MPE Forth 6 Cross Compiler

User Manual

MPE Forth 6 Cross Compiler

User manual

Manual revision 1.600

Date 08 October 2003

Software

Software version 6.20

Package Number:	

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 from USA: 011 44 23 8063 1441
Fax: +44 (0)23 8033 9691 from USA: 011 44 23 8033 9691

e-mail: mpe@mpeltd.demon.co.uk
tech-support@mpeltd.demon.co.uk

web: www.mpeltd.demon.co.uk

Acknowledgements

MPE would like to thank the following people for their involvement in the production of this product:

Mark Davis, Stephen Pelc, Matthew Purvis, Paul Richards

Licence terms

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 6 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 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.

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. If you fill out the registration form enclosed and send it back to us, we will put you on our mailing list. 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.

Table of Contents

List of Figures

List of Tables

1 Installing the system

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

System requirements

To install and use the development system you need:

- PC with Windows 95 or NT or higher with 32 Mb or more of RAM.
- At least 8-20 Mbytes of free disc space, depending on the amount of CPU specific documentation provided.

Running the installer

The software is supplied on a CD. Use the Windows Explorer or 'My Computer' shortcut to navigate to your CD drive. To install the development system, double-click the file SETUP.EXE

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.

Everything you need can be accessed throught the AIDE shell. Many people find it useful to put a shortcut to AIDE\AIDE.EXE on the desktop.

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 DOC directory. They will be called RELEASE.XC6 and RELEASE.cpu, where for example RELEASE.51 refers to the 8051 compiler and RELEASE.ARM refers to the ARM/StrongARM compiler.

EPROM emulator drivers

The development system is supplied with facilities for the LeBurg EPROM emulator, series two. If you have the earlier series, please contact MPE if you do not have the

TSR021 or TSR041 drivers. All the necessary drivers are supplied with the EPROM emulators.

The installer needs to know what PC port address to map the emulator driver to. Note that for use with Windows 95, the DOS driver must be included in your AUTOEXEC.BAT file.

Later versions will support the MPE PowerROM Target Access Systems over Ethernet.

Port access under Windows 2000/NT

Direct access to I/O ports is required for the Leburg EPROM emulator drivers, the SPI parallel port drivers, and other target access drivers. If you are using Windows 2000/NT (any version) or later, direct port I/O requires a driver that permits this access, otherwise you will trigger a Windows 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" to point to GIVIO.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 in the section after the **`CROSS-COMPILE`** directive in which the compiler is configured.

2 The 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 6 cross-compiler with source code. Note that VFX compilers are only supplied with source code after a non-disclosure agreement has been signed.
- 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 debugging, but is smaller than the standalone Forth. Treat the target code as a resource for you to read and extend.
- Drivers for the LeBurg EPROM emulators
- The AIDE development environment AIDE is documented in a separate on-line manual.
- Tools directory. This includes file format converters from the memory images generated by the MPE Forth 6 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.XC6 and RELEASE.cpu 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 (XC620MAN.PDF) and the CPU specific manual.
- Target Code manuals. From v6.2 onwards, target code code is documented using MPE's DOCGEN system supplied with VFX Forth for Windows. 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\CPUCODE.PDF where CPU 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.

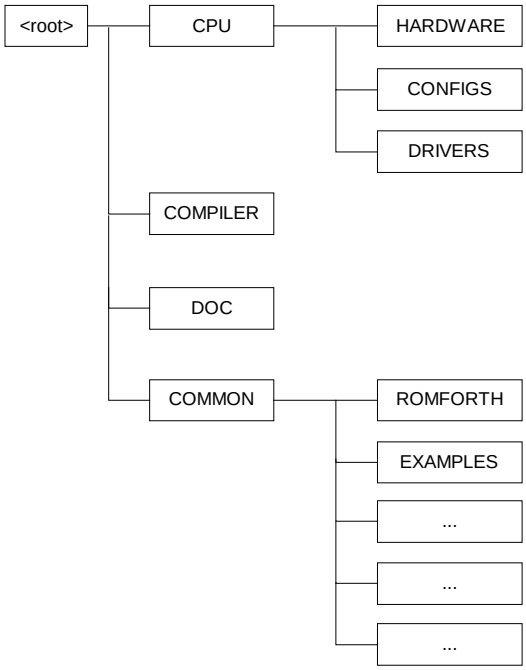

Figure 1: Directory structure

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 for non-VFX compilers, so that you can extend the compiler and rebuild it from scratch if required. Source code for VFX compilers is available under special terms which include a non-disclosure agreement.

The compiler can automate the generation of paged targets and also has a built-in cross-assembler. The compiler executable and associated files are in the directory COMPILER and the source is in the directory COMPILER\SOURCE if provided.

Standalone target Forth

A standalone target Forth is supplied with the full compiler, and not with the IRTC and Stamp verions.

Source code is supplied for developing a standalone target Forth. 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 or terminal emulator. This makes onsite

debugging without the cross-compiler very easy, and the Forth can be used for debugging, maintenance, and terminal configuration.

It also has a bigger wordset than an Umbilical Forth target, and consequently requires more memory. If you require the multitasker and don't want to change any of the supplied code, you must generate a standalone target Forth. The installer places the target source code in the directories COMMON and CPU. See the chapter on Generating a standalone Forth for details.

Umbilical Forth

Source code is supplied to generate an Umbilical Forth. Umbilical Forth is a significantly smaller Forth than the standalone target Forth, so that an interactive Forth can be generated which has a minimimal memory footprint. Umbilical Forth does not have all words defined in the standalone target Forth, but is useful if ROM space is at a premium. The Umbilical Forth source code is in the directories COMMON and CPU.

In most cases (except for 8051, 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.

Leburg EPROM emulator drivers

The cross-compiler can directly download code, as it is generated, to a LeBurg EPROM emulator. This is done by a TSR in the COMPILER directory.

Note that if you are using Windows NT/2000 or any other version of Windows that treats direct port I/O as a privileged instruction, you must install the NTPORT.EXE file from the COMPILER\XTRA directory as described in the installation section of the manual and modify your control fileto include the **NT-ACCESS-PORTS** directive.

Documentation directory

Much of the documentation is available on-line from 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 DPANS.HTM

The generic cross compiler and CPU specific manuals are supplied as PDF files. These may have been updated since the printed manuals. 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.

Control files

In nearly all cases, the cross compilation process is controlled by a “control” file, which defines the characteristics of the target hardware and memory layout and specifies which files to compile. You will find several in the CPU\CONFIGS directory/folder. For your job, copy one of the existing files and modify 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.

Changes from v6.0 to v6.1

The following are the major changes from the previous release of the cross compiler.

Inproved target performance. The VFX compilers have been improved since v6.0, and the 16 bit targets have been changed from direct threaded code (DTC) to subroutine threaded code (STC) and some optimisation. On an 8051, the v6.1 target can be twice as fast with no increase in code size. On other 16 bit targets, such as the 68HC12, the improvement is considerably more. VFX targets are ten to fifteen times faster than their DTC predecessors.

Target code disassembler. You can now disassemble any definition, whether Forth or code. You can also see some of the new code generation optimisations.

Generic I/O permits you to add new I/O devices very easily and to use the standard I/O words such as **KEY** and **EMIT** with them. Each task may access a different default device, and the default can be changed at any time.

Enhanced multitasker. The v6.1 target multitasker is fully list driven, which reduces the RAM and code space needed. There is a lso a stripped down version for use with the single chip model. The source code for these taskers is in the files MULTIxxEXT.FTH and MULTIxxINT.FTH.

TIMEBASE time and delay management system. Using a single periodic clock interrupt, you can generate delays, timeouts, and periodic execution very simply, using phrases such as:

```
' <action> <ms> AFTER    \ once after <ms> ms
' <action> <ms> EVERY    \ every <ms> ms
```

Enhanced Umbilical Forth. Previous versions of the compiler only supported Umbilical Forth with a serial line and an EPROM emulator. From 6.1 onwards, Umbilical Forth also supports other protocols, such as SPI access for the Atmel 89S8252 and BDM access for Motorola CPU32 cores. This allows fully interactive programming and debugging through the PC parallel port and frees up the target serial port.

Heap code is now provided with all targets, not just the VFX compilers.

Automatic build numbering system. You can embed a build version string into the application. This string can be automatically updated after each successful build of your project.

More examples.

Enhanced documentation. The manual has been revised, incorporating many comments suggestions from users since the introduction of the Forth 6 compilers.

This compiler is now available in three versions, Professional, IRTC, and Forth Stamp.

Professional. The full compiler with all tools, compiler source code, all target source code, floating point, multitasker(s), TIMEBASE system, heap, automated test code, Umbilical Forth and Forth Stamp hardware (if available).

IRTC. The same compiler and hardware but without compiler source code, and with Umbilical Forth only. TIMEBASE, floating point, automated test code, and bank switched targets are not supported.

Forth Stamp. As the IRTC compiler, but with code space restricted to 8k bytes. No cross reference facilities are available. No multitasker is provided.

Changes from v6.1 to v6.2

Examples. The EXAMPLES directory includes target code for a State Machine compiler, PID loop controllers, and a complete file system. The I2C support has been overhauled and new devices added.

Kernel. Numerous detail changes for efficiency and to support extensions such as the PowerNet v3.0 TCP/IP stack. The 32 bit heap code has been overhauled for speed and size. The ROMFORTH facilities have been extended with XMODEM in both directions.

Compiler. The following INTERPRETER directives are new:

```
[REQUIRED] IT HIDE REVEAL KB MB TESTING [TEST TEST]
.FORWARDS
```

The use of structure fields in COMPILER ... TARGET definitions produces optimised code in VFX compilers.

Documentation. The high level kernel code in the COMMON directory now has separate documentation.

AIDE shell. LOCATE supports external editors as well as ForthEd.

Learning Forth

If you are unfamiliar with Forth, MPE can supply a range of books and training courses. 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.

3 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.

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 **FROM-FILE** or **INCLUDE**, 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. If a predefined macro cannot be found, the compiler will look for an environment variable of the same name. The '%' characters are not part of the macro name. Note that **"" <text>" SETMACRO <name>** can be placed on the cross compiler command-line and thus you can specify a root directory in a Windows short-cut.

If you need macros that are common to both the IDE and the cross compiler, use environment variables.

Compiler macros

The compiler can be used independently of the IDE. However, the IDE supplies the compiler with a set of text macros that define the names of the various source directories. Some installers provide the IDE with the names of the compiler directories via a Forth source file, DIRS.FTH, which is always placed into the <root> directory. When the user invokes a cross-compiler, the IDE includes this file before your control file on the compiler's command-line. Hence, you don't have to do anything to obtain this information.

If the IDE is not used, then this information will have to be provided to the compiler some other way. There are basically three ways of doing this:

- Include the file DIRS.FTH (if it exists) yourself via the compiler's command-line, perhaps in a short cut, or by including it in the control file.
- Define the required set of text macros as environment variables - just take DIRS.FTH and use a text editor to convert it into a number of SET commands. In this case, no "alien" source files need be included before your control file. Note that the names are fixed, so this approach will only work if you are using a single cross-compiler.
- Include the relevant macros in your control file.

The text macros required by the standard control files are as follows:

Macro Name	Default Setting
Always present:	
CpuDIR	<root>\cpu
CommonDIR	<root>\common
May be present:	
DIRROMHOM	<root>\cpu
DIRROMCFG	<root>\cpu\configs
DIRROMDRV	<root>\cpu\drivers
DIRROMCOM	<root>\common
DIRROMFTH	<root>\common\romforth
DIRROMEXA	<root>\examples

Table 1: Standard control file text macros

Directory structures

For reference, the directory structure of the cross-compiler is listed below with a description of each directory's contents.

Directory	Contains
<root>	Installer files and DIRS.FTH
CPU (e.g. 8051)	CPU-specific kernel source files
CONFIGS	Example control source files
DRIVERS	Serial and other driver source files
COMPILER	Compiler .EXE and error messages files
CPU (e.g. 8051)	CPU specific cross compiler source code
COMMON	Cross compiler common source code
PFW	Host Forth for the cross compiler
DOC	Help files and other documentation
COMMON	Non CPU-specific kernel source files
DRIVERS	Serial and LED driver source files
EXAMPLES	Chip-independent test and example source
ROMFORTH	Chip-independent ROMFORTH source files
AIDE	AIDE executables, data, configuration files

Table 2: Cross-compiler directory structure in detail

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

4 Generating a standalone Forth

This chapter describes how to generate a ROMmable target ANS Forth for your target board. It 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

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 ROM 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.

The control file

The control file contains all the details of your board that the cross-compiler needs to know. These include:

- the memory map of your board
- whether you wish a log to be displayed
- the number of tasks in your system
- the clock rate of your board

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.

