area to hold the Forth kernel and common application code, a bank switched ROM area for code expansion, a bank switched RAM area for data logging, and a non-switched RAM area for system variables and stacks.

In order to configure the system, you must provide two words, `PAGE@` and `PAGE!` which are used to find the current paging state and to set a new one. These words use the same page identifiers used by the `PAGES` directive.

```
PAGE@    \ -- page-id
PAGE!    \ page-id --
```

15.3.2 Flash layout control

When programming paged Flash, e.g. for a 68HC12/9S12 CPU, programming tools often require a physical base address in the Flash, rather than the 64k addresses used in the `SECTION` and `BANK` definitions. When a hex file is output, the base address of a section can be overridden using

```
physaddr SetFlashBase
```

immediately after the section is defined, e.g.

```
$8000 $BFFF cdata bank rombank   \ 16k bank in 64k address range
  $30 pages bank0
    $0C0000 SetFlashBase         \ where this bank is in the Flash
  $31 pages bank1
    $0C4000 SetFlashBase         \ where this bank is in the Flash
  $32 pages bank2
    $0C8000 SetFlashBase         \ where this bank is in the Flash
```

15.3.3 Executing words in another page

Execution of a word in another page is performed by the word `PAGE-EXECUTE`, which performs page selection and restoration for you. The high level version of this word is in the file *PAGING\PAGING.FTH*, which you should modify to suit your own hardware.

```
PAGE-EXECUTE   \ i*x xt pageid - j*x
```

When compiling code into pages, the compiler keeps track of the selected page, and if a reference is made to code in an unselected page, the compiler will generate the necessary page switch and restore code automatically. You cannot forward reference a word in another page.

15.3.4 Using CDATA pages

CDATA page management

`CDATA` pages are usually used with processors that do not have a large enough addressing range for the code that must run on them. There is an overhead in calling a word in another page because all such calls are made by `PAGE-EXECUTE`, which has to save and restore the current code page around the call. As a result, most users partition the code so that inter-page calls do not produce any significant performance overhead.

Multitasking and interrupts

Because all inter-page calls restore the previous page, the paging mechanism has no impact of on the multitasker unless `PAUSE` is used within a page. If any word that calls the scheduler is used in a page, the multitasker code should be modified to save and restore the page. You can use the code for `PAGE@` and `PAGE!` as a model.

Similarly if interrupt routines are in pages, the interrupt handlers must restore the previously active pages.

In many bank switched systems it is better to be safe than sorry and the simplest thing to do is to save the bank switch system state as part of the scheduler action and in interrupt handlers.

CDATA page vocabularies

The cross compiler treats `CDATA` pages as memory areas that have vocabularies. When a page is defined, the compiler creates a vocabulary of the same name as the page in the compiler.

When a page is referred to, the compiler performs the following actions:

- the page becomes the current code page in the bank.
- the vocabulary for the previously selected page in the same bank is removed from the search order.
- the vocabulary for the newly selected page becomes the top of the search order.

Consequently, you may need to use `ALSO` and `PREVIOUS` with page names in order to keep the Forth kernel in the search order. Assuming that the Forth kernel is all in the ROM section in the example above, the following code switches between the banks:

```
ONLY FORTH  ALSO BANK0 DEFINITIONS \ Use BANK0

BANK1 DEFINITIONS                  \ change to BANK1

BANK2 DEFINITIONS                  \ change to BANK2
```

Be aware that if you define vocabularies inside a CDATA page, you are responsible for removing them from the cross compilers search order before changing pages.

Because the cross compiler provides the interactivity for Umbilical Forth, this section also applies to interactive use of an Umbilical Forth system

Using CDATA pages interactively

This section discusses using vocabularies and pages interactively with a standalone Forth interpreter running on the target hardware. It is assumed that the reader understands the use of vocabularies.

When a banked CDATA page is defined, the compiler reserves two cells for page vocabulary links and some space in the current UDATA section. Any vocabularies defined in this bank will not be linked into the normal vocabulary chain, but into a chain anchored in the first cell of the page. As a result, switching between pages on a standalone target Forth does not affect the normal search order and the words in pages would be inaccessible even if heads were generated for them.

In order to provide interactive access to paged words, the compiler can be told to construct special vocabularies which automatically handle bank switching and the search order. Once all the memory sections have been defined to the compiler, the directive MAKE-PAGE-VOCS (used when the kernel is the active code page) causes the compiler to construct special vocabularies in the kernel. These vocabularies use the run time action of PAGE-VOCABULARY instead of VOCABULARY. The action of PAGE-VOCABULARY is as follows:

- Make itself the CONTEXT vocabulary
- Restore VOC-LINK to its initial value. This removes the previously selected code page from the search order.
- Select the required page as the current page in that bank.
- Add the pages own vocabularies (if any) to the VOC-LINK chain.

Note that MAKE-PAGE-VOCS must be used when the kernel page is the active code page. The data structure of a PAGE-VOCABULARY is the same as that of a normal VOCABULARY except that two more cells, containing the page identifier and page base address have been added to the CDATA portion of the vocabulary.

15.3.5 IDATA and UDATA pages

The action of IDATA and UDATA page selection is simply to make them the current page of their type. You can use these pages to expand the data area available to your application. For example, some embedded systems use bank switched data pages as mass storage. This is a typical way to use multi-megabyte memory cards in data loggers built around a processor with a restricted memory space.

Any routine that changes a current data page should be careful to restore it before calling the scheduler. As with CDATA pages the simplest thing to do is to save the bank switch system state as part of the scheduler action and in interrupt handlers.

15.3.6 Miscellaneous

N.B. 16 bit targets only: In order to ease development of paged systems, e.g. Freescale 9S12,

some additional directives have been provided to deal with addresses that need to be held in page:addr form, where the current page's bank identifier is in the upper 16 bits, and the 16 bit address is in the lower 16 bits.

`: PL:  \ -- ; -- page:chere ; PL: <name>`
Behaves like `L:` but returns a page:addr 32 bit address.

`: l>hilo  \ page:addr -- page addr`
Converts a paged address to separate items, addr on top.

`: l>lohi  \ page:addr -- addr page`
Converts a paged address to separate items, page on top.

15.4 Output file formats

Binary image files with a *.IMG* extension are always produced. You can change the default extension using the directive `setBinExt`, e.g.

```
setBinExt .bin
```

You can generate additional output formats for `CDATA` sections. An output format selector for the additional formats can be placed in the control file. They are:

`: NoHex  \ --`
Do not generate hex files (default).

`: HEX-I16  \ --`
Intel Hex for 64k bytes maximum as used for 8-bit CPUs. Generates *sectionname.hex*.

`: HEX-I32  \ --`
Intel Hex for 32 bit linear addresses, e.g. ARM. Generates *sectionname.hex*.

`: HEX-S19  \ --`
Motorola S19 format - 16 bit addresses. Generates *sectionname.s19*.

`: HEX-S28  \ --`
Motorola S28 format - 24 bit address range, e.g. 68HC12/9S12. Generates *sectionname.s28*.

`: HEX-S37  \ --`
Motorola S37 format - 32 bit address range, e.g. 683xx. Generates *sectionname.s37*.

`: ELF-format  \ machine flags --`
Select ELF output format using the given e_machine and e_flags values.
The initial execution address can be set for S28 and S37 formats by:

```
<addr> SetBoot
```

15.5 Aligning generated code

Many processors require XTs to start on even addresses, so that instructions start on an even address. To instruct the compiler to do this, use `ALIGN-EVEN`. Other processors require XTs to be 4-byte aligned. In this instance use `ALIGN-LONG`.

15.6 Numbers and 16 bit targets

This section only applies to 16 bit targets.

Double numbers and floating point numbers are converted to the format used by 16 bit targets. This means that the interpreted behaviour of double number operators may not give correct results. This conversion can be disabled and re-enabled by the directives HOST-MATH and TARGET-MATH. These is useful when calculating such things as baud rate divisors using EQUates defined in the control file.

```
HOST-MATH
  <perform calculation> EQU <equate-name>
TARGET-MATH
```

15.7 Enabling floating-point

If you want the compiler to be able to handle floating-point numbers, you need to instruct it with the word REALS. The default is integer only. Floating point can be disabled by INTEGERS. Note that for 16-bit targets, number formats are affected by the HOST-MATH and TARGET-MATH switches.

In MPE example control files, there is an EQUate SOFTFP? which should be set non-zero for compilation of the target floating point code.

```
1 equ SoftFP?
```

15.8 Turning the log on and off

The cross-compiler log can either display minimal information (when off) or information on the items compiled (when on). To turn the log on, use LOG. To turn the log off, use NO-LOG.

It is sometimes useful to sprinkle LOG and NO-LOG in a file when tracing obscure compilation errors or compile-time stack faults.

15.9 Conditional compilation

Conditional compilation is used to selectively compile portions of code. Four words are available to do this, [IF], [ELSE], [ENDIF] and [THEN]. These are analogous to IF, ELSE, ENDIF and THEN. They can be used within Forth words to selectively compile portions of code, or can be used outside a Forth word to selectively compile whole words.

15.9.1 An example

Two code examples are shown below. The examples given perform conditional compilation inside and outside a colon definition.

Conditional compilation while interpreting

The example shown below compiles one of the PRINT1OR2s. Which one is compiled is dependent on the value of 1OR2?. If it is set to one, PRINT1OR2 displays a one when executed. If it is set to two, PRINT1OR2 displays a two.

```
1 EQU 1OR2?
1OR2? 1 = [IF]        \ If 1OR2?=1, PRINT1 will be compiled
: PRINT1OR2   \ - ; Display a one
 ." 1"
;
[ELSE]                \ If 1OR2?=2, PRINT2 will be compiled
: PRINT1OR2   \ - ; Display a two
 ."2"
;
[THEN]                \ End marker for conditional compilation
```

Conditional compilation while compiling

Using conditional compilation within a colon definition is slightly more complicated. This is because you need to write a word which places a number on the cross-compiler's stack during cross-compiling. An example is shown below where an equate `3OR4?` is added to the compiler. This can then be used to control compilation.

```
3 EQU 3OR4?     \ add the word 3OR4? As an EQUate
: PRINT3OR4     \  ; Display a three or four
  [ 3OR4? 3 = ] [IF]            \ EQUate is interpreted
  [IF]
    ." 3"                       \ Display a three
  [ELSE]
    ." 4"                       \ Display a four
  [ENDIF]
;
```

15.9.2 [DEFINED] and [UNDEFINED]

The words `[DEFINED]` and `[UNDEFINED]` are used to find out if a particular word has already been defined, and return a flag. This is particularly useful when you want to keep a common body of code, yet provide for assembly language versions for slow processors. The following code allows a high-level version of a word to be defined if no previous version exists.

```
[UNDEFINED] <FOO> [IF]
: <FOO>

;
[THEN]
```

15.9.3 [REQUIRED]

This word is used by the library mechanism (see below). `[REQUIRED] <name>` returns true if `<name>` has been referenced but has not yet been defined. `<Name>` may be a word or a label.

```
[required] foo [if]
: foo  ;
[then]
```

15.10 Library files

When you need to keep code size to a minimum, the cross-compiler can resolve forward references by scanning library source files repeatedly until no more forward references can be resolved. This is done by defining a group of files that can be scanned. This should be done as the last action of the control file, although the compiler will permit scanning of library files anywhere. The log will show the number of passes made over the library files.

```
LIBRARIES
  all from-file <filename1>
  all from-file <filename2>

END-LIBS
```

Within each library file, the code is compiled normally, except that the word `[REQUIRED]` is used to control conditional compilation of each word in the file.

```
[REQUIRED] <name> [IF]
: <name>  ;
[THEN]
```

The code between `[IF]` and `[THEN]` will only be compiled if `<name>` has been forward referenced, i.e. it is required.

15.11 Loading binary data

The `DATA-FILE` directive loads a binary image file into target memory at `HERE` and reserves space for it, returning the size of the file. This is useful for adding data such as externally generated font tables and web pages. The file is loaded into the current section, so make sure to use `CDATA` or `IDATA` as appropriate. Macros in the file name are expanded but no default extension is assumed. For example:

```
cdata create image
  data-file %AppDir%\image.bin
  cr . ." bytes loaded"
```

15.12 Test code

The directives `TESTING [TEST` and `TEST]` support incorporating test code local to the definition that the code tests. The DOCGEN/SC extension can be used for safety critical systems to produce FDA (US Food and Drug Administration) standard documentation directly from the source code and to extract separate test files.

In order to allow test code to be built into the source code, conditional compilation of test code is provided, controlled by the word `TESTING`.

```
0 TESTING  \ test code will NOT be compiled (default)
1 TESTING  \ test code will be compiled
```

Test code should be surrounded by the markers `[TEST` and `TEST]`.

```
0 TESTING
[TEST
  This will all be ignored
TEST]
1 TESTING
[TEST
: foo .... ;
TEST]
```

In the first example all the code between `[TEST` and `TEST]` will be ignored. In the second case the code between `[TEST` and `TEST]` will be compiled.

15.13 C header files

In order to ease inclusion of the vast number of peripheral registers by name in modern microcontrollers, you can often cut and paste the definitions from C or assembler header files.

```
// - comment to end of line
/* comment N.B. white space delimited */
```

For `#DEFINE` note that the text up to the end of the line is evaluated once at compile time and produces an `EQU`ate of that single integer value.

15.14 Direct port access

If you are using Windows NT/2000/XP/Vista or any other version of Windows that treats direct port I/O as a privileged operation, and you want to drive ports directly, you must install the driver from the *NTPORT.EXE* file from the *COMPILER\XTRA* directory as described in the installation section of the manual. You must also modify your control file to include the `NT-ACCESS-PORTS` directive.

15.15 Split bootloader and application

To permit applications to be upgraded in the field, there are a number of systems that are split into a bootloader and and an application. At power up, the bootloader checks for a valid application and executes it if found. If there is no application or a magic token is used, the bootloader waits for a new application to be downloaded and flashed. In many cases the bootloader can replace itself. Such as system requires no connection between the bootloader and the application apart from some agreement on data layout.

As the complexity and features of the bootloader increase, it becomes larger. In turn, many of the hardware features used by the bootloader such as USB and TCP/IP, are also used by the application. Duplicating such ode reduces the code space for applications.

If we make some simple assumptions, much tighter integration of the bootloader and application can be achieved. In the the system here, the primary assumptions are:

1. If the bootloader changes, a new application matching the new bootloader must be used.

2. Application code can change if the bootloader remains the same.
3. The application is a continuation of the bootloader. The application is linked in if present.
4. Vocabulary/wordlist data in RAM is only in the first `IDATA` section defined.

The sequence is to cross compile the bootloader and application together. This allows the application to use any and all words from the bootloader, reducing overall size. Only sections for the bootloader are defined. the bootloader finishes with `BootFinis` instead of `Finis`. The bootloader's primary `CDATA` section is made read-only. The sections needed by the application are defined and compilation of the application proceeds. RAM needed by the application can be the same `IDATA` and `UDATA` sections as for the bootloader.

```
cross-compile
...         \ sections and code for the bootloader
bootfinis
...         \ sections and code for the application
appfinis
```

`BootFinis` generates initialisation data, saves the files and carries on. `AppFinis` generates more initialisation data, saves files and finishes cross compilation. The sequence is such that the bootloader target code is a separate application independent of the application code. It is assumed that you have designed the bootloader start up sequence so that it can detect the presence of a valid application after enough checks, e.g. CRC, that the application matches the bootloader.

The application start up code responsible for linking in the extended application dictionary and initialising the extended RAM areas. Example control files and target code are available from MPE, as is consultancy support.

`: BootFinis     cc/i    \ --`

Used in a split bootloader/application system to mark the end of the bootloader portion and the start of the application code.

`: AppFinis      cc/i    \ --`

Used in a split bootloader/application system to mark the end of the application code. Used in these systems instead of `finis`

`: flush-idata   cc/i    \ --`

Used with the bootloader portion of the code. If not already done, flush the primary vocabulary data to RAM and then copy the used portions of the `IDATA` sections to the current `CDATA` section. This directive is often used when the size of a binary file needs to extended to a certain alignment. The alignment code then follows after `flush-idata`. See also `lay-idata`.

```
cdata flush-idata            \ lay IDATA sections NOW
here $1FF +  $-0200 and org  \ force to 512 byte boundary
```

`: appFlush-idata        cc/i    \ --`

Used with the application portion of the code. As `flush-idata` above, but for the application portion of a split bootloader/application system.

```
cdata appflush-idata          \ lay IDATA sections NOW
here $1FF +  $-0200 and org   \ force to 512 byte boundary
```

16 VFX code generator

The VFX code generator is a black box that simply does its job of compiling and optimising your code, and usually no user intervention is required. Some implementations may have switches for special cases such as for dealing with loop alignment. These will be documented in the target specific section of the manual.

On a job using 60MHz LPC2000 ARM, we ran out of serial ports, and needed a few more running at 9600 baud and below. We wrote a bit-banged UART driver running from a 38400Hz timer interrupt and implemented two bidirectional serial ports. All the code was written in high level Forth. Using an additional I/O line to mark interrupt entry and exit, the worst-case interupt-handling time observed was 1.5 microseconds.

16.1 Inlining

The VFX code generator gives some control over the use of inlining, controlled by the word `INLINING (n -- )`. When the code generator has completed a word, the length of the word is stored in the symbol table. When the word is to be compiled, its length is compared against the value passed to `INLINING`, and if the length is less than the system value, the word is not referenced but is compiled inline, with the procedure entry and exit code removed. This avoids pipeline stalls, and is very useful for short definitions.

By default four constants are available for inlining control, although any number will be accepted by `INLINING`.

```
NO INLINING                \ 0, inlining turned off
NORMAL INLINING            \ 12-16, ~10% increase in size
AGGRESSIVE INLINING        \ 255, useful when time critical
ABSURD INLINING            \ 4096, unlikely to be useful
```

You can use `INLINING` anywhere in the code outside a definition. For very small words, a better technique is usually to define the word as a compiler macro:

```
compiler
: foo  ...  ;
target
```

This usually gives the optimiser more opportunities. Compiler macros are discussed in more detail later in this chapter.

Processors such as ARM and MIPS store the return address in a register. Inlining and the `+LEAFCALLS` or `ISLEAF` directives can interact to produce incorrect code. They have been interlocked to cause an error message if both are selected.

The following words are used immediately after a definition to control the inliner.

```
INLINE                      \ mark a CODE definition
INLINE-ALWAYS               \ will always be inlined
INLINE-NEVER                \ will never be inlined
```

16.2 Colon definitions

Any word that uses words that affect the return stack such as `EXIT`, or takes items off the return stack that you didn't put there in the same word, will automatically be marked as not being able to be inlined.

Everything that will cause inlining to fail causes inlining of the word to be disabled.

