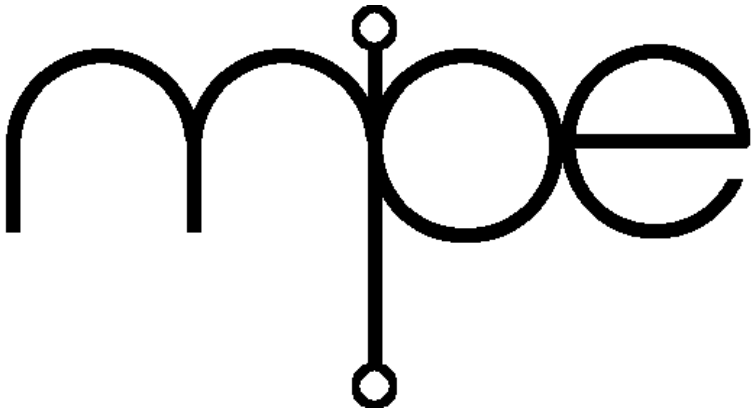
Forth 7 ARM Cortex-M Cross Compiler

with VFX code generation

Microprocessor Engineering Limited



Published by Microprocessor Engineering

MPE VFX Forth ARM Cortex-M Cross Compiler
User manual
Manual revision 7.6
15 July 2023

Software
Software version 7.6

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
e-mail: mpe@mpeforth.com
tech-support@mpeforth.com
web: www.mpeforth.com

Table of Contents

1 Licensing and other matters 1
- 1.1 Commercial use 1
- 1.2 Community licence 1
  - 1.2.1 Distribution of application programs 1
  - 1.2.2 Distribution of files 2
  - 1.2.3 Warranties, support, and copyright 3
- 1.3 Enterprise licence 3
  - 1.3.1 Distribution of application programs 3
  - 1.3.2 Distribution of files 4
  - 1.3.3 Warranties, support, and copyright 4

2 Introduction 5
- 2.1 What you get 5
- 2.2 Running the compiler 5
  - 2.2.1 Package and binary naming 5
  - 2.2.2 Launching the compiler 6
- 2.3 Downloading the target image 6
- 2.4 Supported Cortex CPUs and hardware 7
- 2.5 Supported ARM CPUs and hardware 7
- 2.6 New targets 8

3 How Forth is documented 9
- 3.1 Forth words 9
- 3.2 Stack notation 10
- 3.3 Input text 11
- 3.4 Other markers 12

4 ARM/Cortex VFX code generator 13
- 4.1 Directives 13
- 4.2 ARM and Cortex intrinsics 16
- 4.3 Cortex-specific intrinsics 17
- 4.4 Optimiser in-built macros 18
- 4.5 Efficient coding practice 18

5 ARM Cortex Cross assembler 21
- 5.1 Why write in assembler? 21
- 5.2 Creating Forth words in assembler 21
  - 5.2.1 Defining assembler words 21
  - 5.2.2 Writing assembler words 22
  - 5.2.3 Register names 22
  - 5.2.4 Preserving the Forth registers 22
  - 5.2.5 Executing an assembler word 24
- 5.3 Assembling into memory 24
- 5.4 Creating defining words in assembler 25
- 5.5 Structured programming 25
  - 5.5.1 Control structures 25

5.5.2 Labels ... 27
5.5.3 Local labels ... 28
5.6 Creating macros ... 28
5.6.1 Defining a macro ... 28
5.6.2 Using a macro ... 29
5.7 Debugging ... 29
5.8 CPU selection ... 29
5.9 Number bases ... 29
5.10 Thumb-2 instruction set ... 30
5.10.1 Base instruction set ... 30
5.10.2 Integer DSP ... 41
5.10.3 Floating point for Cortex-Mx and ARM32 ... 44
5.11 ARM instruction set ... 47
5.12 Register naming ... 49
5.13 Immediate constants ... 50
5.14 Shift operations ... 51
5.15 Addressing modes ... 51
5.15.1 Pre-indexed addressing ... 51
5.15.2 Post-indexed addressing ... 52
5.15.3 PC relative addressing ... 53
5.15.4 Byte and half word addressing ... 53
5.16 Register lists ... 53
5.17 Switching between Forth and Assembler ... 54
5.18 Glossary ... 55