The memory map

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

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

Setting the start and end of ROM

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 ROM used for code, rom-end is the address of the end of ROM 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 $, e.g

```
$0100
```

This area also contains any data defined by **CDATA** during the cross-compilation. This directive is discussed elsewhere in the manual.

Setting the start and end of initialised RAM

The start and end of the initialised RAM area is defined by using the compiler directive **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 ram-start and ram-end are, by default, in decimal, but can be entered in hex by preceding them with a $. <name> is not actually used but must be stated.

The initialised RAM area contains any data defined by **VARIABLE** or **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.

Setting the start and end of uninitialised RAM

The start and end of the uninitialised RAM area is defined by using the compiler directive **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 $. **<name>** is not actually used but must be stated.

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

Setting the compilation areas

The compiler must be instructed to compile into the areas defined by **SECTION**. Therefore, after the memory map is defined you must code:

```
<name1>
<name2>
<name3>
```

where <name1> is the name of the ROM area, <name2> is the initiialised RAM area, and <name3> is the uninitialised RAM area.

An example

If your target board has a memory map as in the figure above, your control file should be modified so that it reads:

```
$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 and 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.

Modifying the serial line drivers

Your target board 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 ROM\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 need to write all the words required to:

- initialise the UART
- send a character
- receive a character
- test if a character has been received

All four words will normally be Forth **CODE** definitions. This is required so that the send and receive words are as fast as possible. Example serial line drivers in the files ROM\DRIVERS can be used as a template. As with the control file it is normally easier to modify an existing serial line driver file rather than creating your own from scratch.

Two types of serial driver can be written:

- interrupt driven
- polled

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.

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.

Initialising the serial line

The word **INIT-SER** must perform all the UART initialisation required. This includes setting:

- the baud rate
- any handshaking required
- the number of data bits

- the number of stop bits
- the parity to be used

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.

Sending a character to the host

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

- waits for the transmit line to become available
- transmits a character to the host
- increments the **USER** variable **OUT**

The method used can be either a polled or interrupt driven driver but must be called **(EMIT)**. Once **(EMIT)** is written, it must be assigned to the defered word **EMIT**. The stack effect of **(EMIT)** is:

```
(EMIT)    \ char -- ; send char to host
```

Receiving a character from the host

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

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

The method used can be polled or interrupt driven but the word must be called **(KEY)**. Once **(KEY)** has been written, it must be assigned to the deferred word **KEY**. The stack effect of **(KEY)** is:

```
(KEY)     \ -- char ; wait for char to be received
```

Detecting a received character

The target needs **(KEY?)** to detect if a character has been received. This can be used as part of **(KEY)**. **(KEY?)** needs to:

- return true on the Forth stack if a character is available (-1)
- return false on the Forth stack if a character is not available (0)

Once **(KEY?)** is written, it must be assigned to the defered word **KEY**. The stack effect of **(KEY?)** is:

```
(KEY?)     \ -- t/f ; true if character received
```

Setting up the system

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

Setting up the hardware

To generate an interactive Forth target you need:

- A PC
- A serial cable
- A target
- An EPROM emulator or programmer

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

Setting up the software

To compile source code that generates a standalone Forth target, you need to 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 APP and APPDIR macros so that the cross-compiler knows where your files are located. This can be done from within the IDE.

Cross-compiling

Now the hardware and software have been setup, you can now cross-compile the source code to generate an executable image.

Creating an image

To cross-compile the source, ensure that the cross-compiler macros are set up correctly and point to your control file. Press the cross-compile toolbar button to begin compilation. The compiler displays its sign-on message and then compiles the source code.

The cross-compile log

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 Table 3: Key to cross-compiler log.

The output can be sent to a file or to the printer. Note that turning the log on to the screen slows down the compiler considerably, but is useful when you have a lot of 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.

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 the 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 **LOG** with **NO-LOG** in the control file.

Sending the log to a file

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

```
FILE: <name>
```

where **<name>** is the filename to generate. This directive must be placed before the command **CROSS-COMPILE**. A macro is provided that can be set from within the IDE to turn the log on or off.

Sending the log to a printer

The cross-compiler will send the log to a printer. To do this, use:

```
PRN:
```

before the command **CROSS-COMPILE**.

The compilation summary

Once the cross-compiler has finished cross-compiling the 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, the initialisation code copies this table into the initialised RAM areas. These initial values of variables will be modified in RAM when you store into a variable.

Code	Compiled type	Code	Compiled type
VR	Variable	FV	Floating-point variable
CN	Constant	FC	Floating-point constant
LB	Label	FA	Floating-point array
:	Colon definition	EQ	Equate
CD	Code definition	CR	Child of **`CREATE … DOES>`**
DF	Deferred word	US	**`USER`** variable
VC	Vocabulary		

Table 3: Key to cross-compiler log

The created image

The image created by the cross-compiler is a straight binary executable. It can be downloaded to a suitable EPROM emulator or programmer. The file has the name given when defining the memory map using the compiler directive **`SECTION`**. It has the extension .IMG, which cannot be changed.

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.

Downloading the compiled image

Once the source code has been compiled the image needs to be downloaded to an EPROM emulator or programmer.

Downloading to a LeBurg EPROM emulator

The MPE cross-compiler supports direct compilation into the LeBurg EPROM emulators (series 2 onwards). If you have a LeBurg EPROM emulator, you can make a short cut to the EPEM4.COM program by adding an external tool to AIDE.

Downloading to a different emulator or programmer

The binary image can be downloaded to any EPROM emulator as long as the emulator's software supports binary image files.

Running the target Forth

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

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 ‘PowerTerm’.

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 if no reset is on the board, turn the board's power off and on again.

The sign-on

Once the board has been reset, the target should sign-on. You should see a message similar to that in figure below. The version number and the number of bytes free will depend on your system. You now should have a working Forth. If the target did not show the message, then you may have a problem with:

- the serial line drivers

- the memory map definition
- your target board
- your EPROM emulator/programmer
- Direct port access under Windows NT/2000

Each of these should be checked.

```
MPE Hitachi H8/300H ANS ROM PowerForth v3.00
16383 bytes free

ok
```

Figure 2: Target sign-on

The 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.

The 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'.

Your target board

It is always necessary to check the obvious. Is the serial line connected? Has your target board got power? EPROMs/RAM plugged in correctly? Are jumpers set correctly?

Your EPROM emulator/programmer

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

Testing the Forth - an example

Once the Forth has signed-on, you need to test that it's 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.

Cross-compiling an application

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

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
```

This file contains some useful words for cross-compiled targets, but is not essential.

Running your application

To compile the application you need to:

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

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

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. Your application does not necessarily need to be interactive, so the compiler directive **NO-HEADS** can be used. This removes all the word headers, so making the final image more compact.

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 in Figure 3 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 .
    1+
  AGAIN                   \ Application never ends
;

MAKE-TURNKEY RUN
```

Figure 3: Example turnkey application

5 Generating an Umbilical target

This chapter describes how to generate an Umbilical Forth target for your target board. It 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

Requirements for Umbilical Forth

To generate an interactive target you require:

- a LeBurg EPROM emulator for fast download
- interrupt driven serial drivers

If you want to define new words interactively, you need to use a LeBurg EPROM emulator. When the cross-compiler generates code, it will write to the emulator. This normally `upsets' the processor so the processor should be put to sleep while waiting for serial communications. Once the UART becomes available, the processor will be taken out of sleep mode and will continue processing.

Is your target already supported?

The cross-compiler ships with at least one, usually more, control files for various commercial target boards. By using one of these control files, the installation of an Umbilical Forth target for your board will be greatly simplified.

If you do not have one of these boards you will have to create a control file and serial line drivers for your board.

The control file

The control file contains all the details of your board that the cross-compiler needs to know. These include:

- the memory map of your board
- whether you wish a log to be displayed

- the clock rate of your board's crystal

As well as containing configuration information, the control file contains 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.

Creating a control file

To create a new control file, copy an existing one and then modify it to match your board. This is normally easier than generating one from scratch. Example control files are in the directory CHIP\CONFIGS.

The memory map

The memory map describes the addresses where ROM and RAM start and end in your target system. The word **SECTION** defines an area of memory that will become the current memory area of its type when used. Three memory types are defined:

- **CDATA** - Code – where code is compiled
- **IDATA** - Initialised data – where data that must be initialised is placed
- **UDATA** - data that should not be initialised such as battery backed RAM or EEPROM.

The directives **CDATA IDATA** and **UDATA** select which type of memory the Forth words below affect:

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

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 board.

The memory map is described in three parts:

- the start and end of code ROM
- the start and end of initialised RAM
- the start and end of uninitialised RAM

Setting the start and end of ROM

The start and end of ROM are defined by using the compiler directive **SECTION** used in the form:

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

where rom-start is the address of the start of ROM, rom-end is the address of the end of ROM 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 by a $.

This area also contains any data defined by **CDATA** during the cross-compilation. This directive is discussed elsewhere in the manual.

Setting the start and end of initialised RAM

The start and end of initialised RAM are defined by using the compiler directive **SECTION** used in the form:

```
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 ram-start and ram-end are, by default, in decimal, but can be entered in hex by preceding then by a $.

The initialised RAM area contains any data defined by **VARIABLE** or **VALUE** or **IDATA** during the cross-compilation. These directives are discussed elsewhere in this manual.

Setting the start and end of uninitialised RAM

The start and end of this RAM is defined by using the compiler directive **SECTION**, used in the form:

```
ram-start ram-end UDATA 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 ram-start and ram-end are, by default, in decimal, but can be entered in hex by preceding them by a $. **<name>** is not actually used but must be stated.

The unitialised RAM areas contain data space allocated by **BUFFER:** or **UDATA** during the cross-compilation.

Setting the compilation areas

The compiler must be instructed to compile into the pages defined by **SECTION**. Therefore, after the memory map is defined you must code:

```
<name1> <name2> xDATA
```

where <name1> is the name of the ROM area, <name2> is the RAM area, and xDATA is one of **CDATA IDATA** and **UDATA** (normally **CDATA**).

An example

If your target board has a memory map as in the figure, your control file should be modified so that it reads,

```
$00000 $07FFF CDATA SECTION Kern
$08000 $0FFFF IDATA SECTION Kern-data
$10000 $1FFFF UDATA SECTION Kern-uram
Kern Kern-data Kern-uram CDATA
```

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

Modifying the serial line drivers

Your target board 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 in the directory CHIP\DRIVERS. Look here first, as new drivers may have been added since the manual was written.

For interactive compilation with Umbilical Forth through the EPROM emulator, the processor must be put to sleep while compilation is in progress. In practice, this means that the processor is put to sleep while waiting for a character, and is restarted by the receiver interrupt. If the processor cannot be put to sleep, interactive compilation can be achieved by placing the receiver-polling loop in RAM, so that the EPROM is not used while the CPU is polling for keyboard input.

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

- Initialise the UART
- Send a character
- Receive a character
- Test if a character has been received
- Export the names to the link driver

All four words will normally be Forth **CODE** definitions. This is required so that the send and receive words are as fast as possible. Example serial line drivers in the files in the DRIVERS directory can be used as a template. As with the control file it is normally

easier to modify an existing serial line driver file rather than creating your own from scratch.

Two types of serial driver can be written:

- interrupt driven
- polled

Interrupt driven

An interrupt driven serial line can only be used if the UART generates interrupt signals when characters are received. For interactive use, the processor must also be capable of being put to sleep. 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.

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.

Initialising the serial line

The word **`INIT-SER`** must perform all the UART initialisation required. This includes setting:

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

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.

Sending a character to the host

The target code needs to be able to send a character to the host for display. 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 **(EMIT)** is:

```
(EMIT)    \ char -- ; send char to host
```

Receiving a character from the host

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

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

The method used can be polled or interrupt driven. Once **(KEY)** has been written, it must be assigned to the deferred word **KEY**. The stack effect of **(KEY)** is:

```
(KEY)     \ -- char ; wait for char to be received
```

Note that the version of **KEY** that you export to the link driver must not call the multitasker, and must put the CPU to sleep if an EPROM emulator is being used.

Detecting a received character

The target needs **(KEY?)** to detect if a character has been received. This can be used as part of **(KEY)**. **(KEY?)** needs to:

- return true on the Forth stack if a character is available (-1)
- return false on the Forth stack if a character is not available (0)

Once **(KEY?)** is written, it must be assigned to the defered word **KEY?**. The stack effect of **(KEY?)** is:

```
(KEY?)     \ -- t/f ; true if character received
```

Exporting the names

The Umbilical Forth link uses standard names for its link drivers. These must be associated with your words. Note that the version of **KEY** that you export to the link driver must not call the multitasker, and must put the CPU to sleep if an EPROM emulator is being used. The standard way to export the names is to use the **SYNONYM <new> <old>** notation, which creates a new name for an existing word. This is usually done in the control file just before compiling the files MESSAGES.FTH and TARGEND.FTH.