Note that when words are inlined, the effects may not be as expected.

```
: A  ;                  \ inlined
: B  A  ;              \ A inlined, B can be inlined
: C  B  B  ;          \ A, B inlined, C can be inlined
```

If you want to prevent a word ever being inlined, follow it with `INLINE-NEVER`. This is usally only necessary after you have done something particularly carnal in nature.

16.3 CODE definitions

By default `CODE` definitions are not marked for inlining because the assembler cannot detect all cases which may upset the return stack. If you want to make a code definition available for inlining, follow it with the word `INLINE`.

If you want the word to be inlined regardless of the state of `INLINING`, use `INLINE-ALWAYS`.

16.4 COMPILER directives

The VFX optimisers significantly reduce the need to code in assembler. However, some impact can be made by replacing very small definitions with compiler directives. Every time the VFX optimiser has to generate a call, it has to generate a canonical Forth stack. If you replace a short definition with a compiler directive, the optimiser does not call it, but compiles it as if from source code. Thus:

```
: foo       \ addr -- addr
  3 cells + @
;
```

can be replaced by

```
compiler
: foo       \ addr -- addr
  3 cells + @
;
target
```

On many target CPUs, especially those with good indexed addressing modes, the resulting code is shorter. `COMPILER` directives allow you to retain the code modularity of short Forth definitions without the calling overhead. You can explore this quite quickly, and the compiler section reports and file compilation reports will give you a good indication of whether you are winning. How much gain in code density you will get is often non-obvious, and the only way to get a feel for it is to play with the compiler.

17 Debugging tools

The tools described in this chapter can be used in interactive mode or during batch compilation.

17.1 INTERACTIVE mode

When `INTERACTIVE` is used after `CROSS-COMPILE` and before `FINIS`, the compiler will not exit after compilation finishes, but will enter an interactive mode in which the symbol table and image data are preserved. This allows you to use the other debugging tools with a standalone target compilation.

17.2 XDASM, DASM, DIS

Except for legacy compilers that use Indirect or Direct Threaded Code, Forth 7 compilers include a disassembler that can be used at any time.

```
XDASM <name>
DASM <name>
DIS <name>
```

will disassemble the word <name>.

17.3 LOCATE

When the compiler is active use the phrases:

```
LOCATE <name>
LOC <name>
```

to see the source code of word `<name>`. If you enter the compiler at the end of compilation, use the words `XLOCATE` or `XLOC` instead.

When running the compiler in AIDE, use the *IDE -> Configure* menu to define the editor in which the file is displayed. When running the compiler alone, use its *Options -> Set Editor* menu.

17.4 USES

When the compiler is active use the phrase

```
USES <name>
```

to see the words that use the word `<name>`. If you enter the compiler at the end of compilation, use `XUSES` instead.

17.5 XREF, XREF-ALL, XREF-UNUSED

The XREF cross reference system is turned on by +XREFS in the control file. All code after +XREFS will be cross referenced. Use XREFS to turn cross referencing off.

When the compiler is active use the phrase

```
XREF <name>
```

to see the words that use the word <name>. If you enter the compiler at the end of compilation with ESCAPE, use XUSES instead.

XREF-ALL produces a cross refence listing for the whole application. It is of most use when cut and pasted into a text editor for further processing.

XREF-UNUSED produces a list of the words that have not been referenced in colon definitions. XREF-UNUSED can be used to produce a minimum-sized application by removing those words that are unused.

17.6 WORDS

WORDS produces a list of the target words. The following switches control whether or not unresolved target words are shown by WORDS and friends:

```
  +SHOW-UNRESOLVED      \ --
 -SHOW-UNRESOLVED       \ -- ; default
```

17.7 .DWORD, .LWORD .HEX and .DEC

.DWORD, .LWORD and .HEX display an unsigned hexadecimal number:

```
  #123 .DWORD<cr> 0000.007B ok
```

.DEC displays a value as a signed decimal number:

```
  $55AA .dec<cr> 21930  ok
```

17.8 Lists

LABELS produces a list of the target labels.

EQUATES produces a list of the target equates.

Using the word ESCAPE in the control file before the final FINIS enters the cross compiler in host mode so that the debugging tools above can be used. Note that no files are saved. Unless you are debugging an extension to the cross compiler itself, the use of ESCAPE is now deprecated, and you should use INTERACTIVE above instead.

HELP lists the compiler directives, and gives some reminders.

COMPILERS lists all the words which are special when compiling.

INTERPRETERS lists all the words which are special when interpreting.

17.9 Command line switches

These switches can be used on the command line that runs the cross compiler to control its behaviour.

```
: /PAUSEOFF     \ -- ; run in batch mode
```
The compiler will terminate immediately after `FINIS` is used, otherwise it will offer you the choice of re-entering the compiler.

```
: /IDE          \ -- ; run from IDE host
```
When run from AIDE, this command tells the cross compiler to use the AIDE tool capture window as the console window.

```
: /PAGEOFF      \ -- ; inhibit page-throws
```
Prevents the compiler from putting page throw characters in the log.

```
: /COLS         \ cols -- ; log columns per line
```
Specifies the number of columns used in the log. By default the cross compiler will generate three columns, which allows 32 bit numbers to be logged as 8 hexadecimal digits.

```
: /+PAUSES      \ --
```
Enable pauses in console listings for tools such as */fo{WORDS} and `XREF` when more than a certain number of lines have been output. This is the default when the cross compiler is run with AIDE.

```
: /-PAUSES      \ --
```
Stop pauses in console for various listing tools.

18 Debugging Embedded Systems

The essential debugging tools for embedded systems are two LEDs (preferably different colours) driven by spare I/O lines. These pins should also be accessible by oscilloscope probes. You can probably debug anything using two LEDs and an oscilloscope, but it is certainly not the easiest way to do things.

If you can't talk to it, and it can't talk to you, you can't debug it. The commonest and easiest way to talk to an embedded system is over an RS232 link. Despite their disappearance from desktop PCs, real RS232 links are still the dominant way to talk to embedded systems. If your hardware does not have one, consider emulating a serial line using I2C or SPI. Even a pair of I/O lines can be programmed to behave like a serial port it's an interesting programming exercise in itself. For your PC, USB serial adapters are cheap and reliable these days - just use the transmit, receive and ground lines.

Now that we can talk to our hardware, we need to choose how to talk to it. Ideally, we have a Forth on the target hardware and we talk to that. We have to design our application so that it can be debugged. If there is enough RAM, run the Forth interpreter as a task so that it is always accessible. If you are using an Umbilical Forth, set it up so that the link runs in its own task. This way we can use the link as a normal Forth interpreter.

When we can use a Forth interpreter, we are freed from having to insert debug code, recompile, reprogram the Flash and watch. We can use the Forth interpreter to explore observations of the fault and the hypotheses we made from the observations. In combination with our two LEDs and the oscilloscope we can stimulate hardware and make measurements.

18.1 Basic rules

Fix bugs first. The more bugs you have in a system, the more difficult it is to find any of them. A bug is often left in because the original author could not find it. Years later it will make your life intolerable by masking the observation of your bug. Bugs have a habit of collecting in associated groups, often masked by the kluges you put in to get around them.

Crash early and crash often. It is a big temptation to use a lot of defensive programming to cope with earlier programming mistakes. Inside an operating system, coping with bad programmers is probably a must. In an embedded system, it just makes bugs harder to find. If your system faults because it writes to address 0, you will have to find the bug rather than leave it alone.

"*The primary duty of an exception handler is to get the error out of the lap of the programmer and into the surprised face of the user. Provided you keep this cardinal rule in mind, you cant go far wrong.*" *Verity Stob*

When the "surprised face of the user" is part of a bomb disposal machine or an anaesthetic ventilator you have a different class of responsibility. In turn, you can do a great deal to make life easier for yourself.

18.2 Make faults visible

You have carefully written an exception handler so that it dumps all the CPU registers and the stacks. You have even tested this at the console. Did you check that the output still comes out

on the console when a crash occurs in task using another serial line? Did you make sure that interrupt driven serial drivers get re-enabled inside an exception handler? Would it be safer to switch the console driver to polled inside an exception handler? Do you need to change the watchdog period?

Remember those LEDs? Now is probably a good time to turn on the red one. Maybe even flash it a few times and wait a bit before recovering the system.

18.3 Check tasks

Some of our colleagues do not test their code as well as we would like. One result is that we sometimes see stack faults in tasks. After the fault has occurred a system-dependent number of times, the stack overflows and the system crashes. The crash occurs some indeterminate time after the bug first occurred. We should make it visible as soon as possible.

If we use a simple house rule, we can systematically protect all our tasks. The house rule is that there is nothing on the stack at the head of a task's main loop. The example below is taken from USB mass storage code. We could also consider using one of our LEDs to check activity.

```
: doSectorRW    \ --
  consoleIO decimal
  ResetUSBdisk
  begin
    ?StackEmpty                          \ *** check stack
    WaitSecCmd  0 -> DriveComplete
    case DriveBusy?
      SecReadCmd of  Disk>USB  endof
      SecWriteCmd of  USB>Disk  endof
        SenseInvalidOp -> SenseCode
        DrvBadCmd -> DriveStatus
    endcase
    WaitDrvComplete
  again
;
```

The word `?StackEmpty` simply checks the stack depth and issues diagnostic messages to the console. The source code is in *Common\DebugTools.fth* or *Powernet\DebugTools.fth*.

18.4 Recover well

After you have taken suitable action fast enough not to compromise safety, you must recover from the disaster. Recovery in many systems can be as simple as rebooting, but may involve much more or much less, especially if there are safety implications. Rebooting most microcontrollers is achieved by setting up the watchdog to perform a hard reset. Oh whoops, the hardware reset line is input only and is not connected to the Ethernet PHY. Good recovery is often not as simple as it looks.

18.5 Talk to the hardware people

You cannot have the coloured LEDs unless the hardware people know you need them. This is

fine when you are the hardware people too, but not so fine when you really need that RS232 line and you don't have a soldering iron on site. What, you do not take a soldering iron and an oscilloscope with you everywhere?

To talk to dedicated hardware people, you must know enough hardware design to read circuit diagrams. This probably makes you dangerous in their eyes.

What you are trying to achieve is hardware that can itself be debugged, and contains facilities so that you can debug your own code.

18.6 Intepreting crash dumps

For CPUs such as ARM and Coldfire which provide separated fault and exception handling, the target code may provide support for crash dumps. The following discussion is for an ARM CPU. The ideas are the same for all CPUs.

The example below comes from typing

```
55 0 !
```

on the Forth console. The device is an NXP LPC2388 and address zero is Flash to which writes have not been permitted.

```
55 0 !
DAbort exception at PC = 0000:1C64
=== Registers ===
CPSR = 6000:001F
R0  = 0000:0037 0000:1C60 0000:0021 0000:0021
R4  = 0000:0070 0005:FD8C 4000:0298 0000:000C
R8  = 0000:1C60 0000:0000 0000:0000 4000:FEE0
R12 = 4000:FED4 4000:FDCC 0000:4FAC 0000:1C6C

RSP = 4000:FDCC   R0 = 4000:FDE0
--- Return stack high ---
4000:FDDC 0000:51E0 QUIT
4000:FDD8 4000:FED0
4000:FDD4 0000:0000
4000:FDD0 4000:FD94
4000:FDCC 0000:3244 CATCH
--- Return stack low ---

PSP = 4000:FED4   S0 = 4000:FED4
--- Data stack high ---
--- Data stack low ---
rTOS/R10  0000:0000
Restarting system ...
```

The exception occurred in the instruction at $1C64. It was a data abort exception, which means that it was a data load or store at an invalid address.

Assuming that restart is to a Forth console, you can find out where the fault occurred if you have compiled the file *Common\DebugTools.fth* or *Powernet\DebugTools.fth*.

```
$1C64 ip>nfa .name<Enter> !  ok
```

The return stack dump shows that `CATCH` was used, in turn called by `QUIT`, the text interpreter.

Further interpretation requires some knowledge of the use of the CPU registers.

18.6.1 ARM 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.

18.6.2 Interpreting the registers

Using the ARM example above, we can learn more.

```
DAbort exception at PC = 0000:1C64
=== Registers ===
CPSR = 6000:001F
R0  = 0000:0037 0000:1C60 0000:0021 0000:0021
R4  = 0000:0070 0005:FD8C 4000:0298 0000:000C
R8  = 0000:1C60 0000:0000 0000:0000 4000:FEE0
R12 = 4000:FED4 4000:FDCC 0000:4FAC 0000:1C6C
```

The exception occurred in the instruction at $1C64. It was a data abort exception, which means that it was a data load or store at an invalid address.

```
TOS = R10 = 0000:0000
UP  = R11 = 4000:FEE0
PSP = R12 = 4000:FED4
RSP = R13 = 4000:FDCC
```

In general, UP > PSP > RSP. In this case that's good. TOS=0, which we would expect from the phrase:

```
55 0 !
```

We now switch back to the cross compiler, which you did leave running, didn't you? Since we now know that $1C64 is in `!`, we can disassemble it.

```
dis !
!
( 0000:1C60 0100BCE8 ..<h )  ldmia r12 ! { r0 }
( 0000:1C64 00008AE5 ...e )  str r0, [ r10, # $00 ]
( 0000:1C68 0004BCE8 ..<h )  ldmia r12 ! { r10 }
( 0000:1C6C 0EF0A0E1 .p a )  mov PC, LR
16 bytes, 4 instructions.
 ok
```

From this, we can see that the offending instruction is

```
( 0000:1C64 00008AE5 ...e )  str r0, [ r10, # $00 ]
```

Since R10 is 0, we now know that it was attempting a write to 0, which is not permitted.

Provided that you keep words small, the register contents at the crash point, with the stack contents and the disassembly often provide enough information to reconstruct the state of stack on entry to the word.

19 Compilation in detail

This chapter provides more detail on how to get the best out of the compiler. Topics covered include:

- Special compilation behaviour
- Special interpretation behaviour
- Structures
- Defining words

19.1 Special compilation behaviour

Many ANS standard words are treated as special cases during compilation, either because the VFX code generator produces optimised code rather than a call to a target word, or because the word is normally `IMMEDIATE` and is executed during compilation.

The full list can be seen by typing:

```
COMPILERS
```

19.2 Special interpretation behaviour

Some words are only available during interpretation (outside a colon definition), or are treated specially during interpretation. The special behaviour may be required because:

- the words mimic target behaviour, usually by dealing with target memory,
- they are defining words,
- they are compiler directives,
- because they are made available for execution during interpretation.

The full list can be seen by typing:

```
INTERPRETERS
```

19.3 Structures

A named structure is defined using the following template. When the name of a structure is executed its size is returned.

If you have many fields of the same size, you can define your own field types.

```
Size FIELD-TYPE <field-type-name>
```

The template for a structure is:

```
STRUCT <struct-name>
  size1 FIELD <field-name1>
  size2 FIELD <field-name3>
  <field-type-name> <field-name3>
  ...
END-STRUCT
```

The run time action of a `<field-name>` is to add its offset in the structure to the address on the top of the stack. A structure can be used as a field within another structure by using the form:

```
<struct-name> FIELD <field-name>
```

The following example shows the construction of a structure defining a rectangle in terms of two points. The field type `INT` for a single-cell field is predefined.

```
STRUCT POINT    \ -- size
  INT .X              \ addr - addr
  INT .Y              \ addr - addr
END-STRUCT

STRUCT RECT     \ -- size
  POINT FIELD .TOP-LEFT      \ addr - addr
  POINT FIELD .BOTTOM-RIGHT  \ addr - addr
END-STRUCT

RECT BUFFER: NEW-RECT  \ -- addr ; in UDATA section

CREATE ANOTHER-RECT    \ -- addr
  RECT ALLOT           \ in current xDATA section
```

Note that `FIELD` and its children do not impose any alignment on the offset. If you are running with a processor that requires alignment, you must impose it yourself if the structure becomes misaligned, e.g.

```
struct foo
 1 field boo           \ 1 byte field
 aligned               \ forces alignment
 int poo               \ POO is now aligned
end-struct
```

Before a field is defined, the current size of the structure is on the top of the stack. Apart from using words such as `ALIGNED`, you can also perform other arithmetic.

19.4 Allocating memory and variables

This section shows the ANS definitions for each ANS word, and shows how to use them. These words are affected by the current `xDATA` setting, and unless otherwise noted refer to the currently selected data area which is one of `CDATA`, `IDATA` and `UDATA` which select which type of memory the Forth words below affect:

```
, ALIGN ALIGNED ALLOT C, CREATE HERE UNUSED W,
```

As is usual for descriptions taken from a standards document, the prose is turgid and legalese, but it's that way for a reason.

19.4.1 CREATE

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

Skip leading space delimiters. Parse name delimited by a semantics defined below. If the data-space pointer is not aligned, reserve enough data space to align it. The new data-space pointer defines name's data field. `CREATE` does not allocate data space in name's data field.

```
name Execution: ( -- a-addr )
```

a-addr is the address of name's data field. The execution semantics of `name` may be extended by using `DOES>`.

The result of this is to create a reference to the current location. Space can now be reserved using `ALLOT` or data can be laid down using one of the comma words. The example below contains a table of bit masks in the `CDATA` area.

```
CDATA CREATE BITS  \ -- addr ; table of bit masks
  8 C,                              \ size of table
  $01 C,  $02 C,  $04 C,  $08 C,
  $10 C,  $20 C,  $40 C,  $80 C,
```

`BITS` was defined with `CDATA` in effect, so the table is in code space, normally ROM, and is constant. If we had wanted to change this table, we could replace `CDATA` with `IDATA`, and then the table would be in RAM, but initialised at power up. If we just want to reserve an uninitialised area, we could use `UDATA` and `ALLOT`.

```
UDATA CREATE ABUFFER   \ -- addr
  <size> ALLOT
```

Note that it is either invalid or ignored to use the comma words in a `UDATA` section, or to write data to them at compile time. You cannot rely on the behaviour of the compiler under these circumstances.

19.4.2 Commas: , W, C,

These words lay data into the current `xDATA` section. `C,` lays a character (a byte in byte-addressed machines, or a cell in cell-addressed machines), `,` lays a cell, and `W,` lays a 16 bit value in byte-addressed machines. You can use these words as shown in the previous section to lay initialised data at compile time.

19.4.3 ALIGN and ALIGNED

The ANS specification provides these words to provide portability between systems that have different data alignment requirements. For example, a i386 does not require 32 bit data to be on a four byte address boundary. A 68332 requires it on a two byte boundary, and an ARM requires it on a four byte boundary. `ALIGN` forces the section to the next cell-aligned address, and `ALIGNED` will align an address on the stack.

```
: ALIGN         \ --
```

If the data-space pointer is not aligned, reserve enough space to align it.

```
: ALIGNED       \ addr - addr
```

A-addr is the first aligned address greater than or equal to addr.