6 Cortex interrupts ... 61
6.1 Interrupts on the Cortex-M3+ ... 61
6.2 Interrupts on the Cortex-M0/M1 ... 61
6.3 Writing Forth interrupt handlers ... 61
6.3.1 Setting an interrupt ... 62
6.4 Some common problems ... 62
6.4.1 Stack faults ... 62
6.4.2 Source is not cleared ... 62
6.5 Writing assembler interrupt handlers ... 62
6.6 Controlling the interrupts ... 63
6.6.1 Enabling interrupts ... 63
6.6.2 Disabling interrupts ... 63
6.7 A simple example ... 63
6.7.1 The timer ISR ... 63
6.7.2 Initialising the timer ... 63
6.7.3 Setting the vector ... 64
6.7.4 Testing the timer is running ... 64
6.8 Interrupt handlers in detail ... 64
6.8.1 Interrupt protection ... 64
6.8.2 End of interrupt requirements ... 65
6.9 Glossary ... 65

7 ARM interrupts ... 67
7.1 Interrupts on the ARM ... 67
7.2 Writing Forth interrupt handlers ... 67
7.2.1 Setting an interrupt ... 67
7.3 Some common problems ... 67
7.3.1 Stack faults ... 68

7.3.2 Source is not cleared ... 68
7.4 Writing assembler interrupt handlers ... 68
7.5 Controlling the interrupts ... 68
7.5.1 Enabling interrupts ... 68
7.5.2 Disabling interrupts ... 68
7.6 A simple example ... 68
7.6.1 The timer ISR ... 68
7.6.2 Initialising the timer ... 69
7.6.3 Patching your ISR onto the timer ... 69
7.6.4 Testing the timer is running ... 69
7.7 Interrupt handlers in detail ... 69
7.7.1 Interrupt structure ... 70
7.7.2 Setting an interrupt ... 70
7.7.3 Interrupt protection ... 71
7.7.4 End of interrupt requirements ... 71

8 Calling functions in other languages ... 73
8.1 Introduction ... 73
8.1.1 SVC calls ... 73
8.1.2 Jump tables ... 74
8.1.3 Double indirect call tables ... 75
8.1.4 Pointer to double indirect call tables ... 76
8.1.5 Direct calls ... 76
8.2 C comments in declarations ... 76
8.3 Format ... 77
8.4 Calling Conventions ... 77
8.5 Promotion and Demotion ... 77
8.6 Argument Reversal ... 78
8.7 Controlling external references ... 78
8.8 Pre-Defined parameter types ... 80
8.8.1 Calling conventions ... 80
8.8.2 Basic Types ... 80

9 JLink Flash and debug interface ... 83
9.1 Installing Segger software ... 83
9.1.1 Windows ... 83
9.1.2 Mac OS X ... 83
9.1.3 Linux ... 83
9.2 Controlling the J-Link ... 83
9.2.1 Umbilical Forth ... 84
9.3 Initialisation and download ... 84
9.4 Debug and stepping ... 85
9.5 Breakpoints ... 86
9.6 Memory driver ... 86
9.7 J-Link pseudo serial port ... 87
9.8 Interactive debugging ... 87

10 SVD file converter ... 89
10.1 Time and date ... 89
10.2 Data buffers and variables ... 90
10.3 Tag handling ... 92
10.4 File version ... 94

11 Further information 95
11.1 MPE courses 95
11.2 MPE consultancy 95
11.3 Recommended reading 96

Index 97

List of Tables 101

1 Licensing and other matters

The license terms here apply to all versions of VFX Forth 5 and beyond and to the MPE Cross Compilers. Separate sections of this chapter cover both the Community (non-commercial use) and Enterprise (commercial use) licenses.

1.1 Commercial use

Commercial use means that money changes hands, either by the sale of a product or by payment for a job or employment. If commercial use applies to you, your organisation or employer, you need an Enterprise licence.

If you sell an application written with VFX Forth, that is commercial use.

If you sell a service that uses or was developed with VFX Forth, that is commercial use.

If you are paid to write software with VFX Forth, that is commercial use.

If you sell hardware or software but give away software written with VFX Forth to enhance it, that is still commercial use.

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

If you teach a class using VFX Forth in a class, that is a special case, and a Community non-commercial licence is all that is required, both for the teachers and the students, but for the duration of the class only.

1.2 Community licence

The terms in this section apply to compilers supplied with the Community licence.

All applications written with the Community licence must acknowlege this at sign on and in the documentation.

Commercial use with the Community licence is not permitted.

You may not use VFX Forth or MPE cross compilers to produce products that compete with one or more MPE Forth products.

1.2.1 Distribution of application programs

There are several ways in which VFX Forth applications can be distributed. These are:

- Sealed turnkey application with no access to the interactive Forth.