```
Synonym wait-byte  (serkey)
Synonym send-byte  (seremit)
Synonym Wait-Byte? (serkey?)
Synonym Init-XTL   Init-Ser
```

Setting up the system

Setting up the hardware

To generate an interactive Forth target you need:

- A PC
- A serial line
- A target board
- An EPROM emulator or programmer

Your PC needs to have at least one serial line port for connecting to the target board, so making the Forth interactive. The default serial port is set in the umbilical control file. The PowerTerm terminal emulator defaults to COM1.

If the Leburg EPROM emulator is being used, you will also need to connect the emulator to the digital I/O card installed in your PC.

Setting up the software

To compile source code that generates a standalone Forth target, you need to 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 APP and APPDIR macros so that the cross-compiler knows where your files are located. This can be done from within the IDE.

Cross-compiling

Now the hardware and software have been setup, you can now cross-compile the source code which is automatically compiled down to your EPROM emulator.

Creating an image

To cross-compile the source, ensure that the cross-compiler macros are set up correctly and point to your control file. Press the cross-compile toolbar button to begin compilation. The compiler displays its sign-on message and then compiles the source code.

The cross-compile log

Following the compiler sign-on you 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 Table 4.

The output can be sent to a file or to the printer. Note that turning the log on to the screen slows down the compiler considerably, but is useful when you have a lot of 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.

Code	Compiled type	Code	Compiled type
VR	Variable	FV	Floating-point variable
CN	Constant	FC	Floating-point constant
LB	Label	FA	Floating-point array
:	Colon definition	EQ	Equate
CD	Code definition	CR	Child of **CREATE … DOES>**
DF	Deferred word	US	**USER** variable
VC	Vocabulary		

Table 4: Key to cross-compiler log

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 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 **LOG** with **NO-LOG** in the control file.

Sending the log to a file

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

```
FILE: <name>
```

where **<name>** is the filename to generate. This directive must be placed before the command **CROSS-COMPILE**. A macro is provided that can be set from within the IDE to turn the log on or off.

Sending the log to a printer

The cross-compiler will send the log to a printer. To do this, use:

```
PRN:
```

before the command **CROSS-COMPILE**.

The compilation summary

Once the cross-compiler has finished cross-compiling the 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, the initialisation code copies this table into the initialised RAM areas. These initial values of variables will be modified in RAM when you store into a variable.

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.

Downloading the compiled image

Once the source code has been compiled the image needs to be downloaded to an EPROM emulator or programmer.

Downloading to a LeBurg EPROM emulator

The MPE cross-compiler supports direct compilation into the LeBurg EPROM emulators (series 2 onwards). If you have a LeBurg EPROM emulator, you can make a short cut to the EPEM4.COM program by adding an external tool to AIDE.

Downloading to a different emulator or programmer

The binary image can be downloaded to any EPROM emulator as long as the emulator's software supports binary image files.

Running the target Forth

The image generated by the compiler has been downloaded to the target, it is ready to be reset and the Forth tested. If you are not using the EPROM emulator you must transfer the code to the target yourself. If the target is using an Umbilical Forth monitor in EPROM, you will be able to download the code to the target across the target link during the Umbilical Forth startup procedure.

Resetting the target board

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.

The sign-on

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 create a definition. Therefore if you type:

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

FORTH-TEST
```

This should display:

```
HELLO
```

followed by the **ok** prompt. Note that if you have compiled the multitasker and are using an EPROM emulator, it must be disabled with **SINGLE** before any compilation takes place. This has to be done because the tasker never permits the CPU to go to sleep.

If you didn't get this response, then you may have a problem with:

- the serial line drivers
- the memory map definition
- your target board
- your serial line

- your EPROM emulator/programmer
- Direct port access under Windows NT/2000.
- multitasking being enabled and an EPROM emulator is in use.

Each of these should be checked.

Cross-compiling an application

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

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:

```
ALL FROM-FILE <name>
```

To compile your application files you add them to the end of the list.

Running your application

To compile the application you need to:

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

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

Debugging and developing your application

Forth is an interactive language, use this to your advantage by writing small sections of code and testing as you go. Within Umbilical Forth you can compile one file at a time, or enter definitions at the keyboard. They will be compiled and immediately downloaded to the target, where they can be tested, just as with any other interactive Forth. You can use the assembler, and refer to **LABEL**s and **EQU**ates by name.

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. Your application does not necessarily need to be interactive, so the compiler directive **NO-HEADS** can be used. This removes all the word headers, so making the final image more compact.

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 in the figure generates a simple turnkey application when cross-compiled. If you require the use of serial communications or the multitasker, 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.

Also turn off the **EQU**ate **UMBILICAL?** in the control file to remove the Umbilical Forth drivers from the code.

```
: RUN
  (INIT)          \ Init. system (Mandatory!)
  INIT-SER                \ Init. the serial line
  INIT-MULTI              \ If multitasking
  0                       \ counter
  BEGIN
    CR " Hello world!" dup .
    1+
  AGAIN                   \ Application never ends
;

MAKE-TURNKEY RUN
```

Figure 4: Umbilical Forth turnkey application

Debugging the serial link

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. Character 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
```

Using other link drivers

The other 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 which handles programming of the CPU code space. New drivers can be installed

at any time, and users wishing to write a new driver should 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 CPU32 cores such as the 68332 – Umbilical link plus RAM and limited EPROM/Flash drivers

JTAG for ARM cores – Umbilical link plus limited RAM and EPROM/Flash drivers.

JTAG for MSP430 cores – Umbilical link plus limited RAM and EPROM/Flash drivers.

Note that if you are using Windows NT/2000 or any other version of Windows that treats direct port I/O as a privileged instruction, you must install the 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 Optimising your 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 the MPE Development system that helps you with this aim.

Reducing the size of your image

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

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

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.

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.

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.

Factoring your code

When writing in Forth, code should be reused as much as possible. By reusing code, your target image can be reduced greatly. The smaller 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.

Removing excess code

During development, debug and test code is often inserted into the source. This code is easily left and forgotten about. By stripping out this excess code you can gain more space in the EPROM. 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.

Using equates instead of constants

An equate is a constant that just resides within the cross-compiler. It therefore cannot be referenced when interactively debugging on your target system. The actual value of the equate is compiled `in-line' instead of refering to a constant. Therefore you can save some space on the target board for each constant defined but sacrifice some interactivity. This only 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.

Defining an equate

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.

Using an equate

An equate is used in the same way as a constant, by stating its name.

```
0100 EQU ADDRESS
  ADDRESS 4 + EQU ADDRESS2

: SOME-WORD       \ --
  ... ADDRESS ...
  ... ADDRESS2 ...
;
```

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 so 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 (--)**.

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 an interactive Forth in a very compact size (the Umbilical Forth kernel is about 2.5k for 16 bit targets, and 4k for 32 bit targets). The kernel does not contain all the words in the standalone target, so you might have to write a few words to get your code to compile or copy some code from the standalone target Forth. For more details see the chapters on Generating an Umbilical Forth target.

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
- words containing a lot of stack manipulation words (**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 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.

7 Generic I/O

About Generic I/O

Generic I/O allows the Forth words **KEY KEY? EMIT TYPE CR** to use any I/O device. The user variables **IPVEC** and **OPVEC** contain pointers to the current device structure. For example, 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, and because **IPVEC** and **OPVEC** are **USER** variables, 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 perorm the following basic functions.

```
cell     KEY action
cell     KEY? Action
cell     EMIT action
cell     TYPE action
cell     CR action
cell     TYPEC action      Harvard targets (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 **EMIT**s. The output functions update the **USER** variable **OUT** before calling the action.

Creating a new device

When creating a new device driver, make an array that contains the pointers. The following code is taken from the 8051 serial driver.

```
cdata    \ this table goes in CODE space
create SerConsole \ -- addr ; OUT managed by upper driver
tasking? [if]
  ' (mserkey) ,   \ -- char ; schedule, receive char
[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 CDATA
                           \   for Harvard targets only
```

The generic I/O dispatcher 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. This is usually called **(MSERKEY)** in the source code.

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.

8 Multitasker

The multitasker supplied with the MPE development system can greatly simplify complex tasks by breaking 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.

Initialising the multitasker

The multitasker needs to be initialised before use. At compile time the cross-compiler must be told the total number of tasks that your system requires and at run-time, all the tasks must be initialised.

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 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     \ 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
```

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**. **MULTI** starts the scheduler so new tasks can be added.

Stopping the multitasker

To stop the multitasker, use **SINGLE**.

Writing a task

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

Using the scheduler

The multitasker is software scheduled. This means that each task relinquishes control back to the scheduler when it's ready. This is different from a pre-emptive scheduler where the scheduler interrupts a task. A word is supplied so that a task can relinquish control back to the scheduler, **PAUSE**.

Using PAUSE

The word **PAUSE** passes control back to the scheduler, which executes all the other tasks once, then returns back to this task

An example

An example task is shown below. The task is an endless loop with the word **WAIT** embedded in it. When the word **WAIT** is executed, the scheduler reschedules to the next task. The scheduler will not run this task until it has run all other tasks 1000 times. Each time the task is executed, it will emit a beep.

```
: WAIT            \ n -- ; wait for n iterations
  0 ?DO  PAUSE  LOOP
;

: ACT1ON1                 \ – ; An example task
  BEGIN                   \ Start an endless loop
    7 EMIT                \ Produce a beep
    1000 WAIT             \ Reschedule 1000 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.

Task dependent variables

An area of memory is set aside for each task. This memory contains user variables which contain task specific data. For example, the current base is normally a user variable as it can vary from task to task.

Defining a user variable

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 v6.1 kernel code relies on **+USER** and new application code should use **+USER** in preference to **USER**.

Using a user variable

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

Tasks and local variables

Local variables are held on the return stack. 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 **EQU**ate for the return stack size in the control file.

Initialising a task

A task needs to be initialised in order to be run.

```
‘ ACTION1 TASK1 INITIATE
```

where **ACTION1** is to be the action of the task and **TASK1** is the task identifier

The task identifier is used to control the task. Tasks defined by **TASK <name>** return a task identifier when **<name>** is executed.

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.

Starting a task

A task is started by activating it. To activate 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.

Stopping a task

A task may be temporarily suspended. A task may also stop itself.

Temporarily stopping a task

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

```
<task> HALT
```

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

```
<task> RESTART
```

where <task> is the task to restart.

Stopping the current task

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

```
STOP
```

Handling messages

An essential feature of the multitasker is the ability to send and receive messages between tasks.

Sending a message

To send a message to another task, use the word **SEND-MESSAGE**. **SEND-MESSAGE** is used in the form:

```
message task SEND-MESSAGE
```

where message is a 32-bit 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.

Receiving a message

To receive a message, use **GET-MESSAGE**. **GET-MESSAGE** suspends the task until a message arrives. When a message is received the task is re-activated and the sending task and the data is returned.

Creating events

Events are analogous to interrupts. Whereas interrupts happen on hardware signals, events happen under software control.

Writing an event

An event is a normal Forth word. An event is associated to a task so that when the event is triggered, the task is activated. Therefore, an event is usually used as initialisation for a task.

Initialising an event

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

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

where **EVENT1** is your event handler and n is the task number of the task that it is to be associated with.

Triggering an event

There are two ways of triggering an event:

- using SET-EVENT
- setting a bit in the status word

Using SET-EVENT

SET-EVENT is a word that sets an event flag for 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.

Setting a bit in the status word.

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 on the status byte.

This mechanism can easily be used by interrupt code written in assembler to signal that an interrupt has taken place, and that consequent processing should start.

Clearing an event

To stop an event handler being run, use **CLEAR-EVENT**.

Interrupts and critical sections

Sometimes the multitasker has to be inhibited so that other tasks are not run during critical operations that 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 the default cooperative mode, great care must be taken. Flags must 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 documented in the CPU specific section of the manual.

DI -- ; disable interrupts
“d-i”

Globally disable interrupts.

EI -- ; enable interrupts
“e-i”

Globally enable interrupts

SAVE-INT -- x ; save interrupt status
“save-int”

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 the new constructs **[I** and **I]**.

RESTORE-INT x -- ; restore interrupt status
"restore-int"

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 the new constructs **[I** and **I]**.

[I R: -- x ; save interrupt staus, disable inteerupts
"bracket-i"

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.

I] R: ccr -- ; restore CCR from return stack
"i-bracket"

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.