19.4.4 ALLOT

ALLOT is used to reserve space in the current section. Note that, when used in IDATA space, the size of the initialised RAM table added by the compiler at the end of the ROM will be increased. See RESERVE and BUFFER:.

```
: ALLOT          \ n -
```

If n is greater than zero, reserve n address units of data space. If n is less than zero, release |n| address units of data space. If n is zero, leave the data-space pointer unchanged. If the data-space pointer is aligned and n is a multiple of the size of a cell when ALLOT begins execution, it will remain aligned when ALLOT finishes execution.
If the data-space pointer is character aligned and n is a multiple of the size of a character when ALLOT begins execution, it will remain character aligned when ALLOT finishes execution.

19.4.5 HERE (CHERE IHERE UHERE)

These words return the current data space pointer or that of the defined section in the case of the xHERE words.

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

Addr is the data-space pointer.

19.4.6 ORG (CORG IORG UORG)

ORG and friends set the the relevant data space pointer. In classical Forth, this is the variable DP, but does not have to be.

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

Set the data space pointer of the current section.
This directive is often used with HERE to place pieces of assembler code at specific locations, e.g. reset and interrupt entry points. If you use this technique, you may have to add the directive ALL-SAVED to the relevant SECTION declaration.

```
Proc EntryPoint
  ...
End-Code
CHERE  $FF00 CORG   \ code at the top of memory
AsmCode
  jmp  EntryPoint
  ...
End-Code
CORG                \ restore dictionary pointer
```

19.4.7 VALUE and VARIABLE

VALUE defines an initialised variable (size=cell) whose default action is to return its contents (value). To write to it, you must precede it with TO (ANS) or -> (MPE}. The address can be found using ADDR <value>. By definition, the data is in the current IDATA section.

VARIABLE defines a cell-sized variable that always returns its address. In Forth 7, the variable is in IDATA space and is initialised to zero. This prevents errors caused by forgetting to initialise

the variable before use. By legend, this error in a Fortran program was responsible for the loss of one of the Mars probes.

```
5 VALUE FOO
  FOO .  addr FOO @ .
6 to FOO  FOO .
VARIABLE BAR
  5 BAR !   BAR @ .
```

Skip leading space delimiters. Parse name delimited by a space. Create a definition for name with the execution semantics defined below. Reserve one cell of data space at an aligned address. `Name` is referred to as a variable.

```
name Execution: ( -- a-addr )
```

A-addr is the address of the reserved cell. A program is responsible for initializing the contents of the reserved cell.

Skip leading space delimiters. Parse name delimited by a space. Create a definition for `name` with the execution semantics defined below. Reserve two consecutive cells of data space. `Name` is referred to as a two-variable.

```
name Execution: ( -- a-addr )
```

A-addr is the address of the first (lowest address) cell of two consecutive cells in data space reserved by `2VARIABLE` when it defined name. A program is responsible for initializing the contents.

```
: VALUE         \ x "<spaces>name" -
```

Skip leading space delimiters. Parse name delimited by a space. Create a definition for `name` with the execution semantics defined below, with an initial value equal to x. `Name` is referred to as a value.

```
name Execution: ( -- x )
```

Place x on the stack. The value of x is that given when `name` was created, until the phrase `x TO name` is executed, causing a new value of x to be associated with `name`.

19.4.8 BUFFER: and RESERVE

`BUFFER:` is the equivalent, with one important exception, of the code below:

```
UDATA CREATE ABUFFER     \ -- uaddr
  <size> ALLOT

<size> BUFFER: ABUFFER   \ -- uaddr
```

The difference is that `BUFFER:` leaves the currently active section alone, whereas the first example switches it to `UDATA` which is a trap for the unwary.

Associated with `UDATA` sections is second location pointer, which grows down from the top of

the section, allocating space from the top. This can be very useful when careful use of the IDATA and UDATA spaces is required, as the gap between the top of the IDATA section and the bottom of the UDATA section can be made contiguous if the IDATA and UDATA sections are themselves contiguous.

```
: RESERVE       \ n - addr
```

RESERVE takes a required size n, drops the location pointer, and returns the base address addr. RESERVE is mostly used to reserve space for stacks and buffers in the form:

```
  <size> RESERVE EQU <name>
```

19.5 Local variables

The sequence

```
: <name>        { ni1 ni2 ... | lv1 lv2 ... -- o1 o2 }
  ...
;
```

defines named inputs, local variables, and outputs. The named inputs are automatically copied from the data stack on entry. Named inputs and local variables can be referenced by name within the word during compilation. The output names (after –) are dummies to allow a complete stack comment to be generated.

- The items between { and | are named inputs.
- The items between | and – are local values/variables.
- The items between – and } are outputs.

Named inputs and locals return their values when referenced, and must be preceded by -> or TO to perform a store, or by ADDR to return the address. Arrays may be defined in the form:

```
  arr[ n ]
```

Any name ending in the '[' character will be treated as an array, the expression up to the terminating ']' will be interpreted as the size. Arrays only return their base address, all operators are ignored. In the example below, *a* and *b* are named inputs, *a+b* and *a*b* are local variables, and *arr[* is a 10 byte array.

```
: foo           { a b | a+b a*b arr[ 10 ] -- }
  a b + -> a+b
  a b * -> a*b
  cr a+b .  a*b .
;
```

The ANS local variable syntax is also supported, but is not recommended on the grounds of readability and functionality. If you need it the ANS specification is provided in HTML format in the *DOCS\ANSFORTH* directory. Start with *DPANS.HTM*

19.6 Extending the compiler

The compiler allows the user to extend the compiler itself by controlling where new words are

placed. After cross-compilation is started, all new words are placed by default into the target image. The directives in the table below control where new words are placed.

It is a convenient conceptual model to regard these directives as corresponding to vocabularies called *TARGET, *COMPILER, *INTERPRETER, *ASSEMBLER and *HOST. The table shows the conceptual search order generated by the directives.

Directive and corresponding vocabulary	**Action**
TARGET *TARGET	New words are placed in the target image Conceptual search order: *TARGET
COMPILER *COMPILER	New words are added to the cross-compilers compile time behaviour. These words act like IMMEDIATE words in conventional Forth, but are not available during interpretation. All memory access words refer to the target. Conceptual search order: *COMPILER *HOST
INTERPRETER *INTERPRETER	New words are added to the cross-compilers interpret time behaviour. These words are not available during compilation. All memory access words refer to the target. See the next section on defining words for details of the actions for defining words using CREATE ... DOES> or CREATE ... ;CODE. Conceptual search order: *INTERPRETER *HOST
ASSEMBLER *ASSEMBLER	New words are added to the cross-compilers assembler. This directive is usually used to add macros to the assembler. Also searches the INTERPRETER words. Conceptual search order: *ASSEMBLER *INTERPRETER *HOST
HOST *HOST	Exposes the underlying host portion of the cross-compiler so that utility words can be added that will be used later by words defined using COMPILER INTERPRETER or ASSEMBLER. Use of this mode is at your own risk. Finish this mode with TARGET. Conceptual search order: *HOST

Table 19.1: Compiler extension directives

19.7 Defining words

Defining words can be handled in two ways, automatically by the cross-compiler, or explicitly using the extension mechanism discussed above. The objectives behind the two mechanisms are different.

The automatic mechanism aims to be transparent, so that code for the cross-compiler can be the same as that for a hosted Forth. This encourages portability and makes the cross-compiler easier

to use for the majority of defining words. The automatic mechanism copes with the majority of defining words.

The explicit mechanism provides very fine control of the host and target environments, but can be more confusing to use.

19.7.1 Automatic handling

The cross-compiler will automatically build an analogue of the defining word in the hosts conceptual `*INTERPRETER` vocabulary up to the terminating `;`, `DOES>` or `;CODE`. This is triggered by `CREATE`. Consequently, any code between the start of the word and `CREATE` will not have a host analogue. The words between `CREATE` and the terminating `DOES>` or `;CODE` must either be in the `*INTERPRETER` vocabulary or must be target constants or variables, which allows construction of linked lists that refer to target variables.

A target version of the defining portion up to `DOES>` or `;CODE` is built if the target words has heads. The run-time portion of the code is always placed in the target.

Construction of the host analogue is inhibited between the directives `TARGET-ONLY` and `HOST&TARGET`.

Both the defining words below can be handled automatically by the cross-compiler

```
: CON        \ n -- ; -- n ; a constant
  CREATE
    ,
  DOES>
    @
;

VARIABLE LINKIT      \ exists in target
: IN-CHAIN        \ n -- ; -- n ; constants linked in a chain
  CREATE
    ,                            \ lay down value
    HERE  LINKIT @ ,  LINKIT !   \ link to previous
  DOES>
    @
;
```

19.7.2 Explicit handling

Explicit handling uses the compiler directives discussed in a previous section. The explicit mechanism is particularly useful for more complex words, and where no target version of the defining word is required, as is often the case when the Umbilical Forth target is being used.

The examples from the automatic handling section are repeated here using the explicit mechanism.

```
INTERPRETER
: CON           \ n -- ; -- n ; a constant
  CREATE
    ,
  DOES>
    @
;
VARIABLE LINKIT \ exists in target
: IN-CHAIN      \ n -- ; -- n ; constants linked in a chain
  CREATE                        \ only in host
    ,                           \ lay down value
    HERE  LINKIT @ ,  LINKIT !  \ link to previous
  DOES>                         \ run time in target
    @
;
HOST
VARIABLE LINKIT2  \ exists in host
INTERPRETER
: IN-CHAIN2     \ n -- ; -- n ; link variable in host
  CREATE                        \ in host
    ,
    HERE  LINKIT2 @(H) ,  LINKIT2 !(H)
  DOES>
    @
;
TARGET
```

As can be see from the examples above, the automatic handling mechanism is simpler, but the explicit handling mechanism permits finer control over where code is generated, which can be useful when defining words are required and the absolute minimum of target memory is to be used.

19.8 IMMEDIATE words

As with defining words, `IMMEDIATE` words can be handled in two ways. In the first case, `I:` can be used to mark that a host analogue is required. In the second case, a host version of the word is placed in the `*COMPILER` conceptual vocabulary using the `COMPILER` directive. The examples below illustrate the definition of `IF`, which acts like IF but executes the code after `IF` if TOS=0.

19.8.1 Automatic handling

```
I: -IF            \ -- ; always produces target version
  POSTPONE 0=  POSTPONE IF
;  IMMEDIATE
```

The disadvantage of this method is that there will always be a target version, but the only variation from conventional Forth is the use of `I:`.

19.8.2 Explicit handling

```
COMPILER
: -IF              \ -- ; only exists in host
  0= IF                    \ references *COMPILERs 0= and IF
;
TARGET
```

19.9 Checksums

Checksums can be calculated over the current CDATA area. To do this, use the word CHECKSUM.

```
start end location type CHECKSUM
```

where *start* is the first address of the checksum region, *end* is the last address, and *location* is where the checksum is to be placed. The *type* is a constant identifying what sort of checksum is required, and may be chosen from the predefined types:

SIMPLE8	SIMPLE16	SIMPLE32	CCITT
CRC16	LRCC16	SDLC	CRC32
CRCxModem16	CRCxModem16-0		

19.10 Automatic build numbering

The automatic build numbering system allows you to update a build number string every time that a successful compile takes place. This information is stored in a file in the working directory. By default it is called BUILD.NO.

The build file consists of one line of text, which can be any mixture of text and numbers. At every update, all the digits in the text are treated as a single integer which is updated. This allows you to incorporate text in the form:

```
MPE PowerForth v6.40 [build 0030]
```

```
: BUILDFILE     \ "<filename>" -- ; set build file name
```

Sets the name of the build file, e.g.

```
BUILDFILE MYBUILD.NO
```

```
: MAKE-BUILD    \ addr --
```

Read the build file and copy the text to the target as a counted string. Use this to copy the string to a pre-allocated buffer.
Read the build file, and lay the text in the target as a counted string, e.g.

```
CREATE VERSION$  BUILD$,
```

BUILD$, only allocates the space needed to hold the string.

```
: UPDATE-BUILD  \ --
```

Update the build number file. Place this just before FINIS so that a successful build updates the build number.
The following three words can be used during interpretation to compile date and time strings into the target dictionary (as counted strings) to support build identification.

```
: DATETIME$,    \ --
```

Lay the compilation date and time as a counted string in the dictionary.

`: DATE$, \ --`
Lay the compilation date as a counted string in the dictionary.

`: TIME$, \ --`
Lay the compilation time as a counted string in the dictionary.

```
create BuildDate   \ -- addr
  date$, time$,              \ two strings
create DateTime    \ -- addr
  DateTime$,                 \ data and time as one string
```

19.11 Macros in text strings

The word `M",` is available during interpretation to lay down a counted string which includes macros delimited in the usual way by the % character, e.g.

```
CREATE DESCRIPTION  \ -- addr
  M", Reactor type %RTYPE%, boiler %BOILER%"
```

20 Target Forth model

This chapter describes how Forth is laid out on a target board. It is not necessary to read this chapter, but this chapter provides more information if you are interested or if you want to perform modifications to the cross-compiler or target.

20.1 Inside a ROM target Forth

A standalone ROM target Forth communicates with the host on a communications link, usually a serial line. The host needs to be running a terminal emulator, which displays any characters from the target and sends any characters typed at the host's keyboard. The target takes input from and sends output to the serial line, not from a keyboard and to a display. To do this, the generic I/O words `EMIT` and `KEY` use a generic I/O device that accesses the serial line.

20.2 Forth memory map

Apart from the code space itself, there are several important areas of RAM required by any Forth system. The RAM on the target system is split into several areas:

- `USER` area and stacks for eack task,
- `USER` area for high level interrupts,
- Terminal input buffer (TIB) for standalone Forth,
- Serial queues if required.

The remaining RAM is available for use by an interactive Forth as data and dictionary space.

Each task requires separate RAM for the two stacks and the `USER` area (thread-local storage). The three regions are contiguous. In some systems the `USER` area is at the bottom and stacks are at the top. For CPUs which support nested interrupts, the low region will be whichever stack is used for interrupt data, usually the CPU return stack.

If you write high-level Forth interrupt handlers, you can reserve RAM for interrupt handler `USER` areas and stacks.

Standalone Forths require a terminal input buffer to hold a line of text for processing.

If you are running your comms link at speeds above 38400 baud, you will probably need to use input queues on the serial lines. Running a serial line at 115200 baud makes development much more comfortable.

20.3 RAM initialisation

The ANS standard does not require variables (created by words `VARIABLE` or `CVARIABLE`) to be initialised at start up. In MPE PowerForth data created in `IDATA` space is initialised to zero within the cross-compiler. The table of initial values is then copied to the end of the output file when the cross-compiler finishes. The compiler termination report tells you where the table is located.

Two locations (defined as labels) in the target, `INIT-RAM` and `RAM-START`, point to the initial

value table (in ROM), and to the memory area (in RAM) it should be copied to. The table consists of a number of entries containing four fields: len, addr, pageid, len data. This repeats until terminated by an entry with len=0.
Cell: len, a count of the number of bytes to be copied
Cell: addr, the address to which the data should be copied
Cell: pageid, the page id in which the data resides, 0 indicating unpaged memory.
Len bytes: the data to be copied.

The code that performs this copy is in the word `(INIT)` in *COMMON\KERNEL62.FTH*.

In addition to using the memory store operators `C!`, `W!` and `!`, RAM may be initialised when space is allotted using cross-compiler words that use `,`, `W,` or `C,`. It is safest to explicitly initialise all variables and data areas in `COLD` or `ABORT`. This protects the system from errant behaviour after error recovery or power failure. It is worth remembering that a Mariner probe was lost because of an uninitialised Fortran variable!

For Umbilical Forth targets, an `EQU`ate `INIT-IDATA?` may be present to control whether the additional start up code to perform initialisation is compiled. This saves code space when initialisation of `IDATA` space is not required.

20.4 Implementation model

The Forth implementation on modern CPUs and 32 bit targets uses subroutine threaded code and the VFX code generator. On entry to and exit from words, the top of the data stack is kept in a register. Other registers are used for the two stack pointers, the pointer to the `USER` area, and a local variable frame pointer. The assignment of the registers is given in the assembler chapter of the CPU specific manual.

Most targets includes a very simple non-optimising compiler. The VFX code geneator is quite big (5000 lines of code), and so is reserved for the cross compiler and for Forth systems hosted by an operating system. If you want to port the VFX code geneator to a target system, please contact MPE.

Some compilers for older targets will maintain compatibility with Forth 5 and will generate Direct Threaded Code (DTC).

20.5 Forth models

Two different targets are provided in the `COMMON` directory. The first is a standalone Forth that can be debugged interactively using a dumb terminal. The Forth provides all the facilities you need. Source code can be downloaded to the Forth and debugged on the target. The target Forth provides interpretation and compilation facilities.

The second is a Forth called Umbilical Forth that is tuned for single chip applications. Unlike the Standalone Forth, Umbilical Forth requires the Umbilical Forth message passer in the *TARGEND.FTH* file for interpretation and compilation, which is provided by a server on the host PC (see below). The Umbilical Forth kernel is typically less than 4k bytes for 32 bit targets, or 2k bytes for 16 bit targets. These figures will vary between different processors. One of our managed to get below 512 bytes on an 8-bit CPU!

All directories use the same implementation model, and so code from one system can be used

by another. Thus an application using Umbilical Forth as a basis can safely use code from the stand-alone Forth. This does not apply on some processors such as the 8051, where stacks may be in different address spaces in the stand-alone and Umbilical models. In this case there may be a separate set of UMB files that match the ROM model. Note that all the Umbilical Forth message handling source code is in high-level Forth.

20.6 Inside Umbilical Forth

Umbilical Forth interacts with you in the same way as a ROM target Forth, but the mechanism that provides the interaction with the target is totally different. When you reset the target and the board signs-on, you are still running the cross-compiler. Umbilical Forth is therefore an extension of the cross-compiler to provide interactive cross interpretation and cross-compilation.

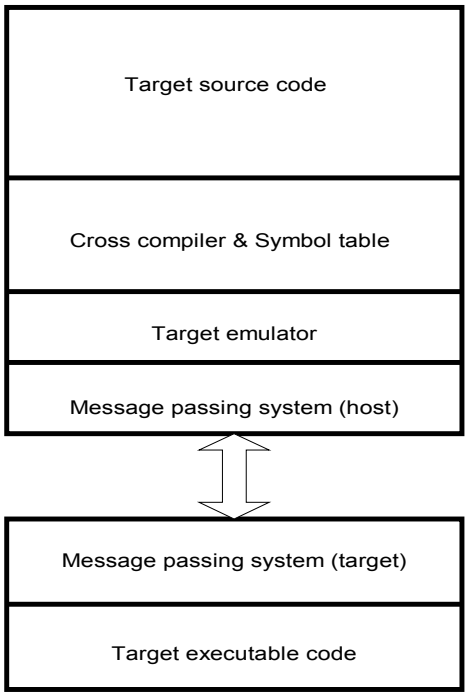


Figure 20.1: Umbilical Forth structure

When a word is cross-compiled, the cross-compiler places information in the symbol table. The symbol table therefore contains the XT of the word in the target image. By using a message-passing system between the cross-compiler and the target, the XT of the word can be passed to the target. The target can then execute the word on the target passing parameters to and from as appropriate. Therefore, the target does not need any headers in the target image, nor does the target need any of the code to process the headers.