- Sealed except for engineering and maintenance access by the developer.
- Open Forth interpreter/compiler provided for the end user.

Sealed turnkey applications

Providing that the user can have no access to the underlying Forth and its text interpreter, turnkey applications written in VFX Forth may be distributed without licence. An acknowledgement of the VFX Forth Community licence is required at start up of the application.

Engineering and maintenance access

If the developing organisation wishes to provide what the user sees as a sealed turnkey application, but in which an open Forth can be exposed for engineering and maintenance access by the developer organisation no licence will be charged for. However a license agreement must be signed with MPE in order to protect MPE's copyright. An acknowledgement of the VFX Forth Community licence is required at start up of the application.

If the company or person responsible for maintenance is not the developer then the maintenance company or person must have a licence.

Our objective here is to protect our copyright and to ensure that no undocumented Forth systems are shipped.

User open Forth interpreter

In order to distribute a system with an open Forth interpreter for the end user, a licence agreement must be signed with MPE.

Our objective here is to protect our copyright and to ensure that no undocumented Forth systems are shipped.

1.2.2 Distribution of files

Unless special license terms say otherwise, this section applies.

Shipped applications may be based on the files *VfxForth_x64_lin.elf* or *VfxForthB_x64_lin.elf*.

Object code generated from the source files can of course be included in your applications. MPE source files and all other files including editors, support programs and shared libraries are part of the development environment, which may not be distributed without prior permission in writing from MicroProcessor Engineering. However, the INI parser libraries, *mpeparser.dll* or *libmpeparser.** may be distributed with your applications - these files are distributed under an MIT license.

The source directories provided with VFX Forth and the MPE Forth cross compilers may not be distributed, and remain the intellectual property of MicroProcessor Engineering Ltd. Some source directories, e.g. the INI parser, contain additional licenses which apply to those directories only.

1.2.3 Warranties, support, and copyright

We try to make VFX Forth as reliable and bug free as we possibly can. We support our products. If you find a bug in VFX Forth or its associated programs we will do our best to fix it. Please send us sample code and a listing of the problem. We will then let you know of an update when we have fixed the problem. Do however, check with us first in case the problem has already been fixed. Technical support is only provided for the current shipping version of VFX Forth.

Make as many copies as you need for backup and security.

1.3 Enterprise licence

The terms in this section apply to compilers supplied with commercial use permitted.

If you have a subscription, commercial use is only permitted while the subscription is valid, i.e. paid for.

You may not use VFX Forth or MPE cross compilers to produce products that compete with one or more MPE Forth products.

Unless otherwise stated, all files are copyright MicroProcessor Engineering Limited.

1.3.1 Distribution of application programs

There are several ways in which VFX Forth applications can be distributed. These are:

- Sealed turnkey application with no access to the interactive Forth.
- Sealed except for engineering and maintenance access by the developer.
- Open Forth interpreter/compiler provided for the end user.

Sealed turnkey applications

Providing that the user can have no access to the underlying Forth and its text interpreter, turnkey applications written in VFX Forth may be distributed without royalty. An acknowledgement will be gratefully appreciated.

Engineering and maintenance access

If the developing organisation wishes to provide what the user sees as a sealed turnkey application, but in which an open Forth can be exposed for engineering and maintenance access by the developer organisation no royalty will be charged. However a license agreement must be signed with MPE in order to protect MPE's copyright. If the company responsible for maintenance is not the developer then the maintenance company must have a license.

User open Forth interpreter

In order to distribute a system with an open Forth interpreter for the end user, a license agreement and royalty terms must be agreed with MPE. MPE is able to help you supply selected portions of the development environment, or to provide end user documentation. The cost of such licenses will depend on the facilities required.

1.3.2 Distribution of files

Unless special license terms say otherwise, this section applies.

Shipped applications may be based on the files *VfxForth_x64_lin.elf* or *VfxForthB_x64_lin.elf*.

MPE source files and all other files including editors, support programs and shared libraries are part of the development environment, which may not be distributed without prior permission in writing from MicroProcessor Engineering. However, the INI parser libraries, *mpeparser.dll* or *libmpeparser.** may be distributed with your applications - these files are distrubuted under an MIT license.

The source directories provided with VFX Forth may not be distributed, and remain the intellectual property of MicroProcessor Engineering Ltd. Some source directories, e.g. the INI parser, contain additional licenses which apply to those directories only.