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 TCB 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
    0 ,  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 it is available for use again, 0 indicating in use by a task

```
: signal        \ addr -- ; increment counter,
                \ so making it available
  save-int                          \ must be interrupt safe
  1 over +!  cell+ off     \ inc. counter, release
  restore-int
;
```

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 **SIGNAL**s that it has finished with the resource.

```
: request       \ sem -- ; get access to semaphore
  >r
  begin
    save-int  r@ @ 0=              \ n.b test and set
   while
    restore-int  pause     \ operations must be
  repeat                   \ non-interruptible
  -1 r@ +!                         \ got it, decrement counter
  self r> cell+ !          \ mark resource as mine
  restore-int                      \ re-enable interrupts
;
```

The multitasker internals

A multitasker tries to simulate many processors with just one processor. It works by rapidly switching between each task. 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.

The scheduler's data structure

The Forth multitasker creates a task control block for each task. The task control block (TCB) is a data structure which contains information relevant to a task (see below). The status byte (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 next's 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 task's event handler	Cell	10/20

Table 5: 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 6: Task status byte

A simple example

The following example is a simple demonstration of the multitasker. Its simple role is to display a hash (#) every so often, but leaving the forground Forth running. To use the multitasker you must cross-compile the file MULTI*.FTH into your target.

Defining a simple 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
  [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 ...
;
```

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.

Activating the example task

To activate (run) the example task, type:

TASK TASK1

```
ASSIGN ACTION1 TASK1 INITIATE
```

This will activate **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 notice that the Forth is still running. After a few seconds another hash will appear. This is the example task working in the background.

Controlling the example task

The example task can be controlled in several ways:

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

Changing the rate of hashes

Changing the variable **DELAY** can 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. Therefore the rate of displaying hashes halves.

Halting the example task

Typing the task's number followed by **HALT** halts the task:

```
TASK1 HALT
```

You notice that the hashes are not displayed.

Restarting the halted task

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

```
TASK1 RESTART
```

You notice that the hashes are displayed again.

Restarting the task from scratch

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

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

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

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, be careful to make sure that the multitasker is 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 an EPROM emulator will crash the running target.

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. The single chip version of the multitasker does not include event handling, messages, or semaphores. Details of this multitasker are provided in the CPU specific section of the manual.

Converting to the v6.x multitasker

Configuration

The 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
```

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 task's **USER** area, and the task control information occupies the start of the user area. This address is referred to as a task identifier.

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 provided by the new word **MS**, which is supplied by the code in TIMEBASE.FTH or another timing system. **MS** waits for the specified number of milliseconds.

```
MS              \ ms --
```

INITIATE and ACTIVATE

The word **ACTIVATE** has been replaced by the word **INITIATE** and **DEACTIVATE** has been replaced by **TERMINATE**.

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

?EVENT

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

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 --
"clear-event-run"

Clears the event run flag for the current task. This is bit 4 in the task status byte.

DI -- ; disable interrupts
“d-i”

Globally disable interrupts.

EI -- ; enable interrupts
“e-i”

Globally enable interrupts

EVENT? -- t/f
"event-query"

Returns true if the event-triggered bit has been set in the current task's status byte.

GET-MESSAGE -- message task
"get-message"

Waits for a message and returns the message and the sending task.

HALT task --
"halt"

Halts the task whose number is given. Do not halt task **MAIN**. Halting a task prevents it responding to messages or events.

INIT-MULTI --
"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 --
"initiate"

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 –
“m-s”

Waits for at least ms milliseconds, the exact time depending on the granularity of the timer.

MSG? task -- t/f
"message-query"

Returns true if the task is holding a message, and is therefore not free to receive another one.

MULTI --
"multi"

Turns the multi-tasker on, by clearing the bit in the TASK# byte in RAM that inhibits the scheduler.

PAUSE --
"pause"

Waits for one iteration of the scheduler.

RESTART task --
"restart"

Restarts a task that was halted by **HALT** or **WAIT**. Unlike **INITATE**, the task resumes where it left off.

RESTORE-INT sr --
"restore-int"

Restore the interrupt enable state previously saved by **SAVE-INT**.

SAVE-INT -- sr
"save-int"

Saves the current state of the interrupt enable, and disables interrupts. See **RESTORE-INT**.

SELF -- task
"self"

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 --
"send-message"

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 --
"single"

Turns off the multi-tasker by setting the scheduler disable bit

STATUS -- n
"status"

Returns the task status byte 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 --
“stop”

Halt the current task until it is **RESTART**ed or **TERMINATE**d

TERMINATE task –
“terminate”

Remove a task from thelist of active tasks and reschedule.

TO-EVENT cfa task --
"to-event"

Sets the CFA 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 --
"wait-event-or-message"

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.

[I R: -- x ; save interrupt staus, disable inteerupts
“bracket-i”

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.

I] R: ccr -- ; restore CCR from return stack
“i-bracket”

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.

9 TIMEBASE

Periodic timers and TIMEBASE

The TIMEBASE code provides a timer system that allows many timers to be defined, all slaved from a single periodic interrupt. The Forth words in the user accessible group documented below are compatible with VFX Forth. This code assumes the presence of a global value or word TICKS which returns a time value incremented in milliseconds. The timebase is approximate, and granularity and jitter are affected by the timer ISR and the time taken by your own code to execute. By default, the timer is set to run every 10ms. The main source code is in the the file TIMEBASE.FTH, and requires a CPU dependent clock interrupt routine which provides **TICKS** and clock interrupt initialisation.

The timer chain is built using a buffer area, and two chain pointers. Each timer is linked either into the free timer chain, or into the active timer chain.

All time periods are in milliseconds. Note that on a 32 bit system such as VFX Forth or an ARM, 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.

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 ms -- timerid/0 ; runs xt once
EVERY             \ xt ms -- timerid/0 ; runs xt every 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.

```
start-timers

: t      \ -- ; will run every 2 seconds
  [char] * emit
;

' t 2 seconds every        \ returns timer id, TSTOP to stop
```

The item on stack is a timer id (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. Timer handles are not addresses, and are integers allocated in sequence. On a 16 bit system, timer handle numbers will only be recycled every 2^16 allocations. If the handle is already in use, it will not be reallocated.

Considerations when using timers

All timers are executed within a single interrupt, and so all timer action words share a common user area. This has some impact on timer action words. Since you do not know in which order timer action words are executed, you must set up any **USER** variables such as **BASE** that you may use, either directly or indirectly.

The interrupt that handles all the timers does not set **IPVEC** and **OPVEC** to a default value. If you are going to use Forth I/O words such as **EMIT** and **TYPE** within a timer action, you MUST set **IPVEC** and **OPVEC** before using the I/O. For the sake of other timer action routines that may still be using default I/O, it is polite to save and restore **IPVEC** and **OPVEC** in your timer action words.

Do not worry about calling **TSTOP** with a timerid that has already been executed and removed from the active timer chain; if **TSTOP** cannot find the timer, it will ignore the request.

Under some conditions, the execution time of all the timer routines may be longer than the requested period of the timer. Try to avoid this situation! In addition, the timer interrupt may be subject to jitter, and because the timer routines are executed in sequence, the start of a timer routine will be dependent on the execution time of the routines before it. If this is serious, code is available from MPE that measures and saves the actual period rather than the nominal period. However, this increases the timer despatch time for each timer.

Implementation issues

The following discussion is only relevant if you want to modify this code. Functionally equivalent code is provided with MPE's VFX Forth systems. In the Windows environment, timer interrupts are implemented by callbacks and critical sections.

By default, the word **DO-TIMERS** is run from within the periodic timer interrupt. If interrupts are not re-enabled after resetting the timer interrupt, you may have latency issues if a number of timers is used, or if the timer routines take a considerable time. In this case, it would be better to set up the timer routine to **RESTART** a task which calls **DO-TIMERS**, e.g.

```
: TIMER-TASK    \ --
  <initialise>
  BEGIN
    DO-TIMERS STOP
  AGAIN
;
```

Such a strategy also permits you to use a fast interrupt, say 1ms, for the clock, and to trigger the **TIMER-TASK** every say 32 ms.

Timebase glossary

#TIMERS -- n ; maximum number of timers
"hash-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
"do-timers"

The central timer despatch routine.

AFTER xt period -- timerid/0 ; xt is executed once
"after"

Starts a timer that executes once after the given period. A timer handle is returned if the timer could be started, otherwise 0 is returned.

EVERY xt period -- timerid/0 ; xt is executed periodically
"every"

Starts a timer that executes every given period. A timer handle 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 --
"t-stop"

Removes the given timer from the active list chain.

PAUSE -- ; multitasker hook
"pause"

Allows the sytem multitasker to get a look in. Under Windows this also allows the message queue to be handled. This word will be a dummy if the equate **TASKING?** is zero, otherwise it will be a call to the routine in the MULTIxx file.

TICKS -- n
"ticks"

Get the current clock value in milliseconds. Note that this routine must be defined in the CPU dependent interrupt handler code file.

LATER n -- n'
"later"

Generates the timebase value for termination in n millseconds time.

EXPIRED n -- flag ; true if timed out
"expired"

Flag is returned true if the timebase value n has timed out. Calls **PAUSE**.

TIMEDOUT? \ n -- flag ; true if timed out
"timed-out-query"

Flag is returned true if the timebase value n has timed out. **TIMEDOUT?** Does not call **PAUSE**, so **TIMEDOUT?** can be used in interrupts, winprocs and callbacks. In particular, **TIMEDOUT?** should be used rather than **EXPIRED** inside timer action words to reduce timer jitter.

MS n --
"m-s"

Waits for n milliseconds. Uses **PAUSE**.

START-TIMERS -- ; Start internal time clock
"start-timers"

START-TIMERS must be provided by the CPU dependent code. It initialises the periodic clock and starts it.

STOP-TIMERS -- ; disable timer interrupts
"stop-timers"

STOP-TIMERS must be provided by the CPU dependent code. It turns off the periodic timer.

10 Heap and memory allocation

ANS Standard

The supplied source code implements the ANS Forth memory allocation word set with extensions.

Source code

The source code is in the files COMMON\HEAP16.FTH (for 16 bit targets) and COMMON\HEAP32 (for 32 bit targets).

The heap is allocated from a predefined section of memory using the equate SIZEOFHEAP to produce a statuc buffer **STARTOFHEAP**. 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**.

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

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

HEAP16

The HEAP16 code is optimised for code density.

The heap is controlled using two cells per block for 16 bit targets. This information is used in three parts:

#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.

HEAP32

The HEAP32 code is optimised for performance and is usually used with the VFX code generator.

The HEAP32 code uses a single 32 bit cell for control, the high byte containing the control information, where $EE = End, $FF = Free, $AA = Allocated. Others may be added later for type management. The bottom 24 bits of the flag are contain the size of the allocated block's data area..

Common

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

Glossary

The following glossary doesn not include all the factors used in the code. If you are interested in the implementation, please read the source code.

ALLOCATE #bytes -- addr status
"allocate"

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
"free"

Attempt to free a heap block. Status=0 for success.

INIT-HEAP --
"init-heap"

Initialise the heap. If you don.t the system will surely crash!

RESIZE addr1 size -- addr2 status
"resize"

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.

SIZE addr -- currsize | -1
"size"

Return the size of an allocated block or -1 if there's an error. Note that the size returned is the actual size of the data area, not the requested size.

.HEAP -- ; display heap info
"dot-heap"

Walk the heap displaying block information.

HEAPOK? -- t/f ; check heap
“heap-o-k-query”

Walk the heap and return **TRUE** if the heap is "well". If the heap is sick, diagnostic information will be displayed.

11 Software floating-point

Although most embedded applications only require integer arithmetic, some do require floating-point. Therefore software floating-point is supplied with the cross-compiler and the target Forth. The target floating point wordset is not fully ANS compliant, but satisfies the needs of embedded systems without undue complexity. The Forth data stack and the floating point stack are the same. The floating point data storage format is not IEEE format, but is optimised for performance on small controllers. If you need a separate floating point stack or IEEE format storage, please contact MPE. Any variations in the implementation will be documented in the target specific section of the manual.

The cross-compiler has a more limited floating-point support than the target, this means that some words are avaliable within colon definitions, but not outside them.

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.

Entering floating-point numbers

Floating-point numbers can be entered in two forms, 1.234 and 0.1234e1

Floating-point numbers are compiled as literal numbers when in a colon definition and placed on the cross-compiler's stack when outside a definition.

The form of floating-point numbers

A floating-point number is placed on the Forth data stack. For 32 bit targets, it 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 **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.

Creating variables

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

```
FVARIABLE VAR1
```

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

Accessing variables

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

Creating constants

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

```
1.234 FCONSTANT CON1
```

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

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.