21 Example control file

The example control file presented here is typical. It is for the MPE ARM Development Kit hardware. Your control file will be different, but the code is commented to show what is important.

21.1 Standard header

The header section contains the copyright notices and a description of the target. It also contains the change history for the system.

```
\ Builds a PowerNet system for the MPE ARM7 Development Kit.

((
Copyright (c) 2003
MicroProcessor Engineering
133 Hill Lane
Southampton SO15 5AF
England

tel: +44 (0)23 8063 1441
fax: +44 (0)23 8033 9691
net: mpe@mpeforth.com
     tech-support@mpeforth.com
web: www.mpeforth.com

From North America, our telephone and fax numbers are:
  011 44 23 8063 1441
  011 44 23 8033 9691

The code is set up to run in a 48k section of Flash
  $1000000 $100BFFF
The boot code remaps the chip select unit and segment mapper to put:
  1Mb Flash     at 0100:0000 to 010F:FFFF Segment 1, r/w, not cached
  512k RAM      at 0000:0000 to 0007:FFFF Segment 2, r/w, cached
  2k local SRAM at 0000:0000 to 6000:07FF Cache mode
  1k Ethernet   at 5000:0000 to 5000:03FF Segment 4, r/w, not cached
The vector table is then copied to address 0.

To do
=====

Change history
==============
))
```

21.2 Text macros

This section handles defining the directory structure of the kernel and application. You can modify this if the directories are moved, and you can also use conditional compilation if you have a different directory structure on your desktop and your laptop.

```
only forth definitions

\ ****************************
\ Define the default directories
\ ****************************

"" ..\common"   setmacro CommonDir  \ where common code lives
"" ."           setmacro CpuDir     \ where CPU specific code lives
"" .\hardware\MpeArmDevKit"
                setmacro HWDir      \ board specific code lives
"" c:\buildkit.dev\software\AddOns\PowerNet\v30dev"
                setmacro IpStack    \ where PowerNet code lives
"" ..\examples\Filesys"
                setmacro FileSysDir   \ where the File System lives
```

21.3 Cross compiler initialisation

Until the word `CROSS-COMPILE` has been run, this is a normal Forth system and the facilities of the host Forth can be accessed. After this, the system is reconfigured as a cross compiler. Because of this, extensions such as macros are compiled before `CROSS-COMPILE`.

This section may include some CPU specific directives. These will be documented in the CPU specific manual. In this case, the ARM version and alignment are specified.

```
\ ********************************************************
\ Turn on the cross compiler and define CPU and log options
\ ********************************************************

include %CpuDir%\macros         \ compiler and assembler macros

\ file: PROG.log                \ uncomment to send log to a file

CROSS-COMPILE

only forth definitions          \ default search order

  no-log                        \ uncomment to suppress output log
  rommed                        \ split ROM/RAM target
  interactive                   \ enter interactive mode at end
  +xrefs                        \ enable cross references
  align-long                    \ code is 32bit aligned
  ARM7                          \ Core of Sharp's LH77790
  32bit-mode                    \ running in 32 bit mode

0         equ false
false not equ true
```

21.4 Configure target

The target has to be configured for memory layout, size of stacks and user areas and so on.

```
\ ****************
\ Configure target
\ ****************

\ What sort of header do we need, default is memory image with no header
 0 equ AIF?                \ true for ARM AIF format

\ Kernel components
 1 equ tasking?            \ true if multitasker needed
   6 cells equ tcb-size    \   for internal consistency check
   0 equ event-handler?    \   true to include event handler
   0 equ message-handler?  \   true to include message handler
   1 equ semaphores?       \ true to include semaphores
 1 equ timebase?           \ true for TIMEBASE code
 0 equ softfp?             \ true for software floating point
 0 equ FullCase?           \ include ?OF END-CASE NEXTCASE extensions
 0 equ target-locals?      \ true if target local variable sources needed
 0 equ romforth?           \ true for ROMForth handler
 0 equ blocks?             \ true if BLOCK needed
 $20000 equ sizeofheap     \ 0=no heap, nz=size of heap
   1 equ heap-diags?       \   true to include diagnostic code
 0 equ paged?              \ true if ROM or RAM is paged/banked
 0 equ MPE-SET?            \ compatibility with MPE v5 targets
 0 equ ENVIRONMENT?        \ true if ANS ENVIRONMENT system required
 0 equ ColdChain?              \ true if cold chain system needed.

\ Clock, serial and ticker rates
#24000000 equ system-speed      \ System clock rate in HZ.
#38400 equ console-speed        \ Serial port speed in BPS.
#38400 equ console0-speed       \ Serial port 0 speed in BPS.
#38400 equ console1-speed       \ Serial port 1 speed in BPS.
#38400 equ console2-speed       \ Serial port 2 speed in BPS.
2 equ console-port         \ Designate serial port for terminal.
#10 equ tick-ms            \ TIMEBASE tick in ms

\ version numbers
 char 6 equ mpe-rel        \ x in Vx.yz
 char 1 equ mpe-ver        \ y in Vx.yz
 char 0 equ usrver         \ z in Vx.yz

\ define stack and user area sizes
$0200 equ UP-SIZE          \ size of each task's user area
$0200 equ SP-SIZE          \ size of each task's data stack
$0200 equ RP-SIZE          \ size of each task's return stack
up-size rp-size + sp-size +
  equ task-size            \ size of TASK data area
UP-SIZE equ INTRAM         \ space used by interrupt page

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

\ define nesting levels for interrupts and SWIs
1 equ #IRQs               \ number of IRQ stacks,
```

```
                          \ shared by all IRQs (1 min)
0 equ #SWIs               \ number of SWI nestings permitted (0 is ok)

\ *****************
\ default constants
\ *****************

cell equ cell             \ size of a cell (16 bits)
0 equ false
-1 equ true

\ ********************
\ Define memory layout
\ ********************

$00000000 equ link-address  \ for a binary image
                            \ - usually starts at zero on the ARM
                            \ Used by the AIF header
$00000000 $0001FFFF cdata section ADKnet    \ 128k program
$01000000 $010FFFFF cdata section PROGf     \ 1Mb of Flash
$00020000 $0002FFFF idata section PROGd     \ 64k IDATA RAM
$00030000 $0006FFFF udata section PROGu     \ 256k UDATA RAM
$00070000 $007FFFFF udata section VideoRAM  \ 64k video RAM
\ N.B. Change INITNET.FTH if you change this.

Interpreter
: prog adknet ;             \ synonym for common code
target

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


\ *********************************
\ USER area and Multi tasker equates
\ *********************************
\ Assume stacks grow down: user area, sp stack, rp-stack
\ Main User/Task stack for USR/SVC operation
\ The return stack must be the lowest of RSP, PSP and UP
\ in order to permit fast interrupt nesting. In order for
\ the initialisation code in MULTIARM.FTH to work, INIT-U0
\ must be the highest.

rp-size sp-size + equ TASK-U0      \ initial offset of user area
rp-size sp-size + equ TASK-S0      \ initial offset of data stack
rp-size           equ TASK-R0      \ initial offset of return stack

task-size reserve equ INIT-T0      \ base of main task area
  init-t0 task-u0 + equ INIT-U0    \ base of main user area
  init-t0 task-s0 + equ INIT-S0    \ top of main data stack
  init-t0 task-r0 + equ INIT-R0    \ top of main return stack

task-size #SWIs * reserve drop     \ space for SWI nesting
```

```
tib-len reserve equ INIT-TIB       \ base of TIB

\ IRQ stacks ; nestable up to #IRQs
0 reserve equ IRQ_STACK_TOP        \ top of IRQ stacks
task-size #IRQs * reserve          \ bottom of IRQ stacks
  equ IRQ_STACK_BASE

PROGd
  sec-top 1+ equ UNUSED-TOP        \ top of memory for UNUSED
PROG
```

21.5 Kernel files

This section uses the information defined earlier to pull in the required files for the Forth kernel.

```
\ ************
\ Kernel files
\ ************

  include %CpuDir%\sfr790A         \ LH77790A Special function registers.
  include %CpuDir%\initARM         \ Generic startup code (*required*).
  include %HwDir%\Boot\InitNet     \ Devkit start up code for boot loader
  include %CpuDir%\codeARM         \ low level kernel definitions
  include %CommonDir%\kernel62     \ high level kernel definitions
  include %CpuDir%\intARM          \ exception handlers
  include %CpuDir%\drivers\Ser790i \ Debug Uart - channel 2
  include %CommonDir%\devtools     \ DUMP .S etc development tools
  include %CommonDir%\voctools     \ ORDER VOCS etc
  include %CommonDir%\methods      \ target support for methods
  include %CpuDir%\local           \ local variables

tasking? [if]
  include %CpuDir%\multiARM        \ multi-tasker, MUST be before TIMEBASE
[ELSE]
  : pause ;
[then]

timebase? [if]
  include %CommonDir%\timebase     \ time base common code
  include %CpuDir%\drivers\Tick790 \ timer tick
[then]

environment? [if]
  include %CommonDir%\environ      \ ENVIRONMENT?
[then]

SIZEOFHEAP [if]
  include %CommonDir%\heap32       \ memory allocation set
[then]
```

```
softfp? [if]
  include %CpuDir%\softfp                \ floating point
  include %CommonDir%\softcom       \ common floating point code
[then]

romforth? [if]
  include %CommonDir%\RomForth\link         \ appl. rom link
  include %CommonDir%\RomForth\iodef        \ link i/o
  include %CommonDir%\RomForth\filetran     \ ascii file uploader
  include %CommonDir%\RomForth\xmodem       \ XMODEM downloader
  include %CommonDir%\RomForth\intelhex     \ Intel Hex downloader
  include %CommonDir%\RomForth\textfile     \ XSHELL textfile support
\ include %CommonDir%\RomForth\blocks       \ XSHELL blocks support
[then]

mpe-set? [if]
  include %CpuDir%\mpe_supp                 \ MPE v5 compatibility word set
[then]

\ *************
\ End of kernel
\ *************

internal
: .CPU          \ -- ; display CPU type
  ." MPE ARM ANS ROM PowerForth v6.20"
;
external

: ANS-FORTH     \ -- ; marker
;
```

21.6 Application code

The application code example here is MPEs PowerNet TCP/IP stack, which uses its own build file, but requires configuration through a number of equates and some compiler and interpreter extensions.

```
\ ************************
\ Add application code here
\ ************************

interpreter
: const equ  ;
\ Define this as CONSTANT to get interactive access to the
\ constants.
Target

ProgF
  sec-base equ Flashbase
Prog
```

```
compiler
: ForceUncached  ;      \ addr -- addr'
target
interpreter
: ForceUncached  ;      \ addr -- addr'
target
   include %CpuDir%\drivers\29F040B.fth

create EtherAddress     \ -- addr
\ Holds the Ethernet MAC address (six bytes). Note that you
\ must obtain these from the IEEE (www.ieee.org) or from other
\ sources.
  $00 c, $10 c, $8B c, $F1 c, $44 c, $20 c,

create IpAddress        \ -- addr
\ Holds the Ethernet IP address (four bytes).
  192 c,  168 c,  1 c,  251 c,              \ assign these as required

$50000000 equ EtherBase \ -- addr
0 equ SMC16?    \ -- flag ; true for 16 bit access code
0 equ fastCPU?  \ -- n    ; true for fast CPU
0 equ smcDiags? \ -- flag ; true for Ethernet diagnostics
0 equ eeprom?   \ -- flag ; true for attached EEPROM
1 equ sniff?    \ -- flag
  include %CpuDir%\drivers\smc91c9x.fth
  include %HWDir%\hware\Led.fth

: reboot        \ --
\ Reboot the CPU (equivalent to a hardware reset). This word
\ is used by NETBOOT.FTH if present.
  $07 $FFFFAC30 !  begin again
;

\ *** Define these constants carefully! ***
\ These assume that the bottom 128k of Flash is used for the
\ boot code, the middle is unused, and the final 64k is used
\ for data storage.
\ N.B. These constants are affected by the SECTION definitions.
0 equ BootMenu?                     \ -- n ; nz to compile boot menu
flashbase constant BootFlash        \ base address of boot Flash
                                    \ after mapping
$00020000 constant BootLen          \ length of boot Flash
$00000000 constant BootRAM          \ addr of boot code after mapping
$01020000 constant userflash        \ -- addr ; base address of user flash
$0        constant userflashlen     \ -- n ; size of user flash
$01070000 constant dataflash        \ -- addr ; base address of data flash
$00010000 constant datalen          \ -- n ; size of data flash
\ Where applications are copied to from the user flash
$00010000 constant AppRam           \ -- addr ; application area
$00060000 constant Applen           \ -- n ; length of application area
1 equ CPU=ARM                       \ if defined, selects ARM specific code
```

```
  include %CpuDir%\drivers\netcode \ Network order and CPU dependent
  include %CpuDir%\drivers\netboot \ Network boot loader

\ PowerNet configuration and setup
 1 equ ethernet?           \ nz for Ethernet systems
 0 equ slip?               \ nz to include SLIP
 0 equ tftp?               \ nz to include TFTP
 1 equ tcp?                \ nz for TCP as well as UDP
 1 equ telnet?             \ nz to include Telnet
 1 equ echo?               \ nz to include Echo
 0 equ snmp?               \ nz to include SNMP
 1 equ diags?              \ nz to include diagnostics (recommended)
  include %IpStack%\PowerNet.bld
```

21.7 End of compilation

All the files have been compiled. All that is required is library file resolution and some sanity checks.

```
\ ******************
\ End of compilation
\ ******************

libraries       \ to resolve common forward references
  include %CpuDir%\libARM
  include %CommonDir%\library
end-libs


\ *************
\ Sanity checks
\ *************

decimal

  cr ." Required USER size is     : " next-user @ .
  cr ." Current USER allocation is: " up-size .
Next-user @ up-size > [if]
\ Check that the USER area is large enough.
  cr ." *** Increase USER area size UP-SIZE in control file ***
  abort
[then]

\ XREF DUP                              \ where is DUP used
\ XREF-ALL                              \ full cross reference
\ XREF-UNUSED                           \ find unused words

\ ********
\ All done
\ ********
```

```
decimal

FINIS
```

22 Interpreter directives

Many of the ANS standard Forth words are available during interpretation. This includes the memory words such as `@` and `!`. Unless otherwise specified, all addresses are target addresses, and a character will be a byte unless the CPU is cell addressed.

22.1 ANS and common words

The majority of the CORE wordset is implemented for use during interpretation. These are not documented unless they have some special impact during cross-compilation of source code.

`USER` variables are not available during interpretation.

Words that convert numbers, e.g. `S>D` behave differently on 16 and 32 bit targets. Because the cross compiler's host Forth is a 32 bit Forth, for 16 bit targets only, the number handling defaults to 16 bits, but can be changed temporarily using `HOST-MATH` and `TARGET-MATH`.

`: >body cc/i \ xt -- pfa`
Only works for children of `CREATE`.

`: >does cc/i \ xt -- runtime`
Given an xt, returns the target runtime address. Only works for children of `CREATE`.

`: >in cc/i \ -- addr`
Not available because `>IN` is usually a `USER` variable.

`: base cc/i \ --`
`BASE` is not available because it is a `USER` variable.

`: BASE-36 CC/I \ --`
Selects a number base of 36 for packing 0..9, A..Z.

`: postpone cc/i \ --`
`POSTPONE` is not available during interpretation.

`: state cc/i`
`STATE` is not available because it may be a `USER` variable.

22.2 Specials

These words are usually only needed when debugging cross-compiler extensions.

`: find cc/i \ caddrt -- symh +/-1 | caddrh 0`
A compiler debugging tool. Given a counted string in the target, returns a host symbol offset and +/-1 if found, otherwise returns an address in the host and zero. Don't ask!

`: '(h) cc/i \ -- xth ; '(h) <name>`
Performs a host `'`.

`: c@(h) cc/i \ addr -- b`
Operates on host memory.

`: w@(h) cc/i \ addr -- w`
Operates on host memory.

`: @(h) cc/i \ addr -- x`

Operates on host memory.

`: c!(h)  cc/i  \ b addr --`
Operates on host memory.

`: w!(h)  cc/i  \ w addr --`
Operates on host memory.

`: !(h)  cc/i  \ x addr --`
Operates on host memory.

22.3 Section handling

`: section  cc/i  \ start end --`
Creates a new region of target memory. The *end* address is the last address **in** the section. Usually used in the form:

```
<start> <end> xDATA SECTION <name>
```

`: bank  cc/i  \ start end --`
Defines a regions of memory which is paged. Usually used in the form:

```
<start> <end> xDATA BANK <name>
```

`: pages  cc/i  \ id --`
Defines a page in the current `BANK`. Used in the form:

```
<id> PAGES <name>
```

`: write-ignore  cc/i  \ --`
Causes writes to the current section to be ignored and reads to return 0. Must be used immediately after a `SECTION` definition, e.g.

```
<start> <end> xDATA SECTION <name>   WRITE-IGNORE
```

`: ignorable  cc/i  \ --`
Causes the next section to be a `WRITE-IGNORE` section and reads to return 0. Must be used immediately before a `SECTION` definition, e.g.

```
<start> <end> IGNORABLE UDATA SECTION <name>
```

`IGNORABLE` is mostly used to provide a catch-all section for Umbilical Forth operations. This is usually defined as the first section so that it is searched last. It prevents the host allocating any memory for the section.

`: write-invalid cc/i  \ --`
Causes writes to the current section to generate an error and reads to return 0. Must be used immediately after a `SECTION` definition, e.g.

```
<start> <end> xDATA SECTION <name>   WRITE-INVALID
```

`: write-fail  cc/i  \ --`
Causes writes to the current section to generate an error. Reads return the last data written. Used when section data is still needed but the section may no longer be written to.

`: write-mem  cc/i  \ --`
Causes writes to the current section to be enabled again, reversing the action of `write-fail`.

`: all-saved  cc/i  \ --`
Causes the whole of a section to be saved. Usually used immediately after a `SECTION` definition for CPUs that boot from the top of memory, e.g.

```
  <start> <end> xDATA SECTION <name>   ALL-SAVED
```

`: code/data cc/i \ --`
Marks the current section in a Harvard target as shared CODE/DATA. Such sections are occasionally found in targets that map an address range into both CODE and DATA address spaces.