1.3.3 Warranties, support, and copyright

We try to make VFX Forth as reliable and bug free as we possibly can. We support our products. If you find a bug in VFX Forth or its associated programs we will do our best to fix it. Please send us sample code and a listing of the problem, and let us know the serial number of the prodct. We will then send you an update when we have fixed the problem. Do however, contact us or your supplier first in case the problem has already been fixed. Please note that the level of Technical Support that we can offer will depend on the Support Policy purchased with VFX Forth. Technical support is only provided for the current shipping version of VFX Forth.

Make as many copies as you need for backup and security. The distribution is not copy protected. VFX Forth is copyrighted material and only **one** copy of it should be in use at any one time. Contact MPE or your vendor for details of multiple copy terms and site licensing.

As we sell copies of VFX Forth through dealers and purchasing departments we cannot keep track of all our users. If you have not already been in contact with us, please send your details to

`mailto:techsupport@mpeforth.com`

2 Introduction

2.1 What you get

The MPE Forth 7 cross compiler for the ARM Cortex-M family consists of five portions which will have been installed for you.

1. AIDE front end (Windows only).
2. Forth 7 cross compiler for ARM and Cortex-M0/M1/M3/M4/M7.
3. Standalone Forth source code and configuration files. Note that ARM Forth Stamp compilers do not include the PowerFile target file system, PowerView GUI, PID loop controller and state machine compiler. The Stamp compiler is restricted to 120k of code space and 64k of RAM, and configurations and device drivers are only supplied for the following CPU families: NXP LPC21xx; NXP LPC17xx; Freescale Kinetis; ST STM32F4xx.
4. Umbilical Forth source code and configuration files. The best reference for the target code is to read it yourself.
5. Support tools - MAKE utility, Intel Hex and Motorola S-record converters plus a range of additional tools. See the *TOOLS* folder.

The *Docs* folder contains the cross compiler and CPU specific manuals. This folder also contains CPU and chip documentation, so browse it to see what is there. The *Cortex/Manual* folder contains a manual for the Cortex-specific target code. The *Common/Manual* folder contains a manual for common high-level target code used by all CPU targets.

Many folders contain release notes in files called *Release.xxx.txt*. These contain a change history.

2.2 Running the compiler

The mechanics of compilation are covered in the general cross-compiler manual, *xc7man.pdf*. This section covers how to launch the cross itself.

2.2.1 Package and binary naming

The compilers are named according to the pattern below. They can be found in the <target>/Compiler directory. Under macOS and Linux, there will be copies in the /usr/local/bin directory.

```
x<target>_<cpu>_<os>[.<bin>]

<target> = CPU for which the cross-compiler produces code, e.g.
           ArmCortex
           MSP430
           x86
           x64

<cpu>    = x86     32 bit Intel - host operating system CPU
           x64     64 bit AMD/Intel
           arm     32 bit ARM
           aa64    64 bit ARM

<os>     = win     Windows 32/64 - host operating system
           lin     Linux 32/64
           mac     OS/X and macOS 32/64

<bin>    = exe     Windows 32/64 - host operating system file type
           elf     ELF 32/64, e.g. Linux
           mo      Mach O for Mac OS/X
```

Under macOS and Linux, the installation script will produce a link of the form

```
x<target>
```

The links are the recommended way to run the compiler. Under Linux and OS/X, the links are symlinks in the /usr/local/bin directory. Under Windows (must be 7+) these are symlinks in Windows\System32, e.g.

```
mklink xArmCortex.exe x:\buildkit.dev\software\COMPILER\xArmCortexHfp_x86_win.exe
```

2.2.2 Launching the compiler

At launch time, the cross-compiler is a normal Forth system. When it executes the word `CROSS-COMPILE`, it "pulls down the shutters" and reconfigures itself as a cross-compiler. When you run the cross-compiler you can pass Forth commands on the command line, e.g.

```
xArmCortex  include MyControlFile.ctl
```

2.3 Downloading the target image

If the board you are using does not have a pre-installed Forth kernel, you will have to program the image produced by the cross compiler using a suitable bootloader or JTAG unit.

Most ARM and Cortex targets supported by MPE include a utility word `REFLASH ( -- )`, which will download a new image into the target and reboot the board. The `REFLASH` tools are documented in the target code manuals.

The Segger JLink tools are supported for Flash programming and debugging. See the separate chapter for more details.

2.4 Supported Cortex CPUs and hardware

The Cortex family of CPUs is supplied by a large number of silicon vendors. The target code will run on any of the CPUs, and there are UART and timer drivers for several versions. Look in the *Cortex/Drivers* and *Cortex/Hardware* folders to see if your Cortex hardware is already supported.