Calculating sines, cosines and tangents

To calculate sine, cosine and tangent, use **FSIN**, **FCOS** and **FTAN** respectively. They take either an angle in degrees or radians, depending on which is set at the moment. See Setting degrees or radians.

Calculating arc sines, cosines and tangents.

To calculate arc sine, cosine and tangent, use **FASIN**, **FACOS** and **FATAN** respectively. They return an angle in degrees or radians, depending on which is set. See Setting degrees or radians.

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.

Calculating powers

Three power functions are supplied:

- e^x
- 10^x
- x^y

Calculating e^x

To calculate e^x, use **FE^X. FE^X** takes x as a floating-point number.

Calculating 10^x

To calculate 10^x, use **F10^X**. **F10^X** takes x as a floating-point number.

Calculating x^y

To calculate x^y, use **FX^Y**. **FX^Y** takes x and y as floating-point numbers.

Setting degrees or radians

The angular measurement used in the trigonometric functions can be set to be either degrees or radians. To set it to degrees, use the word **DEGREES**. To set it to radians use the word **RADIANS**.

Converting between degrees and 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.

Displaying floating-point numbers

Two words are available for displaying floating-point numbers, **F.** and **E.**. The word **F.** takes a floating-point number off 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.

Changes from v6.0

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.

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.

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.

Glossary

In the following glossary, you will find all the words that you are likely to need when using software floating-point; the words omitted are, in general, subroutines used by words in the glossary.

N.B. Abbreviation: f.p. = floating-point

D>F d -- f
"d-to-f"

Converts a double integer to a normalized f.p. number.

DEG>RAD f1 -- f2
"deg-to-rad"

Convert f1 degrees to its corresponding number of radians.

DEGREES --
"degrees"

Switches floating-point calculations to be done in degrees.

DNORM d n -- f
"d-norm"

Normalize double number d by n left shifts. Leaves a f.p. number on the stack.

E. f --
"e-dot"

Print the f.p. number on the stack in exponential form.

F, f --
"f-comma"

Compile the f.p. number on the top of the stack.

F. f --
"f-dot"

Print the top f.p. number on the stack in free format.

F! f addr --
"f-store"

Store the f.p. number f at address addr.

F+ f1 f2 -- f3
"f-plus"

Add together the top two f.p. numbers on the stack and put the f.p. result on the stack.

F- f1 f2 -- f3
"f-minus"

Subtract the top f.p. number on the stack from the second f.p. number on the stack, and put the f.p. result on the stack.

F* f1 f2 -- f3
"f-star"

Take the top two f.p. numbers off the stack, multiply them together, and leave the f.p. result on the stack.

F/ f1 f2 -- f3
"f-slash"

Divide the second f.p. number on the stack by the top f.p. number and leave the f.p. result on the stack.

F< f1 f2 -- flag
"f-less-than"

Leave true flag if f1<f2. Otherwise, leave a false flag.

F<0 f -- flag
"f-less-than-0"

Leave a true flag if f<0. Otherwise, leave a false flag.

F= f1 f2 -- flag
"f-equals"

Leave a true flag if the top two f.p. numbers on the stack are equal. Otherwise leave a false flag.

F0= f -- flag
"f-0-equals"

Leave a true flag if the f.p. number on the top of the stack is zero.

F> f1 f2 -- flag
"f-greater-than"

Leave a true flag if f1>f2. Otherwise, leave a false flag.

F>0 f -- flag
"f-greater-than-zero"

Leave a true flag if the f.p. number on the top of the stack is greater than zero.

F>D f -- d
"f-to-d"

Leave the integer part of f as a double number on the stack.

F>S f -- n
"f-to-s"

Takes the single number integer part of f and puts it on the stack.

F# -- f [executing]
"f-hash" -- [compiling]

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.

F#IN -- f 3 | 0
"f-hash-in"

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 3) is returned in the same way as an indicator is returned by **FNUMBER?**.

F@ addr -- f
"f-fetch"

Fetch the f.p. number from address addr and put it on the stack.

F10^X f1 -- f2
"f-10-to-the-x"

Raise 10 to the power f1 and put the result on the stack.

FABS f -- |f|
"f-abs"

Returns the modulus of the f.p. number on the top of the stack.

FACOS f1 -- f2
"f-a-cos"

Leave, on the stack, the angle (in degrees or radians) whose cosine is f1, such that 0<=f2<=180 (f2 in degrees).

FARRAY fn-1..f0 n -- [parent]
"f-array" n -- fn [child]

When generating the array, take n f.p. numbers and n, and compile them into the array. When executing the child word, take n and place f.p. number n from the array onto the stack. Note that the numbering in the array goes 0,1,..n-1.

FASIN f1 -- f2
"f-a-sine"

Leave, on the stack, the angle (in degrees or radians) whose sine is f1, such that -90<=f2<=90.

FATAN f1 -- f2
"f-a-tan"

Leave, on the stack, the angle (in degrees or radians) whose tangent is f1, such that -90<f2<90.

FCONSTANT f -- [parent]
"f-constant" -- f [child]

Floating-point equivalent of **CONSTANT**. Use in the form:

```
<f.p. number on stack> FCONSTANT <name>
```

FCOS f1 -- f2
"f-cos"

Take the cosine of f1 (degrees or radians) and put it on the stack.

FDROP f --
"f-drop"

Drop the f.p. number on the top of the stack.

FDUP f -- f f
"f-dup"

Duplicate the f.p. number on the top of the stack.

FE^X f1 -- f2
"f-e-to-the-x"

Raise e, the exponential number, to the power f1 and put the result on the stack.

FFRAC f1 f2 -- f3
"f-frac"

Leave the fractional remainder from the division f1/f2. The remainder takes the sign of the dividend.

FINT f1 -- f2
"fint"

Place the f.p. integer value of f1 on the stack.

FLITERAL f --
"f-literal"

When compiling, compile f as a literal. For example,

```
: ABCD    [ calculate f ] FLITERAL  ;
```

Compilation is suspended for the compile-time calculation of f. Execution of **ABCD** leaves f on the stack.

FLN f1 -- f2
"f-log-base-e"

Take the logarithm of f1 to base e and put the result on the stack.

FLOG f1 -- f2
"f-log-base-10"

Take the logarithm of f1 to base 10 (decimal) and put the result on the stack.

FMAX f1 f2 -- max{f1,f2}
"f-max"

Put the greater of the top two f.p. numbers onto the stack.

FMIN f1 f2 -- min{f1,f2}
"f-min"

Put the lesser of the top two f.p. numbers onto the stack.

FNEGATE f -- -f
"f-negate"

Negate the f.p. number on the top of the stack.

FNUMBER? addr -- 0|n 1|d 2|f 3
"f-number-query"

Converts string at address addr to either a single, double or floating-point number along with 1, 2, or 3 respectively. If a 0 is left on the stack then **FNUMBER?** was unable to convert the string.

FOVER f1 f2 -- f1 f2 f1
"f-over"

Floating-point equivalent of **OVER**.

FROT f1 f2 f3 -- f2 f3 f1
"f-rote"

Floating-point equivalent of **ROT**.

FSEPARATE f1 f2 -- f3 f4
"f-separate"

Leave the signed integer quotient f4 and remainder f3 when f1 is divided by f2. The remainder has the same sign as the dividend.

FSIGN f -- f flag
"f-sign"

Leave the f.p. number and a flag on the stack. Leaves a true flag if f is negative, else leaves a false flag.

FSIN f1 -- f2
"f-sine"

Leave the floating-point sine of f1 (degrees or radians) and put it on the stack.

FSQR f1 -- f2
"f-s-q-r"

Take the square root of the floating-point number on the top of the stack and put the result onto the stack.

FSWAP f1 f2 -- f2 f1
"f-swap"

Floating-point equivalent of **SWAP**.

FTAN f1 -- f2
"f-tan"

Take the tangent of f1 (degrees or radians) and put the result on the stack.

FVARIABLE --
"f-variable"

Floating-point equivalent of **VARIABLE**. Set up an fvariable by typing:

```
FVARIABLE <name>
```

FX^N f1 n -- f2
"f-x-to-the-n"

Raise f1 to the power n (n integer), and put result on the stack.

FX^Y f1 f2 -- f3
"f-x-to-the-y"

Raise f1 to the power f2 and put the result on the stack.

INTEGERS --
"integers"

Switches the action of **NUMBER?** to be **INTEGER?**. This action reverses that of **REALS**. Both **REALS** and **INTEGERS** are in the **FORTH** vocabulary.

RAD>DEG f1 -- f2
"rad-to-deg"

Convert f1 radians to degrees, and put result on the stack.

RADIANS --
"radians"

Switches floating-point calculations to be done in radians.

REALS --
"floats"

Switches the action of **NUMBER?** to be **FNUMBER?**. This action can be reversed by **INTEGERS**. Both **REALS** and **INTEGERS** are in the **FORTH** vocabulary.

S>F n -- f
"s-to-f"

Converts a single number to a normalized f.p. number

12 ROM PowerForth utilities

Supplied as source in the ROMFORTH directory are utilities to:

- compile source code on your target board from the cross-compiler IDE
- upload a binary image from your target to your PC
- download a binary image to your target from your PC

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 EPROM 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 ROMs 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.

Compiling text files

Source text files can be compiled from the host PC onto the target system. This saves time in not having to cross-compile the entire source if a small modification is made. The utilities permit text file to be split into pages for better layout when printed. An ASCII Form Feed character (decimal 12) separates one page from another.

The required files

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

Compiling a specified text file

To compile all or part of a specified text file onto your target, use **`GET`** or **`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

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 the cross-compiler IDE or a suitable communications package will be required.

XMODEM binary image download

Binary images can be downloaded to your PC using the XMODEM protocol.

Required files

To use this utility you must cross-compile the file XMODEM.FTH (also called BIN-DOWN.FTH in some targets).

Using the XMODEM binary download utility

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

```
addr #bytes BIN-DOWN
```

where addr is the start address and #bytes is the number of bytes to 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

Intel hex download

Binary images can be downloaded to your PC using the Intel hex format.

Required files

To use this utility you must cross-compile the file INTELHEX.FTH.

Using the Hex download utility

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.

ROM PowerForth

ROM PowerForth can be used to generate a stand-alone Forth system. With these utilities, you can generate an EPROM 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.

Hardware requirements

To develop an application using ROM PowerForth, your board requires an area which:

- is always EPROM
- is always RAM
- is RAM for development and EPROM for application

EPROM area

The area that is always EPROM contains the development kernel.

RAM area

The area that is always RAM is used for variables and all changeable data.

RAM/EPROM area

This area is used to develop your application. Therefore, it must be RAM while developing. Once your application is developed, the application's image must be saved into battery-backed RAM or EPROM. Therefore, this area must have the ability to be alterable but also non-volatile.

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:

- EPROM holding development kernel
- RAM which holds the variables and changeable data
- 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 EPROM as ROM PowerForth occupies less than 32k bytes of memory. Therefore, a header board can be made which converts one socket into two. For example, if the socket normally takes a 27512 EPROM, a board can be made which has a 32k 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 EPROM.

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/EPROM area. Alternatively, it can be copied into an EPROM if the board allows.

Configuring a turnkey application

The word **SETUP** takes the address of the word passed to it and marks this in the RAM/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
```

Discarding the application RAM area

The application can be discarded by typing:

```
0 ROM !
```

Changing the application RAM start address

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

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.

Glossary

BIN-DOWN addr len --
"bin-down"

Transmits a target image in XMODEM format to the host. AIDE can receive this file if the file server facilities are enabled.

CLS --
"c-l-s"

Clears the display by sending a trigger character (code 3) to the host.

GET <name>--
"get"

Compiles from a specified text file <name> on the host AIDE file server. File loading can be nested.

HEX-DOWN addr len --
"hex-down"

Transmits a target image in Intel Hex format to the host. The host can receive this file by enabling logging/capture.

INCLUDE <name>--
"include"

Compiles from a specified text file <name> on the host AIDE file server. . File loading can be nested.

13 Controlling compilation

While cross-compiling, the cross-compiler needs to be instructed on how to configure itself. You need to tell the cross-compiler:

- when to start compiling
- when to stop 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

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

Starting the cross-compiler

To start cross-compiling, use the word **`CROSS-COMPILE`**. Any code after this directive will be compiled into the target image instead of compiled onto the cross-compiler.

Stopping the cross-compiler

To mark the end of the cross-compilation phase, use **`FINIS`** or **`UMBILICAL-FORTH`**. **`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 with an Umbilical Forth target.

Defining memory - Sections and the xDATA directives

Regions of memory, known as sections, are defined in the control file by the **`SECTION`** directive. The cross-compiler treats all memory as fitting into three types of memory, code, initialised, and uninitialised, and maintains a current section for each type. 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 of the section, **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 in use of that type. When a section name is executed, it becomes the current section

CDATA is used to define areas of memory that contain code, usually ROM. **IDATA** is used to define areas of memory that can be 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 with headers 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** or **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 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.

Selecting section I/O

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.

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
```

IN-EMULATOR (offset --) causes the section memory to be in an EPROM emulator. The offset value is the offset from the start of the EPROM set 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. The EPROM emulator is accessed through a defined interface used the MPE/Leburg EPROM emulators. Any other EPROM emulators that provide this interface can also be used.