`: single-section-only cc/i \ --`
Used for hosted systems to indicate that the current section contains the `CDATA`, `IDATA` and `UDATA` sections. The current section should be the only section defined.

`: single-section? cc/i \ -- flag`
Returns true if `single-section-only` has been used.

`: data-file cc/i \ -- size ; DATA-FILE <filename>`
Loads a binary image from disk to target memory at `HERE` and `ALLOT`s the target memory. The size of the target data is returned.

`: idata cc/i \ --`
Switch to the current `IDATA` section.

`: udata cc/i \ --`
Switch to the current `UDATA` section.

`: cdata cc/i \ --`
Switch to the current `CDATA` section.

`: COrg cc/i \ addr --`
Define a new compilation address in the current `CDATA` section. New data will be laid at this target address.

`: IOrg cc/i \ addr --`
Define a new compilation address in the current `IDATA` section. New data will be laid at this target address.

`: UOrg cc/i \ addr --`
Define a new compilation address in the current `UDATA` section. New data will be laid at this target address.

`: Org cc/i \ addr --`
Define a new compilation address in the current section. New data will be laid at this target address.

`: origin cc/i \ -- addr(t)`
Returns start address of the current `CDATA` section.

`: sec-base cc/i \ -- addr(t)`
Returns start address of the current section.

`: sec-top cc/i \ -- addr(t)`
Returns end address of the current section.

`: sec-len cc/i \ -- len`
Returns the length of the current `CDATA` section.

`: sec-end cc/i \ -- addr(t) ; returns BP of current section`
Returns the buffer pointer address in the current section. This is the end address less the amount `RESERVE`d.

`: reserve cc/i \ n -- addr(t)`

Allocates *n* bytes down from top of the current `UDATA` section, returning the start address of the buffer.

`: unused cc/i \ -- n`
Return the amount of free space (BP-DP) in the current section.

`: .section-report cc/i \ --`
Displays a report about the memory usage of each section.

`: .sections cc/i \ -- ; show section status`
Displays a report about the memory usage of each section, and indicate the current section.

`: .curr-sections cc/i \ --`
Display the current section of each type. The current type is marked with an asterisk.

`: flush-idata cc/i \ --`
If not already done, flush the primary vocabulary data to RAM and then copy the used portions of the `IDATA` sections to the current `CDATA` section. This directive is often used when the size of a binary file needs to extended to a certain alignment. The alignment code then follows after `flush-idata`. See also `lay-idata`.

```
cdata flush-idata            \ lay IDATA sections NOW
here $1FF +  $-0200 and org  \ force to 512 byte boundary
```

`: lay-idata cc/i \ --`
Add the the used portions of the `IDATA` sections to the current `CDATA` section. There are no interlocks. Each `IDATA` section is laid down as

```
 len | addr | pageid | len bytes ...
```

The list is finished by a length of zero.

`: appFlush-idata cc/i \ --`
As `flush-idata` above, but for the application portion of a split bootloader/application system.

```
cdata appFlush-idata         \ lay IDATA sections NOW
here $1FF +  $-0200 and org  \ force to 512 byte boundary
```

`: Download-sections cc/i \ --`
Download the CDATA memory sections to target Flash. This requires taget CPU-specific downloader software (and perhaps hardware) to have been installed. Consult the target-specific manual for details.

`: -download-prompts cc/i \ --`
`Download-sections` will be performed without a prompt.

`: +download-prompts cc/i \ --`
`Download-sections` will be performed with a prompt. This is the default condition.

`: +SaveCdataOnly cc/i \ --`
When the output files are written, only the `CDATA` sections are written.

`: -SaveCdataOnly cc/i \ --`
When the output files are written, all section types are written. This is the default.

`: setBinExt \ -- ; setBinExt .ext`

By default, binary output files have the extension ".img". You can change this to another extension. For example, some Flash programmers require binary files to have ".bin" extensions.

```
setBinExt .bin
```

`: +NamesWithCRCs \ --`
When section files are generated, a four-digit hex CRC is added to the file name, e.g. a section called `Prog` may be saved as the file *Prog12AB.img*.

`: -NamesWithCRCs \ --`
CRCs are not appended to section file names. This is the default.

`: TargetFlashStart \ addr --`
Used in a non-Harvard target to indicate that the target can compile into Flash or EEPROM that starts at the given address, e.g.

```
$F000 TargetFlashStart
```

The given value will be the initial value of the target `DP` and the target `RP` (if it exists) will be set to the end of the IDATA section.

22.4 Comma and friends

`: ,c cc/i \ x --`
Compiles into the current `CDATA` section.

`: w,c cc/i \ w --`
Compiles into the current `CDATA` section.

`: c,c cc/i \ b --`
Compiles into the current `CDATA` section.

`: ,i cc/i \ x -- ; compiles into IDATA`
Compiles into the current `IDATA` section.

`: w,i cc/i \ w -- ; compiles into IDATA`
Compiles into the current `IDATA` section.

`: c,i cc/i \ b -- ; compiles into IDATA`
Compiles into the current `IDATA` section.

`: ,(r) cc/i \ x --`
Compiles into the current `IDATA` section.

`: w,(r) cc/i \ w --`
Compiles into the current `IDATA` section.

`: c,(r) cc/i \ b --`
Compiles into the current `IDATA` section.

`: align cc/i \ --`
Force target alignment in the current section.

`: Calign cc/i \ --`
Force target alignment in the current `CDATA` section.

`: Ialign cc/i \ --`
Force target alignment in the current `IDATA` section.

`: Ualign cc/i \ --`
Force target alignment in the current `UDATA` section.

`: aligned cc/i \ addr -- a-addr ; 3 + -4 and`
Given an address pointer this word will return the next `ALIGNED` address subject to system wide alignment restrictions. This is a `NOOP` if alignment is not enabled.

`: aligning? cc/i \ -- flag`
True if aligning is turned on.

`: Here cc/i \ -- addr(t)`
Return the dictionary pointer for the current `xDATA` section.

`: Chere cc/i \ -- addr(t)`
Return the dictionary pointer for the current `CDATA` section.

`: Ihere cc/i \ -- addr(t)`
Return the dictionary pointer for the current `IDATA` section.

`: Uhere cc/i \ -- addr(t)`
Return the dictionary pointer for the current `UDATA` section.

`: there cc/i \ -- addr`
Return the dictionary pointer for the current `IDATA` section. OBSOLETE - use `IHERE` instead.

`: Allot cc/i \ #bytes --`
Adjust dictionary pointer for the current `xDATA` section.

`: Allot&Erase cc/i \ #bytes --`
Adjust dictionary pointer for the current `xDATA` section, and fill the space with zeros.

`: CAllot cc/i \ #bytes --`
Adjust dictionary pointer for the current `CDATA` section.

`: IAllot cc/i \ #bytes --`
Adjust dictionary pointer for the current `IDATA` section.

`: UAllot cc/i \ #bytes --`
Adjust dictionary pointer for the current `UDATA` section.

`: Allot-RAM cc/i \ #bytes --`
Adjust dictionary pointer for the current `IDATA` section. OBSOLETE - use `IALLOT` instead.

`: bin-aligned cc/i \ addr u -- addr'`
Force alignment of *addr* to a *u* byte boundary where *u* is a power of two.

`: bin-align cc/i \ n --`
Force alignment of the current section to a *u* byte boundary where *u* is a power of two.

22.5 Defining words

`: : cc/i \ --`
The cross-compiler's version of `:`.

`: :noname cc/i \ -- xt`
The cross-compiler's version of `:NONAME` returns a target xt/address.

`: I: CC/I \ --`
Behaves like `:` but also builds a host analogue. Used occasionally for words that are `IMMEDIATE` in the target. Mostly superceded by `INTERPRETER ... TARGET`

`: set-compiler cc/i \ xt --`
Hosted VFX Forth targets only: Set the code generator field of the last word defined.

`: COMP:  cc/i  \ --`

Hosted VFX Forth targets only: Set up target code generator field and start a `:NONAME` definition to perform compilation actions for the last word defined.

`: AsmCode  cc/i  \ -- ; replacement for ASSEMBLER`

Starts a section of assembler code.

`: Code  cc/i  \ -- sys`

Used in the form `CODE <name>` to start the word `<name>` that is written in assembler.

`: Icode  cc/i  \ --`

Used in the form `ICODE <name>` to start the word `<name>` that is written in assembler. The word will **always** be inlined.

`: End-Code  \ sys --`

Finish an `AsmCode`, `CODE`, or `ICODE` definition.

`: immediate  cc/i  \ --`

Marks the last target definition as `IMMEDIATE`.

`: +FlashCompile  \ --`

The target compiles with the immediade bit set to 0 for `IMMEDIATE` words. This is used by systems that compile directly into Flash which is erased to $FF.

`: -FlashCompile  \ --`

The target compiles with the immediade bit set to 1 for `IMMEDIATE` words. This is the default condition and is for systems that compile into RAM or Flash erased to $00. Used immediately after a definition, and marks a definition as suitable for binary inlining. Used with `CODE` definitions mostly, in the form:

```
code foo
  ...
end-code inline
```

`: inline-always cc/i  \ -- ; always inline a CODE defn.`

Marks a definition that will always be binary inlined. Use in the form:

```
code foo
  ...
end-code inline-always
```

`: Label  cc/i  \ addr -- ; -- addr ; <addr> LABEL <name>`

Creates a label in the host, which returns the given target address. In essence, just a name for an address. Use as:

```
chere 8 + LABEL <name>
```

`: L:  cc/i  \ -- ; -- chere ; L: <name>`

Generates a `label` at the current `CDATA` address.

```
L: <name>
```

`: proc  cc/i  \ "name" --`

Define a named section of assembler, often used to start a subroutine used within other assembler code. When `name` is referenced, its address is returned. Equivalent to `AsmCode L: <name>`.

```
proc foo   \ returns address when referenced
  ...
end-code

...
  jsr foo
...
```

`: PL:  cc/i  \ -- ; -- chere ; L: <name>`

Only for 16 bit systems. A paged version of `L:`. When reference, returns a 32 bit value with the page number in the high 16 bits and the address within the page in the low 16 bits.

`: l>hilo  cc/i  \ p:a16 -- page addr`

Only for 16 bit systems. Separates a paged address, e.g. from `PL:`, into page and address form.

`: l>lohi  cc/i  \ p:a16 -- addr page`

Only for 16 bit systems. Separates a paged address, e.g. from `PL:`, into address and page form.

`: equ  cc/i  \ x -- ; -- x ; x EQU <name>`

Creates a host word only. When referred to, the value x is interpreted or compiled as literal (number). Use in the form:

```
  $FF equ Amask   \ interpreted as a literal.
```

`: Constant  cc/i  \ x -- ; -- n ; x CONSTANT <name>`

Creates a target `CONSTANT`.

`: const  \ x -- ; -- x ; x CONST <name>`

Creates a target `EQU` or `CONSTANT` according to the setting below. This allows some code to be visible `CONSTANT`s during debugging and smaller headless `EQU`s in release code. This can be particularly useful to make I/O addresses from peripheral files visible during debugging by loading the peripheral definitions file a second time:

```
...
+const-visible           \ CONST = CONSTANT
  include sfrK60.fth
-const-visible           \ CONST = EQU
```

`: -const-visible  \ --`

`CONST` is a synonym for `EQU` (default condition). See `CONST` above.

`: +const-visible  \ --`

`CONST` is a synonym for `CONSTANT`. See `CONST` above.

`: const=equ  \ --`

`CONST` is a synonym for `EQU` (default condition). See `CONST` above.

`: const=constant  \ --`

`CONST` is a synonym for `CONSTANT`. See `CONST` above.

`: 2Constant  cc/i  \ xd --`

Creates a target `2CONSTANT`.

`: Create  cc/i  \ -- ; -- addr ; imitates host CREATE`

Creates a target `CREATE`. Note that the address returned by the child of `CREATE` is dependent on the `xDATA` setting, normally `CDATA`. You can create a table in initialised RAM with:

```
idata create foo   \ -- addr
  1 ,  2 ,  4 ,  8 ,
cdata
```

`: Target-Only  cc/i  \ -- ; only compile into target`
Disable automatic creation of host analogues of `CREATE ... DOES>` words.

`: Host&Target  cc/i  \ -- ; compile into target and host`
Enable automatic creation of host analogues of `CREATE ... DOES>` words.

`: defer  cc/i  \ -- ; DEFER <name>`
Makes a `DEFER`red word in the target.

`: assign  cc/i  \ -- xt ; ASSIGN <action> TO-DO <deferred-word>`
Used in the form

```
assign <action> to-do <deferredword>
```

In practice this is a synonym for `'`.

`: to-do  cc/i  \ xt -- ; ASSIGN <action> TO-DO <deferred-word>`
Used in the form

```
assign <action> to-do <deferredword>
```

`: action-of  cc/i  \ -- xt ; ACTION-OF <deferred-word>`
Returns the current action of a target `DEFER`red word.

`: User  cc/i  \ n "name" -- ; -- addr`
Creates a new `USER` variable in the target at offset *n* of name `name`. Note that during interpretation, you cannot get the address of a `USER` variable as the cross compiler does not know what this will be.

`: Value  cc/i  \ n "name" -- ; -- n`
The cross-compiler's version of `VALUE`.

```
x VALUE <name>
```

`: +ImmVals  cc/i  \ --`
In the target, children of `VALUE` will be `IMMEDIATE` and be state-smart. This is the default condition.

`: -ImmVals  cc/i  \ --`
In the target, children of `VALUE` will not be `IMMEDIATE`, but will set a compiler called `valComp,` in the code generator field of each child.

`: Variable  cc/i  \ -- ;`
The cross-compiler's version of `VARIABLE`. VARIABLE <name>

`: CVariable  cc/i  \ -- ; -- addr`
The cross-compiler's version of `CVARIABLE`, which behaves like `VARIABLE` but only reserves a single byte.

`: WVariable  cc/i  \ -- ; -- addr`
In 32 bit systems, this creates a 16 bit variable.

`: 2Variable  cc/i  \ -- ; -- addr`
The cross-compiler's version of `2VARIABLE`.

`: Buffer:  cc/i  \ #bytes -- ; x Buffer:<name>`
Creates a named buffer of size *#bytes* in `UDATA` space.

`: vocabulary cc/i \ -- ; --`
The cross compiler version of `VOCABULARY`.

`: wordlist cc/i \ -- wid(t)`
The cross compiler version of `WORDLIST`.

`: set-#wid-threads cc/i \ n -- ; must be binary number`
Sets the number of threads used by `WORDLIST`.

`: get-#wid-threads cc/i \ -- n ; must be binary number`
Returns the number of threads used by `WORDLIST`.

`: set-#voc-threads cc/i \ n --`
Sets the number of target threads used by `Vocabulary`. `N` must be 1, 2, 4, 8, or 16.

`: get-#voc-threads cc/i \ -- n`
Returns the number of threads used by `Vocabulary`.

`: asm cc/i \ --`
A `NOOP` for source compatibility with some hosted Forths. OBSOLETE.

`: unhook-asm cc/i \ -- ; NOOP for compatibility with regular Forths`
A `NOOP` for source compatibility with some hosted Forths. OBSOLETE.

22.6 Words involving ' (tick)

`: to cc/i \ xt -- ; to <valuechild>`
Used to change the contents of a child of `VALUE`.

`: -> cc/i \ xt -- ; -> <valuechild>`
A synonym for `TO` above.

22.7 Strings

`: >number cc/i \ ud1 c-addr1 len1 -- ud2 c-addr2 len2`
Accumulate digits from string c-addr1/u1 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 host's `BASE`.

`: PLACE cc/i \ caddr len dest --`
Place the string *caddr/len* as a counted string at *dest*. For Harvard architectures, dest must be in an `IDATA` or `UDATA` section.

`: $, cc/i \ caddr len --`
Lay the string into the dictionary at `HERE`, and reserve space for it. The dictionary space is `not` aligned.

`: "" cc/i \ "text" -- ; -- caddr`
The pre-ANS MPE version of `C"`. OBSOLETE.

`: C" cc/i \ "text" -- caddr`
Returns the target address of a counted string.

`: S" cc/i \ "text" -- caddr len`
Returns the target address and length of a string.

`: Z" cc/i \ "text" -- zaddr`
Returns the target address and length of a zero-terminated string.

`: ,"  cc/i  \ "text" --`
Lays a counted string in the current section. Often used to build string tables, e.g.

```
create StringTable  \ -- addr
 ", first string"
 ", second string"
 ...
 0 c,
```

Note that some targets may force alignment of these strings. This is CPU specific.

`: ",  cc/i  \ "text" --`
A synonym for `,"`.

`: z",  cc/i  \ --`
Lays a zero-terminated string in the current section. Often used to build string tables, e.g.

```
create zStringTable  \ -- addr
 z", first string"
 z", second string"
 ...
 0 c,
```

Note that some targets may force alignment of these strings. This is CPU specific.

`: M",  cc/i  \ -- ; SFP063`
Lays a counted string in the current section. Unlike `",` and `,"` above, any text macros in the string are expanded before the string is built. Often used to build string tables, e.g.

```
create StringTable  \ -- addr
 m", On source line %l%"
 m", in source file %f%"
 ...
 0 c,
```

Note that some targets may force alignment of these strings. This is CPU specific.

`: compare  cc/i  \ caddr1 len1 caddr2 len2 -- +1/0/-1`
`COMPARE`s two strings in target memory.

`: scan  cc/i  \ caddr len char -- caddr' len'`
`SCAN`s in target memory.

`: skip  cc/i  \ caddr len char -- caddr' len'`
`SKIP`s in target memory.

`: -trailing  cc/i  \ caddr len -- caddr len'`
`-TRAILING` in target memory.

`: -leading  cc/i  \ caddr len -- caddr' len'`
`-TRAILING` in target memory.

`: cmove  cc/i  \ addr1 addr2 u --`

Move data area, first byte first.

`: cmove>  cc/i  \ addr1 addr2 u --`
Move data area, last byte first.

`: CMOVEC  cc/i  \ addr1 addr2 u --`
Copy code to data. Harvard targets only.

`: CMOVE->C  cc/i  \ addr1 addr2 u --`
Copy data to code. Harvard targets only.

`: CMOVEC->C  cc/i  \ addr1 addr2 u --`
Copy code to code. Harvard targets only.

`: parse  cc/i  \ char -- caddr len`
Parse the next token from the terminal input buffer using <char> as the delimiter. The next token is returned at `IHERE` as a c-addr/len string description. Note that `PARSE` does not skip leading delimiters.

`: parse(h)  cc/i  \ char -- caddr len`
The original host version of *\fo{PARSE) returning a string in host memory.

`: parse-name  cc/i  \ -- addr len`
`PARSE-NAME` replaces `BL WORD COUNT` in most cases. The returned string is at `IHERE`.

`: parse-name(h) cc/i  \ char -- caddr len`
The original host version of *\fo{PARSE-NAME) returning a string in host memory.

22.8 Escaped strings

The escaped string parser parses a string up to an unescaped '"', translating '\' escapes to characters much as C does. The supported escapes (case sensitive) are:

`\a` BEL (alert)

`\b` BS (backspace)

`\e` ESC (escape, ASCII 27)

`\f` FF (form feed, ASCII 12)

`\l` LF (ASCII 10)

`\m` CR/LF pair - for HTML etc.

`\n` newline - CRLF for Windows/DOS, LF for Unices

`\q` double-quote

`\r` CR (ASCII 13)

`\t` HT (tab, ASCII 9)

`\v` VT

`\z` NUL (ASCII 0)

`\"` "

`\[0-7]+` Octal numerical character value, finishes at the first non-octal character

`\x[0-9a-f][0-9a-f]`
Two digit hex numerical character value

`\\` backslash itself

\ before any other character represents that character

`: \",  cc/i  \ "text" --`
Lay an escaped string into the dictionary as a counted string. The end of the string is not aligned.

`: Z\",  cc/i  \ "text" --`
Lay an escaped string into the dictionary as a zero terminated string. The end of the string is not aligned.

`: C\"  cc/i  \ "text" -- caddr`
An escaped version of `C"`.

`: S\"  cc/i  \ "text" -- caddr len`
An escaped version of `S"`.

22.9 Memory operators

`: Cdump  cc/i  \ addr len -- ; dump CODE space`
Harvard targets only.

`: @C  cc/i  \ addr -- x`
Fetch cell from code space. Harvard targets only.

`: C@C  cc/i  \ addr -- b`
Fetch byte from code space. Harvard targets only.

`: !C  cc/i  \ x addr --`
Store cell into code space. Harvard targets only.

`: C!C  cc/i  \ n addr --`
Store byte into code space. Harvard targets only.

`: w@c  cc/i  \ addr -- w`
Fetch 16 bits from code space. 32 bit Harvard targets only.

`: w!c  cc/i  \ w addr --`
Store 16 bits into code space. 32 bit Harvard targets only.

`: dump  cc/i  \ addr(t) len --`
`DUMP` target memory in 8 bit byte format.

`: ldump  cc/i  \ addr(t) len --`
`DUMP` target memory, displaying 32 bit items. This is useful to see values on little-endian targets. 32 bit targets only.

`: wdump  cc/i  \ addr(t) len --`
`DUMP` target memory, displaying 16 bit items. This is useful to see values on little-endian targets. 32 bit targets only.

`: fillc  cc/i  \ addr len char --`
`FILL CDATA` memory. Harvard targets only.

`: erasec  cc/i  \ addr len --`
`ERASE CDATA` memory. Harvard targets only.

`: blankc  cc/i  \ addr len --`
`BLANK CDATA` memory. Harvard targets only.

`: fill  cc/i  \ addr len char --`

`FILL` `IDATA` or `UDATA` memory. Harvard targets only.

`: erase cc/i \ addr len --`
`ERASE` `IDATA` or `UDATA` memory. Harvard targets only.

`: blank cc/i \ addr len --`
`BLANK` `IDATA` or `UDATA` memory. Harvard targets only.

`: fill cc/i \ addr len char --`
`FILL` target memory. Conventional targets only.

`: erase cc/i \ addr len --`
`ERASE` target memory. Conventional targets only.

`: blank cc/i \ addr len`
`BLANK` target memory. Conventional targets only.

`: marker cc/i \ -- ; MARKER <name>`
Builds a host `MARKER`.

22.10 Files and Paths

`: include cc/i \ "<filename>" --`
Compile a file.

```
include <filename>
```

If the name starts with a '"' the file name contains the characters between the first and second '"' characters but does not include the '"' characters themselves. If you need to include names that include '"' characters, delimit the string with '(' and ')'. In all other cases a space is used as the delimiting character. Text macros are expanded when the file is opened.

`: cwd cc/i \ "<pathname>" --`
Change directory. Synonym for CD, avoids HEX conflict.

`: dir cc/i \ -- ; dir <dirname>`
Display a directory listing.

22.11 Vocabulary handling

In this system, words are looked up in the vocabularies and wordlists in the search order. The first entry is often referred to as the top entry. New words are created in the definitions vocabulary or wordlist.

`: forth cc/i \ --`
Selects the primary target vocabulary as the first in the search order.

`: only cc/i \ --`
Set the minimum search order as the current search order.

`: also cc/i \ --`
Duplicate the first wordlist in the search order.

`: previous cc/i \ --`
Drop the current top of the search order.

`: definitions cc/i \ --`
Set the current top of the search order as the current definitions wordlist.

`: words cc/i \ --`

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

`: +show-unresolved cc/i \ --`
`WORDS` shows unresolved sysmbols.

`: -show-unresolved cc/i \ --`
`WORDS` does not show unresolved sysmbols.

`: vocs cc/i \ --`
Display all vocabularies by name.

`: order cc/i \ --`
Display the current search-order. WIDs created with `VOCABULARY` are displayed by name, others are displayed as numeric representations of the WID.

`: words(h) cc/i \ --`
Display the words in the host search order.

`: vocs(h) cc/i \ --`
List the vocabularies in the underlying host Forth system.

`: order(h) cc/i \ --`
List the search order in the underlying host Forth system.

22.12 Conditional Compilation

The following words allow the use of `[IF] ... [ELSE] ... [THEN]` blocks to control which pieces of code are compiled/executed and which are not. These words behave in the same manner as compiled definitions of `IF ... ELSE ... THEN` structures but take immediate effect even outside definitions.

`: [IF] cc/i \ flag --`
Marks the start of a conditional compilation clause. If flag is TRUE compile/execute the following code, otherwise ignore all up to the next `[ELSE]` or `[THEN]`.

`: [ELSE] cc/i \ --`
Marks the start of the ELSE clause of a conditional compilation block.

`: [THEN] cc/i \ --`
Marks the end of a conditional compilation clause.

`: [ENDIF] cc/i \ --`
Marks the end of a conditional compilation clause.

`: [DEFINED] cc/i \ "<name>" -- flag`
Look to see if the word exists in the target search order and return flag TRUE if the word exists.

```
[defined] foo [if]  ...  [then]
```

`: [UNDEFINED] cc/i \ "<name>" -- flag`
The inverse of `[DEFINED]`. Return TRUE if `<name>` does not exist.

22.13 Debugging aids

`: stack-check cc/i \ --`
Generates an error if the stack depth is not empty.

`: Escape cc/i \ --`
Abandon cross compilation and enter the host Forth.

`: whereis cc/i \ -- ; WHEREIS <name>`
Display the filename and line number for the source code of the word `<name>`.

`: Locate cc/i \ -- ; LOCATE <name>`
Display the source location (file/line) of the word. If configured to do so, the source will displayed in your editor.
Configuration of `LOCATE` depends on the operating system, IDE and your preferred editor.

Windows: If using AIDE, set up your editor using AIDE's menu item:

```
IDE -> Configure Edit/Locate...
```

If using the compiler without AIDE, use the compiler menu item

```
Options -> Set Editor...
```

Linux: As for the host VFX Forth. Run up the cross compiler. Tell the system the editor and locate commands. Then exit. The configuration is now saved.

Set your preferred editor, e.g.

```
editor-is /bin/vi
```

Tell VFX Forth how your editor can be called to go a particular file and line. Use in the form

```
SetLocate  <rest of line>
```

where the text after `SetLocate` is used to define how parameters are passed to the editor, e.g. for Emacs, use

```
SetLocate +%l% "%f%" &
```

The rest of line following `SetLocate` is used as the editor configuration string. Within the editor configuration string 'f' will be replaced by the file name and 'l' will be replaced by the line number. If you use file names with spaces, you should put quotation marks around the `%f%` text macro. Finish the line with `" &"` to run the editor detached from VFX Forth.

OS/X: TBD.

`: Loc cc/i \ -- ; LOC <name>`
Synonym for `LOCATE`.

`: xref cc/i \ "<name>" -- ; XUSES <name>`
If XREFs are enabled, display the words which use `<name>`.

`: uses cc/i \ "<name>" -- ; USES <name>`
If XREFs are enabled, display the words which use `<name>`. OBSOLETE.

`: xref-all cc/i \ -- ; XREF-ALL`
Perform an `XREF` on all target words. The report can be pasted from the screen to an editor for further processing.

`: xref-unused cc/i \ -- ; XREF-UNUSED`
List unused target words.

`: +xrefs cc/i \ --`
Enable cross reference generation.

`: -xrefs cc/i \ --`
Disable cross reference generation.

`: xref-kb cc/i \ kb -- ; set size of XREF table`
Set the size of the cross-reference table (default 1Mb) in kilobytes, e.g. `2048 xref-kb` request a 2Mb table.

`: labels cc/i \ -- ; show label names`
List all the target `LABEL`s.

`: equates cc/i \ -- ; show equate names`
List all the target `EQU`ates.

`: Compilers cc/i \ --`
List all the words that are special when compiling. This list will include user extensions and words that have special compilers/optimisers.

`: Interpreters cc/i \ --`
List all the words that are special when interpreting.

`: Help cc/i \ --`
Displays a short list of tools.

`: .dword cc/i \ u --`
Displays *u* as a 32 bit hex word.

`: .lword cc/i \ u --`
Displays *u* as a 32 bit hex word.

`: .word cc/i \ w --`
Displays *w* as a 16 bit four-digit hex item.

`: .byte cc/i \ b --`
Displays *b* as an 8 bit two-digit hex item.

`: .hex cc/i \ u --`
Displays *u* as a 32 bit hex word.

`: .dec cc/i \ n --`
Displays *n* as a signed decimal number.

`: dis(h) cc/i \ --`
Disassembles a host word.

`: dump(h) cc/i \ addr len --`
`DUMP`s host memory.

22.14 Turnkey

`: make-turnkey cc/i \ -- ; MAKE-TURNKEY <name>`
Tells the compiler which word to run after target initialisation. The xt of the word is placed at the target label `CLD1`.

22.15 Floating point formats, ANS and Forth200x

The ANS Forth standard specifies that floating point numbers must be entered in the form 1.234e5 and must contain a point '.' and 'e' or 'E', and that double integers are terminated by a point '.'.

This situation prevents the use of the standard conversion words in international applications because of the interchangable use of the '.' and ',' characters in numbers. Because of this, the cross-compiler's host VFX Forth uses two four-byte arrays, `FP-CHAR` and `DP-CHAR`, to hold the characters used as the floating point and double integer indicator characters. By default, `FP-CHAR` is initialised to '.' and `DP-CHAR` is initialised to to ',' and '.'. For strict ANS compliance, you should set them as follows **before** `CROSS-COMPILE` is run.

```
: ans-floats    \ --
```
Sets the entry format to strict ANS compliance.
```
  [char] . dp-char !
  [char] . fp-char !
```

```
: mpe-floats    \ -- ; for existing and most legacy code
```
Sets the entry format to current MPE practice.
```
  [char] , dp-char !
  [char] . dp-char 1+ c!
  [char] . fp-char !
```

```
: legacy-floats \ -- ; for legacy code
```
Sets the entry format to legacy MPE behaviour as used by the Forth 5 and Forth 6 compilers.
```
  [char] , dp-char !
  [char] . fp-char !
```

You can of course set these arrays to hold any values which suit your application's language and locale. Note that integer conversion is always attempted before floating point conversion. This means that if the `FP-CHAR` and `DP-CHAR` arrays contain the same character, floating point numbers must contain 'e' or 'E'. If the arrays are all different, a number containing the `FP-CHAR` will be successfully converted as a floating point number, even if it does not contain 'e' or 'E'.

22.16 Floating point

```
: integers      cc/i    \ --
```
Disable floating point conversion.

```
: reals         cc/i    \ --
```
Enable floating point conversion.

22.16.1 Software floating point

The software floating point pack uses different formats for 16 and 32 bit targets. These conversion words are sensitive to the target cell width.

```
: s>f           cc/i    \ n -- f
```
Convert a signed integer to a float.

```
: f#            cc/i    \ -- f ; F# 1.234e5
```
Treat the following text as a floating point number.

```
: f/            cc/i    \ f1 f2 -- f1/f2
```
Floating point divide.

```
: f*            cc/i    \ f1 f2 -- f1*f2
```

Floating point multiplication.

```
: f+            cc/i    \ f1 f2 -- f1+f2
```
Floating point addition.

```
: f-            cc/i    \ f1 f2 -- f1-f2
```
Floating point subtraction.

```
: fdup          cc/i    \ f -- f f
```
Floating point version of `DUP`.

```
: f.            cc/i    \ f --
```
Display a floating point number.

22.16.2 Hardware floating point

If the target is 32 bit and has an IEEE floating point unit, an IEEE pack is provided on the host for help with interpretation. Both the host and the target assume the use of a separate floating point stack as defined in the ANS Forth and the Forth 2012 standards. Look at the target documentation to see if the target handles 32 and/or 64 bit floats. The host can handle both.

```
: s>f           cc/i    \ n -- ; F: -- f
```
Convert a signed integer to a float.

```
: f#            cc/i    \ F: -- f ; F# 1.234e5
```
Treat the following text as a floating point number.

```
: f/            cc/i    \ F: f1 f2 -- f1/f2
```
Floating point divide.

```
: f*            cc/i    \ F: f1 f2 -- f1*f2
```
Floating point multiplication.

```
: f+            cc/i    \ F: f1 f2 -- f1+f2
```
Floating point addition.

```
: f-            cc/i    \ F: f1 f2 -- f1-f2
```
Floating point subtraction.

```
: fdup          cc/i    \ F: f -- f f
```
Floating point version of `DUP`.

```
: fsqrt         cc/i    \ F: f1 -- f2
```
Floating point square root.

```
: f.            cc/i    \ F: f --
```
Display a floating point number.

```
: sf.hex        cc/i    \ F: f --
```
Display a float as a 32 bit hex representation.

```
: df.hex        cc/i    \ F: f --
```
Display a float as a 64 bit hex representation.

```
: sf,           cc/i    \ F: f --
```
Lay a 32 bit floating point number.

```
: df,           cc/i    \ F: f --
```
Lay a 64 bit floating point number.

`: f,  cc/i  \ F: f --`
Lay a target native floating point number. The size is set by `setFloatSize` below.

`: fliteral  cc/c  \ F: f --`
Compiles a floating point literal from the top of the floating point stack, for example : foo ... [123 s>f] fliteral ... ;

`: f#  cc/c  \ --`
Used to compile a floating point literal in the form:

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

`: setFloatSize  cc/i  \ u --`
Set the size (in bytes of memory) of a floating point number.

`: setFloatAlignment  cc/i  \ u --`
Set the alignment of floats in memory.

22.17 Structures

The data structure words implement records, fields, and field types

The following syntax is used:

```
STRUCT <name>    \ -- len
  n FIELD <field1>
  m FIELD <field2>
  ...
END-STRUCT
```

When `<name>` is executed, it returns the size of the structure.

A field adds its base offset to the given address [that of the record or subrecord]. A record returns its length, and so can be used as an input to field.

```
   len FIELD <name>
 n len ARRAY-OF <name>
```

`: STRUCT  cc/i  \ Comp: "name" -- sym addr 0 ; Run: -- size`
Begin definition of a new structure. Use in the form `STRUCT <name>`. At run time `<name>` returns the size of the structure.

`: END-STRUCT  cc/i  \ sym addr size --`
Terminate definition of a structure.

`: FIELD  cc/i  \ offset n "<name>" -- offset' ; addr -- 'addr`
Create a new field within a structure definition of size n bytes.

`: FIELD-TYPE  cc/i  \ n -- : Run: addr -- addr+n ; Child: addr -- addr+n`
Define a new field type of size n bytes. Use in the form `<size> FIELD-TYPE <name>`. When `<name>` executes used in the form `<name> <name2>` a field `<name2>` is created of size n bytes.

`cell(t) field-type int  \ "<name>" -- ; addr -- addr+cell`
Creates a field holding a cell.

`cell(t) field-type ptr  \ "<name>" -- ; addr -- addr+cell`
Creates a field holding a cell that is an address.

22.18 C isms

These words are useful when extracting register definitions from C header files.

`: #define  cc/i  \ <spaces"NAME"> <eol"value-def"> -- ; Exec: -- value`
A simple version of C's #define which creates a `CONSTANT`. Any text between the definition name and the end of the line is `EVALUATE`d when `NAME` is defined. The result of evaluating this text must be a single-cell integer, and is used to create an `EQU`ate.

`: //  cc/i  \ --`
An implementation of the C++ single line comment.

`: /*  cc/i  \ --`
A C comment of the form '/* ... */'. Note that both '/*' and '*/' must be whitespace delimited.

`: enum  cc/i  \ --`
Process an enum of the form:

```
enum <name> { a, b, c=10, d };
```

<name> is ignored. The elements appear as Forth equates or constants, switchable in the same way as `CONST` definitions. The definition may extend over many lines. C comments may occur after the ',' separator, e.g.

```
JIM = 25, // comment about this line
```

22.19 Miscellaneous

`: .sources  cc/i  \ --`
Display a list of source files.

`: .lo  cc/i  \ xxyy -- 00yy`
Extracts the bottom 8 bits of an item. 16 bit targets only.

`: .hi  cc/i  \ xxyy -- 00xx`
Extracts the top 8 bits of an item. 16 bit targets only.

`: ByteRevL  cc/i  \ a:b:c:d -- d:c:b:a`
32 bit byte reversal.

`: ByteRevW  cc/i  \ a:b:c:d -- b:a:d:c`
Byte reverse both 16 bit pairs.

`: ByteRevWZ  cc/i  \ a:b:c:d -- 0:0:d:c`
Byte reverse low 16 bits and then zero extend.

`: ByteRevWS  cc/i  \ a:b:c:d -- s:s:d:c`
Byte reverse low 16 bits and then sign extend.

`: synonym  cc/i  \ -- ; SYNONYM <new> <old>`
Creates a new name (in the host only) for a target word. When the new name is referenced, the old name is used.

`: No-Heads  cc/i  \ --`
Marks the following code as having no target heads, regardless of the use of `INTERNAL` and `EXTERNAL`.

`: internal cc/i \ --`
Marks the following code as having no target heads. See `EXTERNAL`.

`: external cc/i \ --`
Marks the following code as having target heads. See `INTERNAL`.

`: Target-Width cc/i \ width --`
Sets the maximum number of characters in a target head.

`: headerless cc/i \ -- ; for compatibility`
Synonym for `INTERNAL`.

`: headers cc/i \ -- ; for compatibility`
Synonym for `EXTERNAL`.