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

VIA-LINK (--) 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 the EPROM emulator. When the interactive session starts, the user types:

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

So that the RAM areas are accessed across the Umbilical link, so that target memory itself is used.

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

Section toOLS

ORIGIN \ -- addr

Returns start address of 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.

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.

Defining memory – Bank switched systems

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 0 can not be used as an identifier. 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 bank 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 --
```

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

Use of 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** or **WAIT** are 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. Similarly, doing this in the interrupt system can improve code reliability.

CDATA pages and vocabularies

The cross compiler treats **CDATA** pages as special cases of 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 compiler's search order before changing pages.

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. 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, which 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.

IDATA and UDATA pages

Page management

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.

Multitasking and interrupts

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. Similarly, doing this in the interrupt system can also improve code reliability.

Aligning generated code

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

Numbers and 16 bit targets

This 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 **EQU**ates defined in the control file.

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

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.

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 compiler off, use **NO-LOG**.

Conditional compilation

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

An example

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

Conditional compilation outside a colon definition

The example shown below compiles one of the **PRINT1OR2**'s. 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?                       \ Display one or two?
[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 within a colon definition

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 a constant **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? ]       [IF]              \ EQUate is interpreted
  [IF]
    ." 3"                 \ Display a three
  [ELSE]
    ." 4"                 \ Display a four
  [ENDIF]
;
```

[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 other version exists.

```
[UNDEFINED] <FOO> [IF]
: <FOO>
  …
;
[THEN]
```

[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]
```

Library files

When you need to keep code size to a minimum, the cross-compiler can resolve forward references by scanning library 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 condition compilation.

```
[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.

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"
```

Test code

The directives **TESTING [TEST** and **TEST]** support incorporating test code local to the definition the code tests. The DOCGEN/SC extension can be used for safety critical systems to produce FDA (the 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.

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 a C or assembler header files.

```
// - comment to end of line

/* comment N.B. white space delimited */

#define <name> text
```

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.

Direct port access under Windows NT/2000

If you are using Windows NT/2000 or any other version of Windows that treats direct port I/O as a privileged instruction, you must install 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.

14 The 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 the children of **FIELD** and local variables. These will be documented in the target specific section of the manual.

Inlining

Apart from these special cases 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.

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
```

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.

Implementations that use absolute calls will disable inlining of any word that makes an absolute call.

Use of **RECURSE** will disable inlining.

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 sothimg particularly carnal in nature.

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**.

COMPILER directives

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 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. 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.

15 Debugging tools

INTERACTIVE

When **INTERACTIVE** is used after **CROSS-COMPILE** and before **FINIS**, the compiler will not exit after compilation is finished, 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.

XDASM, DASM, DIS

Compilers for subroutine-threaded (STC) targets and compilers with the VFX code generator include a disassembler that can be used from Umbilical Forth, during cross compilation, or if you enter the compiler at the end of compilation

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

will disassemble the word <**name**>.

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** and **XLOC** instead.

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 the word **XUSES** instead.

XREF, XREF-ALL, XREF-UNUSED

The XREF cross reference 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.

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 the word **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.

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
```

LABELS

LABELS produces a list of the target labels.

EQUATES

EQUATES produces a list of the target equates.

ESCAPE

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** instead.

HELP

HELP lists the compiler directives, and gives some reminders.

INTERPRETERS

INTERPRETERS lists all the words which are special when interpreting.

COMPILERS

COMPILERS lists all the words which are special when compiling.

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 -- ; specify number of 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.

16 Compilation in more 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
- Defining words

Special compilation behaviour

The following words are treated as special cases during compilation, either because the code generator/optimiser produces in-line code rather than a call to a target word, or because the word is normally **IMMEDIATE** and is executed during compilation. As the list may have been extended since the manual was written, or because there are CPU specific switches, full lists of the interpretation and compilation words can be obtained using the directives **INTERPRETERS** and **COMPILERS**.

Code generator

!	*	+	+!
+STRING	-	-ROT	-TRAILING
0<	0<>	0=	0>
1+	1-	2!	2*
2+	2-	2/	2@
4*	4+	4-	4/
<	<=	<>	=
>	>=	?DUP	@
C!	C@	CHARS	COMPARE
COUNT	DROP	DUP	NIP
OVER	PICK	ROLL	ROT
SWAP	TUCK	U2/	U4/
U<	U>	W!	W@

Immediate

->	;	;CODE	ADDR
ASCII	EXIT	IF(	LITERAL
LOCALS|	POSTPONE	RECURSE	SEC-BASE
SEC-TOP	TO	TO-DO	{
[	[']	[CHAR]	[COMPILE]
[DEFINED]	[ELSE]	[IF]	[THEN]
[UNDEFINED]	[REQUIRED]	[ENDIF]	

Strings

""	."	ABORT"	C"
S"			

Comments

(	((	\

Control structures

+LOOP	?DO	?OF	AGAIN
BEGIN	CASE	DO	ELSE
END-CASE	ENDCASE	ENDIF	ENDOF
IF	LOOP	OF	REPEAT
THEN	UNTIL	WHILE	

Special case in defining words

,	>NUMBER	?LEAVE	ALLOT
ASSIGN	BLANK	C+!	C,
COMPILE	CREATE	DOES>	ERASE
HERE	I	J	LEAVE
MOVE	W,	WORD	

Special interpretation/compilation behaviour

The following words are treated as special cases during interpretation, because they mimic target behaviour, usually by dealing with target memory, or they are defining words,

because they are compiler directives, or because they are made available for execution during interpretation.

New directives may have been added since this manual was written, and a full list is available by using the words **HELP**, **COMPILERS** and **INTERPRETERS**.

Compiler directives

[ELSE]	[ENDIF]	[IF]
[THEN]	[UNDEFINED]	[REQUIRED]
2VARIABLE	32BIT	8BIT
CCITT	CDATA	CHECKSUM
COMPILER	CORG	CRC16
DEFINE-EMULATOR	E27010	E27020
E27040	E27080	E27128
E2716	E27256	E2732
E27512	E2764	E2764H
E2764L	EMU-IO	END-STRUCT
ESCAPE	EXTERNAL	FIELD
FIELD-TYPE	FINIS	FORTH
FROM-FILE	HOST	HOST&TARGET
HOST-COMPILATION	IDATA	IF(
IMMEDIATE	IN-EMULATOR	INCLUDE
INTEGERS	INTERNAL	INTERPRETER
IORG	LOCATE	LRCC16
MAKE-BUILD	MAKE-TURNKEY	NO-HEADS
ONLY	ORG	PTO
RESERVE	RESTART-COMPILATION	SAVE-COMPILATION
SDLC	SEC-BASE	SEC-END
SEC-TOP	SERIAL	SHOW-CODE
-SHOW-CODE	16BIT	2CONSTANT
SIMPLE16	SIMPLE32	SIMPLE8

STACK-CHECK	SUSPEND-COMPILATION	TARGET	
TARGET-ONLY	TARGET-SUPPORT	UDATA	
UORG	UPDATE-BUILD	USE-ANS-CONTROLS	
USE-MPE-CONTROLS	VIA-LINK	[DEFINED]	

Host referring words

@(H)	!(H)	C@(H)	C!(H)
W@(H)	W!(H)		

Defining words

:	:NONAME	BUFFER:	CODE
CONSTANT	CREATE	DEFER	EQU
MARKER	PROC	STRUCT	USER
VALUE	VARIABLE	VOCABULARY	I:

Assembler control

L$10:	L$1:	L$2:	L$3:
L$4:	L$5:	L$6:	L$7:
L$8:	L$9:	L:	LABEL
POSTFIX	PREFIX		

Target memory and interpretable

!	""	",	#
#> #S	'	(	((
)ELSE(	)ENDIF	)THEN	*
*/	*/MOD	+	+!
+STRING	,	,"	-
-1	-ROT	-TRAILING	-ZEROS
.	."	.R	.S

/	/MOD	/STRING	0
0<	0<>	0=	0>
1	1+	1-	2
2*	2+	2-	2/
2DROP	2DUP	2OVER	2ROT
2SWAP	4	<	<#
<=	<>	=	>
>=	>BODY	>IN	>NUMBER
?DUP	@	ABS	ACTION-OF
ALIGN	ALIGNED	ALL	ALLOT
ALSO	AND	ASCII	ASMCODE
ASSEMBLER	ASSIGN	BASE	BINARY
BLANK	BOUNDS	C!	C+!
C,	C@	CELL	CELL+
CELLS	CHAR	CHAR+	CHARS
COMPARE	COUNT	CR	D+
D<	DABS	DECIMAL	DEFINITIONS
DEPTH	DNEGATE	DROP	DUP
EMIT	ERASE	F#	F*
F+	F-	F/	FALSE
FILL	FIND	HERE	HEX
HOLD	INVERT	KEY	KEY?
LSHIFT	M*	M/MOD	MAX
MIN	MOD	MOVE	NEGATE
NIP	NOT	ONLY	OR
ORDER	OVER	PICK	POSTPONE
PREVIOUS	ROT	RSHIFT	RSHIFTS
S"	S>D	S>F	SCAN

```
 SIGN              SKIP              SM/REM            STATE

 SWAP              TO                TO-DO             TRUE

 TUCK              TYPE              U.                U<

 U<=               U>                U>=               UM*

 UM/MOD            UMOD              W!                W+!

 W,                W@                WIDEN             WITHIN

 WORD              XOR               [’](T)            \

 ]
```

Structures

A named structure is defined using the following template. When the name of a structure is executed its size is returned.

```
Size FIELD-TYPE <field-type-name>

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 a 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.

```
CELL FIELD-TYPE INT

STRUCT POINT                        \ -- size
  INT .X                            \ addr – addr’
  INT .Y                            \ addr – addr’
END-STRUCT

STRUCT RECT                         \ -- size
  POINT .TOP-LEFT                   \ addr – addr’
  POINT .BOTTOM-RIGHT                       \ addr – addr’
END-STRUCT

RECT BUFFER: NEW-RECT

CREATE ANOTHER-RECT                 \ -- addr
  RECT ALLOT
```

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**

The directives **CDATA IDATA** and **UDATA** select which type of memory the Forth words below affect:

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

CREATE

CREATE "<spaces>name" –
“create”

Skip leading space delimiters. Parse name delimited by a space. Create a definition for name with the execution 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.

```
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, 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 these at compile time. You cannot rely on the behaviour of the compielr under these circumstances.

Commas: , C, W,

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.

ALIGN and ALIGNED

The ANS specification provides these words to provide portability between systems that have different data alignment requirements. For example, a 386 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 --
"align"

If the data-space pointer is not aligned, reserve enough space to align it.

ALIGNED addr – addr'
"aligned"

a-addr is the first aligned address greater than or equal to addr.

ALLOT

ALLOT is used to reserve space in the current section. Note that when used in IDATA space, that the size of the initialised RAM table added by the compiler at the end of the ROM may be increased. See **RESERVE** and **BUFFER:**

ALLOT n –
"allot"

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.

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
“here”

addr is the data-space pointer.

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 –
“org”

Set the data space pointer of the current section.

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** . The address can be found using **ADDR**. By definitions, the data is in the current **IDATA** section.

VARIABLE defines a cell-sized variable that always returns its address. In Forth 6, 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 @ .
```

VARIABLE "<spaces>name" –
“variable”

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.

2VARIABLE "<spaces>name" –
“TWO-VARIABLE”

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" –
“value”

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.

BUFFER:

This is the equivalent with one important exception of the code below:

```
UDATA CREATE ABUFFER                \ -- uaddr
  <size> ALLOT

<size> BUFFER: ABUFFER     \ -- uaddr
```

The big 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.

RESERVE

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, ans the gap between the top of the **IDATA** section and the botton of the **UDATA** section can be made contiguous if the **IDATA** and **UDATA** sections are themselves contiguous.

RESERVE n – addr
"reserve"

The word **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>
```

UNUSED

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

UNUSED -- u/n

U/n is the amount of space remaining in the region addressed by HERE , in address units.

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 are dummies to allow a complete stack comment to be generated.

- The items between { and | are named inputs.
- The items between | and -- are local 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

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 following directives control where new words are placed.

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-compiler's 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-compiler's 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-compiler's 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 7: Compiler extension directives

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.

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.

Automatic handling

The cross-compiler will automatically build an analogue of the defining word in the host's conceptual ***INTERPRETER** vocabulary up to the terminating **; DOES>** or **;CODE**. This is triggered by the word **CREATE**. Consequently, any code between the **:** and the **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>
    @
;
```

Explicit handling

Explicit handling uses the compiler directives discussed in the previous section to control how defining words are created. This 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 may be useful when defining words are required and the absolute minumum of target memory is to be used.

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.

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:**.

Explicit handling

```
COMPILER

: -IF             \ -- ; only exists in host
  0= IF                    \ references *COMPILER's 0= and IF
;

TARGET
```

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
```

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 separate 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 v6.20 [build 0030]
```

BUILDFILE "<filename>" -- ; set build file name
"build-file"

Sets the name of the build file. By default it is BUILD.NO. e.g.

```
BUILDFILE MYBUILD.NO
```

MAKE-BUILD \ addr –
"make-build"

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.

BUILD$, --
"build-dollar"

Read the build file, and lay the text in the target as a counted string, e.g.

```
CREATE VERSION$  BUILD$,
```

only allocates the exact amount of space needed to hold the string.

UPDATE-BUILD --
"update-build"

Update the build number file. Place this just before **FINIS** so that a successful build updates the build number.

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%"
```

17 Forth on the target

This chapter describes how a Forth is laid out on a target board. It is therefore not necessary to read this chapter, but this chapter provides more information if you are interested or if you want to perform more advanced modifications to the cross-compiler or target.

Inside a ROM target Forth

A standalone ROM target Forth communicates with the host up a serial line. The host needs to be running a dumb terminal emulator. The terminal emulator displays any characters that arrive from the target and sends any characters typed at the host's keyboard. The target takes input and makes output directly from the serial line, not from a keyboard and to a display. To do this, the deferred words **EMIT** and **KEY** have the actions **SER-KEY** and **SER-EMIT** respectively.

The Forth memory map

A typical Forth system consists of several areas apart from the code space itself. The RAM on the target system is split into several areas:

- a user area for interrupts
- a user area and stacks for eack task
- a terminal input buffer (TIB) for standalone Forth

The remaining RAM is available for use by the Forth as dictionary space.

RAM initialisation

The ANS standard does not require variables (created by words **VARIABLE** or **CVARIABLE**) to be automatically 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 terminates. The compiler termination report tells you where it is located.

For Umbilical Forth targets, an EQUate **INIT-IDATA?** may be present to control whether the additional start up code to perform initialisation is compiled. This saves code space when this feature is not required

Two locations 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 can be found in the word **(INIT)** in COMMON\KERNEL.FTH.

In addition to using the memory store operators **C! W!** and **!**, RAM can be initialised when space is allotted using cross-compiler words that use **,** or **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!

Register usage

The Forth implementation on 32 bit targets uses subroutine threaded code with inlining for speed, with top-of-stack kept in a register. The assignment of the registers is given in more detail in the assembler chapter of the target specific manual.

Threading

For speed, MPE PowerForth uses Subroutine Threaded Code (STC) on 32 bit targets, as it is a good compromise between speed and space. The 16 bit targets may use Direct Threaded Code (DTC) or Subroutine Threaded Code, depending on whether the processor is regularly used on systems with limited memory space and the suitability for code generation. The routine which threads between the Forth words is called **NEXT,**. **NEXT,** is implemented as a macro, which is described in the assembler chapter. In most system **NEXT,** is simply an alias for the CPU return from subroutine instruction.

When suitable, STC compilers produce inline code for suitable primitives. The optimising VFX compilers all produce STC code when optimisation is not possible or is turned off, otherwise they produce native machine code.

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). Umbilical Forth is a system that contains a fully interactive Forth kernel in typically less than 4k bytes (32 bit targets), or 2k bytes (16 bit targets), although these figures will vary between different processors.

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 Standalone Forth. This does not apply on some processors such as the 8051, where stacks may be in different address spaces in the Standalone 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.

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.

When a word is cross-compiled, the cross-compiler places information in the symbol table. The symbol table therefore contains the CFA of the word in the target image. By using a message passing system between the cross-compiler and the target, the CFA 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.

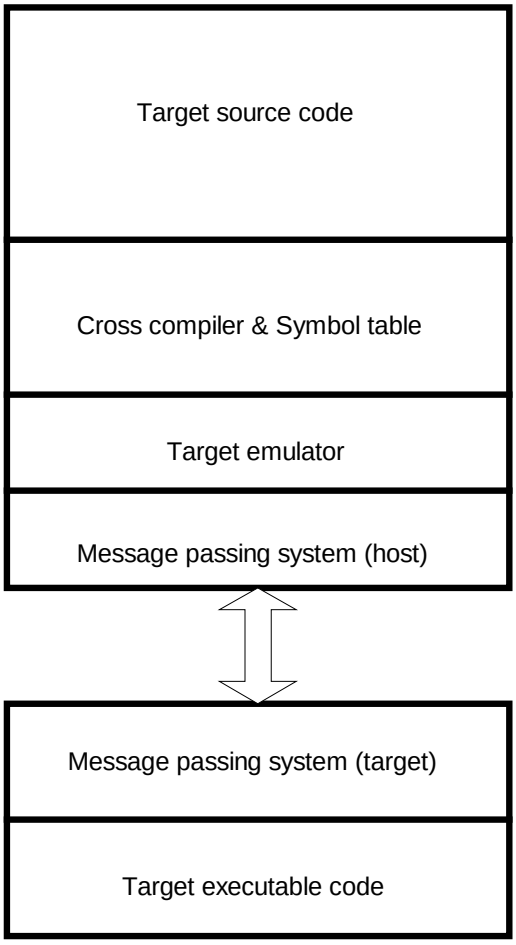

Figure 5: Umbilical Forth structure

18 Optimising development

While developing an application, you cycle through a series of steps:

- editing your source code
- cross-compiling to generate a binary image file
- downloading to an EPROM emulator/programmer
- testing and debugging your code

This development cycle is repeated until all development and debugging is completed. The faster you can go round this cycle, the sooner your application is finished.

Speeding up the compilation

Every time a cross-compilation is carried out, certain sections of code, which are never altered, are compiled again and again. This is particularly the case for the kernel files that generate the Forth image. You can use the partial compilation feature of the cross-compiler to halt the cross-compilation at a strategic position and save the cross-compiler's state. You can then continue cross-compiling from this saved position. In this way, you can dramatically reduce the time the application takes to compile.

Note: Partial compilation cannot be used when directly compiling to an emulator

Saving the compilation state

To stop and save the cross-compilation at a required place, use **SUSPEND-COMPILATION**. **SUSPEND-COMPILATION** is used in the form:

```
SUSPEND-COMPILATION <filename>
```

where <filename> is the name of files the cross-compiler will use to save the state information. The filename is a name **without** an extension.

Restarting from a saved state

To restart from a previously saved cross-compilation state, use **RESTART-COMPILATION**. **RESTART-COMPILATION** is used in the same form as **SUSPEND-COMPILATION**,

```
RESTART <filename>
```

where <filename> is the filename used when saving the compilation state. **RESTART-COMPILATION** must be used after the word **CROSS-COMPILE** and any macros must be loaded.

Note: The image file created by the compiler after a **SUSPEND-COMPILATION** must exist in the compilation directory.

An example

An example control file can be found in the directory ROM\PARTIAL.

Speeding up the download

The cross-compiler has the facility to download the image to the LeBurg emulator while it is compiling. This speeds up the turn-around of the edit, compile, download and test cycle by removing the download step. To download directly to a LeBurg emulator, you need to tell the cross-compiler:

- what size of EPROM it is generating
- the bus width (e.g. 8 bit, 16 bit)
- which page to put in the emulator

You also need to load the driver TSR for your emulator before running the cross-compiler, and to set the I/O port address it uses.

Note: This facility cannot be used with partial compilation.

Setting EPROM size and bus width

To set the size of EPROM to use and the bus width of the target board, use **DEFINE-EMULATOR**. This is in the form:

```
size width DEFINE-EMULATOR
```

where size and width are given in the following tables.

EPROM Size indicator	EPROM size
E2764	8k bytes
E27128	16k bytes
E27256	32k bytes
E27512	64k bytes
E27010	128k bytes
E27020	256k bytes
E27040	512k bytes
E27080	1M bytes

Table 8: EPROM size indicators

Bus width indicator	Bus width
8BIT	8 bit bus width
16BIT	16 bit bus width
32BIT	32 bit bus width

Table 9: Bus width indicators

For example, if your board uses a 27256 and your target has a 16-bit bus width, code:

```
E27256 16BIT DEFINE-EMULATOR
```

This instruction must be placed in your control file **before** the **CROSS-COMPILE** directive.

Setting the page

To send a page to an EPROM emulator, use **IN-EMULATOR** in the form:

```
xxxx IN-EMULATOR
```

where xxxx is the base address in the emulator where to place the image.

Using the emulator driver

To download to the emulator you first need to load an emulator driver. Which TSR you use depends on the emulator you have, and selection of the correct driver will be described in the EPROM emulator manual. One of these emulator drivers should be loaded before you run the cross-compiler. This should be done by loading the relevant TSR driver in your AUTOEXEC.BAT file.

Note that if you are using Windows NT/2000 or any other version of Windows that treats direct port I/O as a privileged instruction, you must install the NTPORT.EXE file from the COMPILER\XTRA directory as described in the installation section of the manual. You must also modify your control fileto include the **NT-ACCESS-PORTS** directive.

19 An example control file

The example control file presented here is typical for v6.2 cross compilers. 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.

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@mpeltd.demon.co.uk
     tech-support@mpeltd.demon.co.uk
web: www.mpeltd.demon.co.uk

From North America, our telephone and fax numbers are:
  011 44 23 8063 1441
  011 44 23 8033 9691

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
==============
))
```

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
```

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
```

Configure target

The target has to be configured as to 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
```

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
;
```

Application code

The application code example here is MPE's 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  ;
\ *G 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
\ *G 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
\ *G 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 \ --
\ *G 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
```

End of compilation

All the files have been compiled. All that is required is library file resolution and some sanity checks.

```
\ *********************
\ *S End of compilation
\ *********************

libraries       \ to resolve common forward references
  include %CpuDir%\libARM
  include %CommonDir%\library
end-libs


\ ***************
\ *S Sanity checks
\ ***************
```

```
decimal

  cr ." Required USER size is     : " next-user @ .
  cr ." Current USER allocation is: " up-size .
Next-user @ up-size > [if]
\ *G 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
```

20 Converting from v6.0 to v6.2

This chapter details the generic changes between v6.0 and v6.2 and shows you how to minimise the impact of the changes and how to take advantage of the new features. Note that all these changes are target source code changes, and you can choose to use your previous code base if you want to.

Generic I/O

The v6.0 versions of **KEY KEY? TYPE EMIT** and **CR** were **DEFER**red. The new v6.1 code is not deferred. Instead two new user variables, **IPVEC** and **OPVEC**, hold the address of a vector table which points to the action of these words.

This structure makes it much easier to add new I/O devices.

Multitasker

The v6.2 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 user instead of the task number by all task words.

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.

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 v6.2 kernel code relies on **+USER** and new application code should use **+USER** in preference to **USER**.

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.

TIMEBASE

Users with beta test versions of this code should note that it now prevents timerids being recycled except at ridiculously long intervals.

Build numbering

Now documented and available.

21 Moving from v5 to v6/VFX cross compilers

Introduction

The process of converting code from a version 5 MPE Forth Cross Compiler to the v6 and VFX compilers is straightforward. 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 the v6 targets have VFX code generators, but some additional work is also required.

Basic v5 to v6 conversion

Memory definitions

The v6 compiler uses the **SECTION** model for memory description. The description of memory is nearly always in 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 v6 control files in the CONFIGS directory for examples.

You should 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.

EPROM emulator

The EPROM emulator directives have changed, but will not affect you if you are not using an EPROM emulator with the v5 code.

The first major change is that defining an EPROM emulator does not force any section to use it. The second is that the **IN-EMULATOR** directive can be applied to any code section defined by **SECTION** or **PAGES**. It takes a 32-bit offset from the start of the EPROM set, and tells the compiler to place the output of that section starting at that offset from the beginning of the EPROM set. This allows a bank switched system to be defined with each page in the bank occupying a different portion of the EPROM.

Assembler changes

The use of the word **ASSEMBLER** to denote the start of a piece of assembly code is no longer supported in v6 compilers, 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 slightly from v5 to v6, especially in that **PAGE-WORD** is now called **PAGE-EXECUTE**, and the parameter passing may be slightly different. This is likely to mean 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 v6 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 v6 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 V6 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 v6 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 to v6 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.

Converting from DTC to STC and VFX compilers

The version 5 compilers produce what is termed direct threaded code (DTC), which is a particular implementation strategy for Forth. The 32-bit v6, some 16-bit v6, and the optimising VFX compilers produce subroutine threaded code (STC) with inlining. The VFX compilers also include the VFX optimising code generators that provide a substantial improvement in performance over simple STC code generation and very little

change in code density and sometimes an improvement that depends heavily on coding style.

The v6 targets are also 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 the manual.

Strategy

In order to convert an application from DTC to STC, it is probably easier to start from the v6 code base, as this will provide an easier long term upgrade path. The recommended stages are:

1) Generate a new v6 kernel for your target

2) Build a conversion harness that provides any missing words

3) Apply all the changes discussed for basic v5 to v6 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 code generator good enough that you can remove many code definitions in favour of high level Forth definitions, so enhancing maintainability and portability?

- Can coded interrupt routines now be rewritten in high level Forth for maintainability and portability?

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 MPE’s 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 v6 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.

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 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.

COMPILER, INTERPRETER, HOST, TARGET and ASSEMBLER

In both version 5 and the version 6 and VFX 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 compiler's 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
```

Umbilical Forth

The Umbilical Forth protocol has been extended and modified slightly. The TARGEND.FTH file used must be the one supplied with the v6 compiler if you want interactive testing. You will not be able to produce a byte for byte equivalent file from a v6 compiler that will run on your target with the v6 compiler, but you should be able to test it with the v5 compiler. Recompiling your code with the old TARGEND file on the v6 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 v6 TARGEND code also has facilities for using the multitasker with Umbilical Forth. This is controlled by the conditional compilation facilities.