`: behead cc/i \ -- ; for compatibility`
`NOOP` for compatibility with old hosted systems.

`: Is-Action-Of cc/i \ addr -- ; <addr> is-action-of <name>`
Used to set the address of the run-time action of a defining word. Mostly used in minimal kernels in which several defining words have the same run-time action.

`: nt-access-ports cc/i \ --`
Initialise direct port access under Windows NT and derivatives.

`: testing cc/i \ n -- ; set testing level, 0=no testing`
Set the testing level, default is zero.

`: [test cc/i \ --`
If `TESTING` has been set non-zero, the code between `[TEST` and `TEST]` will be processed, otherwise it will be ignored.

`: test] cc/i \ -- ; end of [TEST ... TEST] block`
Ends a `[TEST ... TEST]` block.

`: .forwards cc/i \ --`
Show words that have been forward references.

`: XTL? cc/i \ -- flag`
Return true (non-zero) if the Umbilical link is active.

`: +Listing cc/i \ --`
Start listing of source code during compilation.

`: -Listing cc/i \ --`
Stop listing of source code during compilation.

`: Kb cc/i \ n -- nKb ; nKb = n * 1024`
Given n, returns n kilobytes (1024).

`: Mb cc/i \ n -- nMb ; nMb = n * 1048576`
Given n, returns n megabytes (1048576=1024*1024).

`: kHz cc/i \ n -- Hz`
Multiply by 1000.

`: MHz cc/i \ n -- Hz`
Multiply by 1,000,000.

`: reveal cc/i \ --`

Expose the current definition to the dictionary search mechanism in order to make recursive definitions. In some other Forth systems this word is called `RECURSIVE`.

`: hide cc/i \ --`
Hide current definition from dictionary search. Usually used after `REVEAL`.

`: it cc/i \ -- xt`
Get XT of last colon definition

`: Date$, cc/i \ --`
Compile current date as a counted string.

`: Time$, cc/i \ --`
Compile current time as a counted string.

`: DateTime$, cc/i \ --`
Compile date and time as a single counted string.

`: Log cc/i \ --`
Display the compiler word by word log.

`: No-Log cc/i \ --`
Turn off the compiler word by word log.

`: Logging? cc/i \ -- flag`
Returns true if the log is turned on

`: AtCold cc/i \ xt --`
Usually used in the form:

```
' foo AtCold
```

so that `foo ( -- )` is executed at start up. The word must have a null stack effect. See the target manual for details of the implementation. Compilation pauses if `AtCold` is used before the target version has been defined.

`: THROW cc/i \ n --`
For custom error handlers.

`: abort cc/i \ n --`
For custom error handlers.

22.20 Starting and finishing cross-compilation

`: cross-compile \ --`
To start cross-compiling, use the word `CROSS-COMPILE ( -- )`. At this point, the compiler "pulls down the shutters" and enters cross-compilation mode. Apart from compiler directives that are interpreted, code after this will be compiled into the target image instead of compiled onto the cross-compiler.

`: interactive cc/i \ --`
When `INTERACTIVE` is used after `CROSS-COMPILE` and before `FINIS`, the compiler will not exit after compilation finishes, but will enter an interactive mode in which the symbol table and image data are preserved. This allows you to use the other debugging tools with a standalone target compilation.
To mark the end of the cross-compilation phase, use `FINIS` for a standalone application or `UMBILICAL-FORTH` to start debugging an Umbilical Forth system. `FINIS` is used to finish cross-compilation completely, whereas `UMBILICAL-FORTH` is used to finish the batch portion of the

compilation and to start the cross target link ready for interactive testing of an Umbilical Forth target.

`: BootFinis cc/i \ --`

Used in a split bootloader/application system to mark the end of the bootloader portion and the start of the application code.

`: AppFinis cc/i \ --`

Used in a split bootloader/application system to mark the end of the application code. Used in these systems instead of `finis`

`: Afterwards \ -- ; afterwards ." hello 1" cr`

The following text up to the end of the line is saved and will later be executed when `FINIS` is run.

`: fcopy \ -- ; fcopy source dest`

Copy one file to another. Usually used with `Afterwards`. The file names and operation status are displayed.

`: fdel \ -- ; fdel dest`

Delete a file. Usually used with `Afterwards`. The file name and operation status are displayed.

`: bye cc/i \ --`

Used in interactive mode to exit the cross compiler. `UMBILICAL-FORTH` is used to finish the batch portion of the compilation and to start the cross target link ready for interactive testing of an Umbilical Forth target.

`: BatchMode? cc/i \ -- flag`

Return true if the compiler not being run from AIDE or for interactive use. When running the compiler from batch file or script, use the `/PauseOff` command-line switch to turn off the interactive mode.

22.21 Build numbering

The build numbering system allows you to generate a string in the system which can be used for displaying version information.

The system relies on a file (normally called *BUILD.NO*) which holds the complete build version string. The string can consist of any characters, e.g "Version 1.00.0034". The contents of the file can be placed as a counted string in the dictionary. After successful compilation of your application, `UPDATE-BUILD` will update the build number file by treating **all** the digits in the build string as a single number to be incremented.

`: MAKE-BUILD cc/i \ addr(t) --`

Read build file info, copy to target as a counted string.

`: BUILD$, cc/i \ --`

Read build file info, and lay in the target as a counted string.

`: UPDATE-BUILD cc/i \ --`

Update the build number file. Place this just before `FINIS`.

`: BUILDFILE cc/i \ "<filename>" --`

Set the build file name. Use in the form:

```
BUILDFILE BUILD.NO
```

22.22 Checksum generation

Checksums of various types can be generated for your code. More than one checksum can be generated in different memory areas. To avoid problems with forward references, checksums are not generated until `FINIS` has finished patching various labels and has laid the initialised RAM table.

```
: CHECKSUM      cc/i    \ start end addr type --
```

Set a region to be checksummed when `FINIS` executes. More than one region can be checksummed.

`Start` Start address.

`End` Last address to be included in the checksum. For 16 and 32 bit operations, this address must be aligned as required. Note that CRCs are byte operations.

`addr` Where the checksum is placed.

`type` Type of checksum from the words below.

```
: SIMPLE8       cc/i    \ -- n
```

Generate an 8 bit checksum by adding bytes.

```
: SIMPLE16      cc/i    \ -- n
```

Generate an 16 bit checksum by adding byte pairs.

```
: SIMPLE32      cc/i    \ -- n
```

Generate an 32 bit checksum by adding four-byte units.

```
: CCITT         cc/i    \ -- n
```

Generate a CCITT checksum.

```
: CRC16         cc/i    \ -- n
```

Generate a CRC16 checksum.

```
: LRCC16        cc/i    \ -- n
```

Generate an LRCC16 checksum.

```
: SDLC          cc/i    \ -- n
```

Generate an SDLC checksum.

```
: CRC32         cc/i    \ -- n
```

Generate a 32 bit CRC.

```
: CRCxModem16   cc/i    \ -- n
```

Generate a 16 bit XModem CRC. The result is stored in native order.

```
: CRCxModem16-0 cc/i    \ -- n
```

Generate a 16 bit XModem CRC. The result is stored in big-endian order.
`CRCxModem16-0` is usually used when you want to force the checksum of a block to zero, e.g.

```
  <start> <end-2> <end-1> CRCxModem16-0 checksum
```

where <start> is the first address of the block and <end> is the last address in the block. If you later perform an XModem checksum from <start> to <end> the CRC will be zero.

22.23 Disassembler

Compilers generating subroutine threaded or native code include a disassembler.

`: xDISASM/al cc/i \ addr len --`
Disassemble the given range.

`: xdisasm/ft cc/i \ from to --`
Disassemble the given range.

`: xdisasm/f cc/i \ addr --`
Disassemble from the given address until a return is encountered.

`: DISASM/al cc/i \ addr len --`
Disassemble the given range.

`: disasm/ft cc/i \ from to --`
Disassemble the given range.

`: disasm/f cc/i \ addr --`
Disassemble from the given address until a return is encountered.

`: xdasm cc/i \ -- ; XDASM <word>`
Disassemble the given word.

`: dasm cc/i \ -- ; DASM <word>`
Disassemble the given word.

`: dis cc/i \ -- ; DIS <word>`
Disassemble the given word.

`: see cc/i \ -- ; SEE <word>`
Disassemble the given word.

22.24 Library files

A library file contains words which are compiled only if required, i.e. they have been used but not yet defined. Library files are scanned between the directives `LIBRARIES` and `END-LIBS`. The code between `LIBRARIES` and `END-LIBS` is repeatedly interpreted until the number of forward references and unresolved symbols remains constant. For example:

```
libraries
  include libfile1.fth
  include libfile2
end-libs
```

`#16 cells constant /ipNest \ -- u`
Max Staorage size per source nesting

`16 constant #ipNests \ -- u`
Maximum number of nesting levels.

`/ipNest #ipNests * constant #ipnesting \ -- u`
Size of storage array

`#ipNesting buffer: ipArray[] \ -- addr`
Array in which source input positions are saved

```
ipArray[] value CurrIP[]        \ -- addr
```
Points at next free position.

```
: n!            \ xn..x1 n addr --
```
Save n items at addr including n.

```
: n@            \ addr --
```
Retrieve n items saved by N!.

```
: getIP[]       \ addr --
```
Reload input position from buffer.

```
: libraries     cc/i    \ -- 0 #unres #forward ; interpretive loop
```
Starts repetitive execution of the code up to `END-LIBS`.

```
: end-libs      cc/i    \ pass# #unres #forward -- pass# #unres' #forward'
```
Ends the block started by `LIBRARIES`.

```
: [required]    cc/i    \ "<name>" -- true
```
Returns true if symbol exists and is forward referenced. All words in the library file are usually defined in the form:

```
[required] foo [if]
: foo  ...  ;
[then]
```

23 Converting from earlier versions

The v7 compiler does not support the Leburg EPROM emulators.

23.1 From v6.2 onwards

Converting from v6.2 onwards should require no changes to your control files or target code.

AIDE and the host Forth have changed, and the v7 compiler runs much faster. Use the version of AIDE supplied with the v7 compiler. The latest AIDE requires a slightly different configuration which is documented in the AIDE manual and release notes. The key part to remember is that the v7 compiler requires both the `-IDE` and `/IDE` command line switches.

23.2 Converting from v6.0

23.2.1 Generic I/O

The v6.0 and v6.1 target versions of `KEY`, `KEY?`, `TYPE`, `EMIT` and `CR` were `DEFER`red. The new code is not deferred. Instead two new user variables, `IPVEC` and `OPVEC`, hold the address of a vector table which points to the actions of these words. The new system is called *Generic I/O* and is documented in a separate chapter of this manual.

Generic I/O makes it **much** easier to add new I/O devices. Much recent MPE code requires generic I/O and we strongly recommend that you convert to it.

23.2.2 Multitasker

The multitasker is now list driven rather than table driven. This gives faster context switching. The major differences are indicated below.

The control file uses the equate `TASKING?` which is set true or false to control compilation. You do not have to specify the maximum number of tasks.

A task is defined by the word `TASK <name>` which allocates the resources for a task, and returns a taskid at run time. This identifier is the base address of the `USER` area instead of a task number.

The separate task control blocks (TCBs) are no longer required. Instead, the multitasker is controlled by several (currently 6) cells at the start of the `USER` area.

The execution action of a task is no longer held in the TCB. Instead, the word `INITIATE (xt task -- )` replaces `ACTIVATE` to start the task. For symmetry, the word `DEACTIVATE` is replaced by `TERMINATE`.

The word `START:` allows the use of nameless task actions.

23.2.3 User variables

From version 6.1 onwards, the word `+USER` can be used to add a `USER` variable of a given size:

```
<size> +USER <name>
```

The use of +USER avoids any need to know the offset at which the variable starts. The kernel code relies on +USER and new application code should use +USER in preference to USER.

23.2.4 Heap

All targets now come with heap code. There are two versions, *HEAP16.FTH* and *HEAP32.FTH*, which use different control block structures. They are optimised for 16 bit and 32 bit targets respectively. The application word set is the same.

23.3 Upgrading from v5

The process of converting code from a version 5 MPE Forth Cross Compiler. The simplest case is for code bases from the 8 and 16 bit v5 targets that have 16 bit Forth implementations. The stages for these also apply to the 32 bit targets for which there are now VFX code generators, but some additional work is also required.

23.3.1 Basic v5 conversion

Memory definitions

The v7 compiler uses the SECTION model for memory the control file (.CTL extension). Change all the lines of the form:

```
<start> <end+1> ROMBASE <name>
```

to:

```
<start> <end> CDATA SECTION <name>
```

The SECTION model uses different words to return the start and end of a section, so the definition of equates such as EM will need to be changed. See the new control files in the <*cpu*>*/CONFIGS* directory for examples.

You must define at least one CDATA, IDATA, and UDATA section. The v5 compilers have no equivalent of a UDATA section, and this can be a dummy definition, but it must exist.

After all the memory definitions have been made, select a default section of each type and put in CDATA to make CREATE and friends behave like the v5 compilers.

If your processor requires start-up vectors at the end of the kernel code section (e.g. 68HC11), use the SAVE-ALL directive after the definition of the code section. This forces the compiler to save the whole section, rather than just from the start to the current end of the code.

Assembler changes

The use of the word ASSEMBLER to denote the start of a piece of assembly code is no longer supported, and the use of FORTH to end it is now deprecated. Convert all pieces of code that use these words from the form:

```
ASSEMBLER
  ...
FORTH
```

To:

```
ASMCODE
  ...
END-CODE
```

Bank switched systems

The bank switching code has changed, especially in that PAGE-WORD is now called PAGE-EXECUTE, and the parameter passing may be slightly different. This means that you cannot produce a byte for byte equivalent system unless PAGE-EXECUTE is headerless.

Conditional compilation

The previous directives IF(,)ELSE(and)ENDIF are now replaced by their ANS equivalents [IF], [ELSE], [THEN] and the extension [ENDIF] which behaves just like [THEN].

Conditional compilation may be nested.

The words [DEFINED] <name> and [UNDEFINED] <name> can be used to return a flag if the target word <name> has already been defined.

The word [REQUIRED] <name> returns true if a word has been forward referenced but has not yet been defined. This is used with the LIBRARIES and END-LIBS directives to allow you to make files whose contents are only compiled if the words have been referenced but are currently not defined.

Interpreted calculations

These notes only apply to 16-bit targets.

The v7 compilers all use a 32-bit host Forth, whereas the v5 compilers for 16-bit targets used a 16-bit host Forth. Some calculations performed at compile time, such as baud rate calculations, relied on truncation of the 16-bit results. By default, the v7 compilers for 16-bit targets treat numbers in this way. However, the interpreted integer math operators are all 32-bit. If your calculations rely on truncation of 16-bit results, it is better to redo them using 32-bit arithmetic and to use the directives HOST-MATHS and TARGET-MATHS around the calculation so that large literals are not truncated. This often simplifies baud rate calculations where clock frequencies need a 32-bit value, and were represented as double numbers in the v5 code.

Startup code

The compiler directive MAKE-TURNKEY <name> places the xt of <name> at label CLD1. The startup code executes this word. The v5 label STRTUP is no longer needed, and the new entry code should be used in place of the v5 code.

In addition, the structure of the initialised data table header has changed to permit multiple IDATA sections and banked RAM.

Testing

Unless you have used some particularly clever defining words, the stages above are all that is needed to convert direct threaded 16-bit Forths from v5 compilers.

When MPE converts target code from v5, we rename the image files (.IMG extension) as .IMO files, and then ensure that the new IMG file is byte-for-byte compatible with the old one. The DOS FC file utility can used to test this:

```
FC <image>.IMG <image>.IMO /B
```

We suggest that you copy your working target code directory to a new one, and perform the conversion until you obtain byte-for-byte equivalence of your application.

23.3.2 Converting from DTC to VFX compilers

The version 5 compilers produce what is termed direct threaded code (DTC), which is a particular implementation strategy for Forth. The VFX v7 compilers produce subroutine threaded code (STC) with optimisations. The VFX code generator provides very little change in code density and sometimes an improvement that depends heavily on coding style. You can expect a 10:1 improvement in performance with a VFX code generator.

The v7 targets are based on an ANS Forth model, rather than the Forth-83 model used with the v5 target code. Converting from Forth-83 to ANS is covered in a separate chapter of this manual.

Strategy

In order to convert an application from DTC to VFX/STC, it is probably easier to start from the new code base, as this will provide an easier long term upgrade path. The recommended stages are:

1. Generate a new kernel for your target
2. Build a conversion harness that provides any missing words
3. Apply all the changes discussed for basic v5 conversion
4. Convert all code definitions to the new register model used by v6. See the assembler chapter in the accompanying processor specific manual for details. This usually involves switching the data and return stack pointers, and preserving the frame stack pointer if it is used. Compile and test each file in turn. You will probably need to revisit stage 2.
5. Compile your application as a whole. At this stage, you will probably have to go back round through stage 2. Repeat this cycle until you get a clean compile.
6. Test your application as a whole.

Some additional considerations are:

- Is the VFX code generator good enough that you can remove many code definitions in favour of high level Forth definitions, so enhancing maintainability and portability? Mostly, yes.
- Can coded interrupt routines now be rewritten in high level Forth for maintainability and portability? Yes, especially if you are changing hardware at the same time.

COMPILE, and ,

The word `COMPILE, ( xt -- )` compiles the code that calls a definition. This is the only portable way to generate a call to a word. Because of the change from DTC to STC and optimised code, you cannot predict what code will be generated. Any use of the Forth word `,` (comma) to lay code rather than data **must** be replaced by `COMPILE,`.

```
: MYMAGIC
  ...
  [] FOO ,  [] BAR ,
  ...
; IMMEDIATE
```

should be relaced by

```
: MYMAGIC
  ...
  POSTPONE FOO  POSTPONE BAR
  ...
; IMMEDIATE
```

or

```
: MYMAGIC
  ...
  [] FOO COMPILE,  [] BAR COMPILE,
  ...
; IMMEDIATE
Vector tables
```

In direct threaded code, you could lay down the address of a Forth word by turning the compiler on. Two forms of this could be found:

```
CREATE TABLE
  ] A B C D [
L: MYLABEL
  ] FOO [
: BAR
  ... MYLABEL @ EXECUTE ...  ;
```

This worked because MPEs DTC code uses the address of the Forth word as the execution token (xt). However, this is not a portable technique, and fails if the xt is not cell sized (e.g. the MPE 32 bit 8086/186 target uses a 16 bit xt) or generates native code (e.g. `CALL FOO`). The recommended portable technique is:

```
CREATE TABLE
  ' A ,
  ' B ,
  ' C ,
  ' D ,
L: MYLABEL
  ' FOO ,
```

or:

```
L: MYLABEL
  0 ,
  ...
  ' FOO MYLABEL !    \ Avoids forward reference
: BAR
  ... MYLABEL @ EXECUTE ...
;
```

Choice of word names ANS and Forth-83

The ANS Forth committee (in which MPE participated) were careful not to make changes that break existing code. Thus some words whose function varied according to vendor have had name changes. The v7 compilers still generate the MPE versions, but also include the ANS versions. For long term portability of both code and programmers, it is suggested that new code use the ANS versions. The help documentation includes an ANS draft specification that is technically identical to the ratified ANS/ISO Forth specification. Note that this is a standards document, and so is not drafted in the same way as the glossary for a user manual is drafted.