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**.

22 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 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.

Choice of word names – ANS and Forth-83

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 v6 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.

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 it 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 NOT is **INVERT**.

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. **EXPECT** and **SPAN** are declared to be obsolete, and will be removed during the next revision process, which started at the time this manual was in preparation.

S” and C”

Traditionally, Forth has represented strings as a count byte followed by that many characters, in the same way as Pascal has. With the increasing use of zero terminated strings in some 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 has to mean 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.

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
```

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
```

Division

The Forth-83 standard introduced 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).

CREATE and friends

Section E.5 of the ANS specification suggests that, for embedded systems, **CREATE** be made sensitive to the current memory section. This makes it much easier to control where data is laid down, and removes the need for new 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.

>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.

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**.

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**.

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 **EXECUTE**ing 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.

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 **POSTPONE**d 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"
```

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**.

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 **VARIABLE**s 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 **THROW**s 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).

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.

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
```

23 IRTC and Stamp compiler differences

IRTC compilers

The IRTC Forth cross compilers are based on the same source code as the Forth 6 compilers except for the AVR compiler, but with the following restrictions:

Only an Umbilical Forth is provided. An interactive standalone Forth target is not provided.

There is no expansion beyond 64k of code, i.e. the **BANK** and **PAGES** directives are not provided.

Floating point target code is not provided.

Compiler source code is not provided.

Forth Stamp compilers

Versions supplied with the Forth Stamp boards are further limited:

- Code size is limited on the Forth Stamp versions to the size of the internal Flash on the Forth Stamp.
- No multitasker is provided on the Forth Stamp versions
- The **XREF** tools are removed on the Forth Stamp versions

Upgrades are available from the IRTC Forth Stamp version to the unlimited IRTC version and to the full Forth 6 compiler.

Late documentation on all cross compilers is in the DOCS documentation directory. The file RELEASE.XC6 describes late changes to the generic compiler, while RELEASE.xxx describes late changes on the CPU specific code.

24 Technical glossary

Compiler log When each label, variable, constant or colon defintition is cross-compiled the cross-compiler displays a dot or information about the compiled item.

Control file A file which is loaded by the cross-compiler. It contains directives to the cross-compiler and the names of any additional files to be compiled.

Cross-compiler A program which generates executable code for a processor different to that on which it is running.

Dictionary A list of words defined in a Forth system

Event A non-regular occurence. In the multitasker an event is used to trigger a task.

Glossary A list of forth words with their pronunciation, stack effect and a brief description of their action.

Host The platform the cross-compiler runs on. Normally a PC.

Image file The output of the cross-compiler. It has the extension .IMG by default.

Initialised RAM See RAM table.

Kernel The code required for interactive Forth.

Memory map A description of the start and end of ROM and RAM in memory

Multitasker A program which allows a processor to run more than one task by continuously switching between different tasks.

Paged target A system where there is more memory available that can be addressed at one time. Areas of memory can be switched into an addressable range, so simulating a larger address space than is physically possible.

RAM table An area of memory in the ROM that is copied to RAM at startup. It contains any initial values of variables.

ROM target forth A Forth which works on a ROM/RAM system as opposed to a RAM system.

ROM/RAM target A target with code executed out of ROM and data kept in RAM.

Scheduler The part of a multitasker which switches to the next task

Screen file A type of file which Forth source was originally developed in.

Serial line driver The words which interface the target code to the serial line. These are device dependant whereas the rest of the kernel is generic.

Symbol table Used and generated by the cross-compiler. It contains information on each item compiled.

Target The processor or board that the cross-compiler is generating code for.

Target mode One of XShell's modes that acts as a dumb terminal. It lets you communicate with your target board.

Task In a multitasking environment, a task is a stand-alone program that appears to run simultaneously with other tasks.

Task control block Where information about a task is kept. It is used by the scheduler to switch to the next task.

TCB See task control block

UART Universal Asynchronous Receiver/Transmitter - Sends and receives serial data.

Umbilical Forth A reduced Forth designed for single chip targets. Uses a message passing system to commicate with the host.

Unresolved references Any words which are used in the source code but are not defined.

Vocabulary An independently linked subset of the dictionary

25 Error messages

Error messages are kept in the file X*.ERR in the COMPILER directory, where the '*' denotes the processor type. Error numbers start at zero and each error number refers to a line in the file, starting at line zero.

The error messages are listed in different categories:

- general Forth errors
- system messages
- assembler errors (these are listed in the accompanying processor specific manual)
- binary module errors
- source file errors
- operating system errors
- text file errors

General Forth errors 0..15

These are the basic errors of a Forth system.

Error 0 - is undefined. The word is not in the dictionary search order specified, or it was misspelled.

Error 1 - empty stack, the last operation caused a stack underflow. Usually caused by using the wrong number of parameters to a word.

Error 2 - dictionary full, there is no room for more definitions. This error should not arise within the cross-compiler unless you are extending it.

Error 3 - has incorrect address mode.

Error 4 - is redefined - the word's name has been used before. This is only a warning, not a proper error.

Error 5 - is undefined. See error 0.

Error 7 - full stack, there are too many items on the stack. Usually caused by a stack fault in a loop.

Error 8 - cannot open **USING** file. Incorrect file name? Wrong directory?

Error 9 - cannot compile from screen zero.

Error 12 - uninitialised deferred word.

Error 13 - **BASE** must be **DECIMAL**.

Error 14 - missing decimal point. Only found when using floating-point extensions.

System messages 16..31

These are error messages caused by mistreating Forth.

Error 17 - compilation only, use in definition, not when executing. Usually happens when a **;** is missing from a previous word.

Error 18 - execution only - not allowed during compilation. Usually because a **[COMPILE]** is missing in front of an immediate word.

Error 19 - conditionals not paired - overlapping control structures.

Error 20 - definition not finished - a control structure needs correction.

Error 21 - in protected dictionary - the word is below the address in **FENCE**. Not found in the cross-compiler except when modifying the cross-compiler, or in bizarre circumstances with Umbilical Forth.

Error 22 - use only when loading, illegal from the keyboard

Error 23 - block number out of range 0..32767 (0..7FFFh).

Error 24 - reset vocabularies - **CONTEXT** must be the same as **CURRENT** when using **FORGET**.

Error 25 - do not use when loading, only from the keyboard.

Error 26 - initialised RAM size exceeded. Often happens when arrays are defined before variables. To reduce the size of this table, all initialised or preset RAM should be defined before arrays are used.

Error 27 - forward references are illegal between **CREATE** ... **DOES>** and **I:** ... **;** for the cross-compiler.

Error 28 - word between **CREATE** ... **DOES>** or **I:** ... **;** is not in host **FORTH** vocabulary.

Error 29 - illegal internal value - contact MPE.

Assembler errors 32..47, 144...159

These are listed in the accompanying processor specific manual.

Binary module errors 48..63

Error 49 - public words table full - max 32 (decimal) words/module.

Error 50 - module number out of range 0..31 (decimal).

Error 51 - slot already occupied - slot must be empty before entry is made.

Error 52 - not enough memory.

Error 53 - can't load module file - DOS can't find it, or can't read it.

Error 54 - can't free memory - DOS won't let go - see DOS function 49H.

Error 55 - module not present - requested module is not resident.

Error 56 - external references table full - max 32 (decimal) words/module.

Error 57 - unresolved external reference - use RESOLVE-ALL before execution.

Error 62 - illegal operation in slave module.

Error 63 - illegal operation in master module.

Source file errors 64..79

These errors are given by the screen file handlers.

Error 65 - no screen file open. Often a result of a previous operation failing to open or reopen a file.

Error 66 - screen file seek error.

Error 67 - screen file write error.

Error 68 - path not found. Usually because the file or path name has been misspelled.

Error 69 - starting screen number less than ending screen number.

Operating system errors 80..112

Error 81 - invalid function number - OS doesn't know what to do.

Error 82 - file not found - wrong directory or doesn't exist.

Error 83 - path not found - incorrect spelling? - device not installed?

Error 84 - no handle available - all handles are in use.

Error 85 - access denied - e.g. attempt to write to read-only file.

Error 86 - invalid handle - file/path not open?

Error 87 - memory control blocks destroyed.

Error 88 - insufficient memory.

Error 89 - invalid memory block address - OS did not allocate this segment.

Error 90 - invalid environment - previous SET or PATH command bad.

Error 91 - invalid format - ask Microsoft what this one means.

Error 92 - invalid access code.

Error 93 - invalid data.

Error 95 - invalid drive specification.

Error 96 - attempt to remove current directory.

Error 97 - not same device.

Error 98 - no more files to be found.

Text file errors 112..127

These errors are issued by the text file handler.

Error 113 - cannot allocate memory. Each nested file needs about 9k bytes.

Error 114 - cannot free memory.

Error 115 - cannot open file.

Error 116 - cannot close file.

Error 117 - cannot seek to byte requested in file.

Error 118 - read-path error. Disk cannot be read, normally seen only from floppy disks, or failing hard discs.

Error 119 - file nesting depth reached - cannot open another file. You have nested files too deep.

Error 120 - file de-nesting error.

Error 121 - start page number greater than last page number in file.

Overlay load errors 128..143

Error 129 - cannot close overlay file.

Error 130 - overlay file read error.

Error 131 - overlay file write error.

Error 132 - overlay file open error - does file exist?

Error 133 - overlay produced on different version of ProForth.

Error 134 - overlay too big.

Error 135 - overlay must be compiled twice!

Error 136 - overlay length is mod. 256, insert `1 ALLOT' & recompile.

Error 137 - overlay must be longer than 256 bytes.

Error 138 - overlay copies must be the same length.

Error 139 - cannot create overlay file.

26 Further information

MPE courses

MicroProcessor Engineering runs the following courses:

Architectual introduction to Forth

A two-day course for those with little or no experience of Forth. It provides an introduction to the architecture of a Forth system. It shows, by practical example, how software can be coded, tested and debugged, quickly and efficiently, using Forth's interactive abilities.

Embedded software for hardware engineers

A three-day course for hardware engineers needing to construct real-time embedded applications using Forth cross-compilers. Includes multitasking and writing interrupt handlers.

Quick Start Course

A very hands-on tailored course on your site using your own hardware, and including installation of a target Forth on your hardware, approaches to writing device drivers, and and whatever else you need. The course is usually three days long.

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 has considerable knowledge of embedded hardware design. 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 and newspaper presses. We have done projects ranging from a few days to major international projects covering several years and continents and many countries.

Projects at MPE cover topics such as custom compiler developments, including language extensions such as SNMP, and new CPU implementations, custom hardware design and compiler installations, a portable binary system for smart card payment systems, machine controllers, virtual memory systems, and code porting to new hardware or operating systems. We can operate to fised price and fixed term contracts.

We 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)

Recommended reading

For an introduction to Forth:

“Starting Forth” by Leo Brodie

“Thinking Forth” by Leo Brodie

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.

27 Index