For more details see the chapter on converting Forth-83 code to ANS.

23.3.3 CREATE CDATA IDATA UDATA and sections

When a section name is interpreted, its action is to make that section the current section for `CREATE` and words derived from `CREATE`. `CREATE` will return the next address in the selected section. The following words are also affected:

```
, ALIGN ALIGNED ALLOT C, HERE W, UNUSED
```

The result is that if you have three sections `ROM` (CDATA), `IRAM` (IDATA), and `URAM` (UDATA) you must be careful to select the right one before using `CREATE`. The following sequence has different effects according to which section is selected:

```
CREATE FOO
  5 ,  6 ,  7 ,
ROM CREATE FOO          \ FOO points into ROM
  5 ,  6 ,  7 ,         \ table cannot be changed
IRAM CREATE FOO         \ FOO points into IRAM
  5 ,  6 ,  7 ,         \ table is initialised
                        \ and can be changed
URAM CREATE FOO         \ FOO points into URAM
  5 ,  6 ,  7 ,         \ table is invalid!
                        \ URAM values exist only at
                        \ compile time
```

If you have several sections of a type, and all you wanted to do was to select the current section of that type, you could use `CDATA`, `IDATA` or `UDATA` instead.

As a result of these ANS changes, the technique used in version 5 compilers for selecting between ROM and RAM data is neither desirable nor efficient. But it will still work if `CDATA` has been selected. You may find it worthwhile to rewrite defining words that used to use `HERE`, `THERE`, `ALLOT` and `ALLOT-RAM`. Overall, MPE has found the new notation to be far more flexible, and it has been well received.

23.3.4 COMPILER, INTERPRETER, HOST, TARGET and ASSEMBLER

In both version 5 and the version 7 compilers, the use of defining words is mostly handled automatically by the compiler.

For those cases where it is not handled automatically, or because there are compile time words which are not desirable or needed in the target code, a new mechanism has been provided for adding words into the compiler. The actions of these directives are discussed in more detail elsewhere in the manual. These examples are more informal.

The directive `TARGET` is used to return to cross compilation into the target, and should be used to terminate any of the other directives.

The directive `INTERPRETER` compiles new definitions into the cross interpreter, and uses target referring versions of words such as `@` and `!`. Use `TARGET` to return to cross compilation. The following example can be used to add a defining word (that cannot be handled automatically) to the system without having a target version. All the code after `DOES>` is compiled into the target.

```
INTERPRETER
: SEMAPHORE    \ -- ; -- addr [child]
  IDATA
  CREATE
    0 ,               \ counter
    0 ,               \ task id
  CDATA
  DOES>
;
TARGET
```

The directive COMPILER compiles new definitions into the cross compiler, creating a word which is only found at compile time, in other words it is IMMEDIATE but is not found during interpretation.

```
COMPILER
: !++           \ n addr  addr ; store and step address
  TUCK !  CELL +
;
TARGET
```

The effect of this is to add a new word to the compiler, which can reference all the other compiler words. This is effectively a macro. Note that any reference inside such a word to structure words like IF and ENDIF will be taken as references to the compilers versions of IF and ENDIF, and not to the normal Forth versions.

The directive HOST is used to add words to the underlying Forth system. It is useful when adding words that may be used as factors of other words, and where any variables may only exist during compilation.

```
HOST
: FOO .... ;
TARGET
```

The directive ASSEMBLER is used to add macros to the cross assembler.

```
ASSEMBLER
: bar  ...  ;
TARGET
```

23.3.5 Umbilical Forth

The Umbilical Forth protocol has been extended and modified slightly. The *TARGEND.FTH* file used must be the one supplied with the v7 compiler if you want interactive testing. You will not be able to produce a byte for byte equivalent file from a v7 compiler that will run on your target with the v7 compiler, but you should be able to test it with the v5 compiler. Recompiling your code with the old TARGEND.FTH file on the v7 compiler should produce a file identical with that produced by the v5 compiler, and so you should be able to run the code and interact with it using the v5 compiler.

The v7 TARGEND.FTH code also has facilities for using the multitasker with Umbilical Forth. This is controlled by the conditional compilation facilities.

23.3.6 FLOATS and REALS

The word FLOATS used to enable the floating point package conflicts with an ANS word. Its function is replaced by REALS. The package can be turned off by INTEGERS.

24 Converting from Forth-83 to ANS

This chapter is not a complete guide to converting applications to ANS standard Forth. It summarises some of the changes that are likely to affect your applications. A copy of the ANS

specification is supplied with the cross compiler.

Where Forth-83 words and MPE extensions do not conflict with the ANS standard, they have been retained in the cross compiler. Compatibility with previous code generated by the MPE Forth cross compiler v5 (and v4 in most cases) has been retained to the level that v5 code for the 16 bit DTC targets can be used with only minor changes to produce byte for byte identical output.

24.1 Choice of word names

The ANS Forth committee (on which MPE acted as observers) were careful not to make changes that break existing code. Thus some words whose function varied according to vendor have had name changes. The v7 compilers still generate the old MPE words, but also include the ANS versions. For long term portability of both code and programmers, it is suggested that new code use the ANS versions. The help system includes an ANS draft specification that is technically identical to the ratified ANS/ISO Forth specification. Note that this is a standards document, and so is not drafted in the same way as the glossary for a user manual is drafted.

24.1.1 INVERT NOT and 0=

Because there was little commonality between Forth systems in the semantics of the word `NOT`, it has been excluded from the standard. Some vendors, including MPE, use `NOT` to mean a bitwise inversion (logical NOT), and others use it to mean the inversion of a flag (Boolean NOT, or 0=). The ANS word for a logical bitwise NOT is `INVERT`.

24.1.2 EXPECT SPAN and ACCEPT

Because the Forth-83 `EXPECT` does not return the number of bytes actually read, Forth-83 specifies a (`USER`) variable `SPAN` to hold this. ANS Forth defines a word `ACCEPT` which returns the length, rendering `SPAN` redundant.

24.1.3 S" and C"

Traditionally, Forth represented strings as a count byte followed by that many characters, in the same way as Pascal. With the increasing use of zero terminated strings in operating systems, and the increasing use of two-byte (Unicode) and multi-byte character sets, this description of strings has become less portable. Consequently the ANS committee accepted the idea that strings be represented as address and length pairs. For the most part, it is still true that a character usually means a byte, but in the next revision the ANS standard will be modified to make internationalisation easier to handle. In the meantime, it is recommended that new code be written using address/length pairs.

`S" <string>"` compiles a string that returns an address/length pair at run time, whereas `C" <string>"` compiles a string that returns the address of the count byte. The original MPE definition `""` still exists in the cross compiler, but is not recommended for new code.

24.1.4 ASCII CHAR and [CHAR]

The MPE word `ASCII` is state smart. When interpreted it returns the literal value of the following ASCII character. When compiled, it compiles the literal. Because state smart words are increasingly perceived as being capable of causing bugs that are hard to find, the interpretation behaviour is provided by the ANS word `CHAR`, and the compile time behaviour is provided by the ANS word `[CHAR]`.

```
CHAR A CONSTANT FOO
: BAR
  ... [CHAR] A EMIT ...
;
LSHIFT and RSHIFT
```

The MPE words `<<N` and `>>N` are replaced by `LSHIFT` and `RSHIFT` which have the same stack action:

```
x1 u -- x2
```

24.1.5 FORGET and MARKER

The time-honoured word `FORGET <name>` is now deprecated because of the variation in implementations and the portability issues raised by it. The ANS standard specifies the defining word `MARKER <name>` such that when `<name>` is executed, the dictionary is restored to its state before `<name>` was created by `MARKER`.

```
MARKER FOO              \ create a dictionary marker
  ...
FOO                     \ restores state, deleting FOO
```

24.2 Division

The Forth-83 standard mandated floored division. Whatever its merits, this has incurred a performance penalty on most CPUs. In ANS Forth the implementer may choose, and MPE has chosen to return to the usual symmetric division for `/` and words derived from it.

In order to retain the ability to perform floored division, the word `M/MOD` has been replaced by two words, `SM/REM` (symmetric) and `FM/MOD` (floored).

24.3 CREATE and friends

Section E.5 of the ANS specification suggests that, for embedded systems, `CREATE` be made sensitive to a current memory section. This makes it much easier to control where data is laid down, and removes the need for words to refer to each section of memory. This proposal caused much controversy, but some vendors have informally agreed and used a common notation, which is the basis of the MPE `SECTION` notation.

When a section name is interpreted, its action is to make that section the current section for `CREATE` and words derived from `CREATE`. `CREATE` will return the next address in the selected section, with the following words also being affected:

```
, ALIGN ALIGNED ALLOT C, HERE W, UNUSED
```

The result is that if you have three sections ROM (CDATA), IRAM (IDATA), and URAM (UDATA) you must be careful to select the right one before using CREATE. The following sequence has different effects according to which section is selected:

```
CREATE FOO
  5 ,  6 ,  7 ,
ROM CREATE FOO          \ FOO points into ROM
  5 ,  6 ,  7 ,         \ table cannot be changed
IRAM CREATE FOO         \ FOO points into IRAM
  5 ,  6 ,  7 ,         \ table is initialised
                        \ and can be changed
URAM CREATE FOO         \ FOO points into URAM
  5 ,  6 ,  7 ,         \ table is invalid, URAM values
                        \ exist only at compile time
```

If you have several sections of a type, and all you wanted to do was to select the current section of that type, you could use CDATA, IDATA or UDATA instead. Note that the CDATA, IDATA and UDATA directives are not part of the original proposal in section E.5 of the ANS specification.

As a result of these ANS changes, the technique used in version 5 compilers for selecting between ROM and RAM data is neither desirable nor efficient. But it will still work if CDATA has been selected. You may find it worthwhile to rewrite defining words that used to use both HERE, THERE, ALLOT and ALLOT-RAM. Overall, MPE has found the new notation to be far more flexible, and it has been well received.

24.4 >BODY and friends

Because of the number of implementation techniques, and because of the impact of embedded systems, ANS Forth specifies that >BODY is only standard when applied to the children of CREATE, and to words derived from it.

24.5 FLOATS and REALS

The word FLOATS used in v5 to enable the floating point package conflicts with an ANS word. Its function is replaced by REALS. The package can be turned off by INTEGERS.

24.6 CATCH and THROW

Before the ANS specification, Forth lacked a portable nested exception handler. The design of CATCH and THROW is excellent, and MPE recommends that they be used to replace the use of ABORT and ABORT", which can if necessary be defined in terms of CATCH and THROW.

CATCH and THROW are among the most significant introductions in ANS Forth, and enourmously improve the functionality reliability of error detection in Forth.

24.6.1 Description

The following description of the ANS words CATCH and THROW was written by Mitch Bradley:

CATCH is very similar to EXECUTE except that it saves the stack pointers before EXECUTEing the guarded word, removes the saved pointers afterwards, and returns a flag indicating whether or not the guarded word completed normally. In the same way that a Forth word cannot legally play with anything that its caller may have put on the return stack, and also is unaffected by how its caller uses the return stack, a word guarded by CATCH is oblivious to the fact that CATCH has put items on the return stack.

Here's the implementation of CATCH and THROW in a mixture of Forth and pseudo-code:

```
VARIABLE HANDLER        \ Most recent error frame
: CATCH          \ cfa -- 0|error-code
  <push parameter stack pointer on to return stack>
  <push contents of HANDLER on to return stack>
  <set HANDLER to current return stack pointer>
  EXECUTE
  <pop return stack into HANDLER>
  <pop & drop saved parameter stack ptr from return stack>
  0
;
: THROW         \ error-code --
  ?DUP
  IF
    <set return stack pointer to contents of HANDLER>
    <pop return stack into HANDLER>
    <pop saved parameter stack pointer from return stack>
    <back into the parameter stack pointer>
    <return error-code>
  THEN
;
```

The description as written implies the existence of a parameter stack pointer and a return stack pointer. That is actually an implementation detail. The parameter stack pointer need not actually exist; all that is necessary is the ability to restore the parameter stack to a known depth. That can be done in a completely standard way, using DEPTH, DROP, and DUP. Likewise, the return stack pointer need not explicitly exist; all that is necessary is the ability to remove things from the top of the return stack until its depth is the same as a previously-remembered depth. This can't be portably implemented in high level, but I neither know of nor can I conceive of a system without some straightforward way of doing so.

24.6.2 Sample implementation

In most Forth systems, the following code will work:

```
VARIABLE HANDLER        \ Most recent exception handler
: CATCH         \ execution-token -- error# | 0
                ( token ) \ Return address already on stack
  SP@ >R        ( token ) \ Save data stack pointer
  HANDLER @ >R  ( token ) \ Previous handler
  RP@ HANDLER ! ( token ) \ Set current handler to this one
  EXECUTE       ( )       \ Execute the word passed
  R> HANDLER !  ( )       \ Restore previous handler
```

```
  R> DROP       ( )       \ Discard saved stack pointer
  0             ( 0 )     \ Signify normal completion
;
: THROW         \ ?? error#|0 -- ?? error# ;
                \ Returns in saved context
  ?DUP
  IF
    HANDLER @ RP! ( err# ) \ Back to saved R. stack context
    R> HANDLER !  ( err# ) \ Restore previous handler
                  ( err# ) \ Remember error# on return stack
                  ( err# ) \ before changing data stack ptr.
    R> SWAP >R    ( saved-sp ) \ err# is on return stack
    SP!           ( token) \ switch stacks back
    DROP          ( )
    R>            ( err# ) \ Change stack pointer
  THEN
\ This return will return to the caller of catch, because
\ the return stack has been restored to the state that
\ existed when CATCH began execution.
;
```

Note the following features:

- CATCH and THROW do not restrict the use of the return stack.
- They are neither IMMEDIATE nor "state-smart"; they can be used interactively, compiled into colon definitions, or POSTPONEd without strangeness.
- They do not introduce any new syntactic control structures (i.e. words that must be lexically "paired" like IF and THEN).

To handle the case where there is no CATCH to handle a THROW, it is wise to CATCH the main loop of the application. A different solution, if you don't want to modify the loop, is to add this line to THROW:

```
  HANDLER @ 0= ABORT" Uncaught THROW"
```

24.6.3 Stack rules for CATCH and THROW

Let's suppose that we have the word FOO that we wish to "guard" with CATCH. FOO's stack diagram looks like:

```
 FOO     \ a b c -- d
```

Here's how to CATCH it:

```
<prepare argument for FOO>  ( a b c )
['] FOO  CATCH              ( x1 x2 x3 )
IF
  <some code to execute if FOO caused a THROW>
ELSE                          ( d )
  <some code to execute if FOO completed normally>
THEN
```

Note that, in the case where CATCH returns non-zero (i.e. a THROW occurred), the stack depth (denoted by the presence of x1,x2,x3) is the same as before FOO executed, but the actual contents of those 3 stack items is undefined. N.B. items on the stack UNDERNEATH those 3 items should not be affected, unless the stack diagram for FOO, showing 3 inputs, does not truly represent the number of stack items potentially modified by FOO.

In practice, about the only thing that you can do with those "dummy" stack items x1,x2,x3 is to DROP them. It is important, however, that their number be accurately known, so that you can know how many items to DROP. CATCH and THROW are completely predictable in this regard; THROW restores the stack depth to the same depth that existed just prior to the execution of FOO, and the number of stack items that are potentially garbage is the number of inputs to FOO.

24.6.4 Some more features

THROW can return any non-zero number to the CATCH point. This allows for selective error handling. A good way to create unique named error codes is with VARIABLEs as they return unique addresses without having to worry about which number to use, e.g.

```
VARIABLE ERROR1
VARIABLE ERROR2
```

creates 2 words, each of which returns a different unique number. For selective error handling, it is convenient to follow CATCH with a CASE statement instead of an IF. Here's a more complicated example:

```
BEGIN
  ['] FOO CATCH
  CASE
    0       OF  ." Success; continuing"   TRUE   ENDOF
    ERROR1  OF  ." Error #1; continuing"  TRUE   ENDOF
    ERROR2  OF  ." Error #2; retrying"    FALSE  ENDOF
    ( default ) ." Propagating error to another level" THROW
  ENDCASE                   ( retry? )
UNTIL
```

Note the use of THROW in the default branch. After CATCH has returned, with either success or failure, the error handler context that it created on the return stack has been removed, so any successive THROWs will transfer control to a CATCH handler at a higher level.

The CATCH and THROW scheme appealed to people because it is simpler than most other schemes, as powerful as any (and more powerful than most), is easy to implement, introduces no new

syntax, has no separate compiling behaviour, and uses the minimum possible number of words (2).

24.7 POSTPONE

This word was introduced to delay execution of a word without having to know whether the word is immediate or not. Inside a colon definition such as `BAR` below

```
: BAR
  ... POSTPONE FOO ...
;
```

will cause `FOO` to execute when `BAR` executes if `FOO` is `IMMEDIATE`, or if `FOO` is non-`IMMEDIATE`, `FOO` will be compiled when `BAR` executes. In most cases this is what was required, and the words `COMPILE` and `[COMPILE]` can be eliminated. The advantage of this is that the user does not need to know whether the target word is `IMMEDIATE` or not.

24.8 COMPILE, and ,

The word `COMPILE, ( xt -- )` compiles the code that calls a definition. This is the only portable way to generate a call to a word. Because of the change from DTC to STC and optimised code, you cannot predict what code will be generated. Any use of the Forth word `,` (comma) to lay code rather than data must be replaced by `COMPILE,`.

```
: MYMAGIC
  ...
  ['] FOO ,  ['] BAR ,
  ...
; IMMEDIATE
```

should be relaced by

```
: MYMAGIC
  ...
  POSTPONE FOO  POSTPONE BAR
  ...
; IMMEDIATE
```

or

```
: MYMAGIC
  ...
  ['] FOO COMPILE,  ['] BAR COMPILE,
  ...
; IMMEDIATE
```

25 Further information

25.1 MPE courses

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

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

Custom courses are available

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

25.2 MPE consultancy

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

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

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

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

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

- Communications protocols

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

25.3 Recommended reading

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

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

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

For more experienced Forth programmers:

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

Other miscellaneous Forth books:

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

All of these books can be supplied by MPE.

Index

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

X

Z

List of Tables

List of Figures