The standard hardware boards supported by this compiler are:

- Infineon XMC11xx on XMC2Go board,
- STM32072B Discovery board for STM32F0, Also STM32F031, STM32F042, STM32051,
- STM32F103 on Blue Pill and Reva boards,
- STM32F3 on Nucleo board,
- STM32F4 Discovery board,
- STM3240G-EVAL board for STM32F4xx,
- STM32F429-DISC0 or DISC1,
- Olimex STM32-E407,
- Olimex STM32-P107,
- Olimex LPC1766-STK,
- Reva STM32F103R for STM32 devices,
- Keil MCB1114 for LPC1114,
- Nuvoton NuMicro-SDK for NUC1100/120/130/140,
- NXP/FReescale K60 Tower System,
- NXP/Freescale FRDM KL25Z,
- NXP 4078 on Redboard 4078,
- TI LM3S9B92,
- TI TM4C1294 on Launchpad board.

Support for the following devices is planned or in preparation.

- Energy Micro
- Atmel
- ST STM32F1 low power devices

2.5 Supported ARM CPUs and hardware.

The ARM family of CPUs is supplied by a large number of silicon vendors. The target code will run on any of the CPUs, and there are UART and timer drivers for several versions. Look in the *ARM/Drivers* and *ARM/Hardware* folders to see if your ARM7/9 hardware is already supported.

The standard hardware boards supported by this compiler are:

- MPE USB ARM Stamp (NXP LPC2106)
- Olimex LPC-P2148 (NXP LPC2148)
- Keil MCB2300 with LPC2388
- Hitex STR912 EvalBoard (ST STR912FAW44)
- Atmel EB55 (Atmel AT91M55800A)

2.6 New targets

MPE has ported the Forth system to a much wider range of ARM CPUs than the ones mentioned above, so look in the *ARM/Hardware* and *Cortex/Hardware* folders to see if your hardware is already supported.

MPE can provide assistance with board installations and drivers for new devices.

3 How Forth is documented

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

When you browse the words in the Forth dictinary using `WORDS` or when reading source code you may come across some words which are not documented. These words are undocumented because they are words which are only used in passing as part of other words (factors), or because these words may change or may not exist in later versions.
"*Documentation is like sex: when it is good, it is very, very good; and when it is bad, it is better than nothing.*" - *Dick Brandon*

3.1 Forth words

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

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

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

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

- The definition line.
- The description.

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

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

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

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

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

This is a "quick reference" comment.

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

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

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

3.2 Stack notation

```
before -- after
```

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

An action on the return stack whill be shown

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

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

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

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

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

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

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

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

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

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

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

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

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

3.3 Input text

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

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

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

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

3.4 Other markers

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

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

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

`M` Affected by multi-tasking

`U` A user variable.

4 ARM/Cortex VFX code generator

In the main the VFX code generator is a black box. There are however a few control switches which are useful in limited cases. Note that because of the efficiency of the VFX code generator, it is very rarely necessary to use assembler code. The main exceptions are when dealing with very CPU specific code such as the cache control routines.

As of July 2018, local variables for Cortex-M0 are fully operational.

4.1 Directives

```
: +short-branches       cc/i    \ --
```
You save memory by compiling branches with an 8 bit offset. This is the default condition.

```
: -short-branches       cc/i    \ --
```
You may need this for big conditional structures.

```
: [-short-branches      cc/i    \ -- x
```
You may need this for big conditional structures.

```
  [-short-branches ... short-branches]
```

```
: [+short-branches      cc/i    \ -- x
```
You can save memory by compiling branches with an 8 bit offset.

```
  [+short-branches ... short-branches]
```

```
: [short-branches       cc/i    \ -- x
```
Save the current short/long branch state.

```
  [short-branches ... short-branches]
```

```
: short-branches]       cc/i    \ x --
```
Restore the saved short/long branch state.

```
: [short-branches?      cc/i    \ -- x
```
Return true if short/long branches are enabled.

```
  [short-branches ... short-branches]
```

```
: +use-litpool  \ -- ; enable use of literal pool
```
Enables use of the literal pool, which results in shorter code at the expense of more data cache activity. This is usually a better option than generating four-instruction sequences. `+USE-LITPOOL` is the default for the ARM instruction set because of the significant reduction in code size. `+USE-LITPOOL` is not permitted for Cortex-M3 onwards, but is required for Cortex-M0/M1.

```
: -use-litpool  \ -- ; disable use of literal pool
```
Disables use of the literal pool. Immediate values that cannot be represented by an ARM 12 bit literal may take up to four instructions. `+USE-LITPOOL` is the default for the ARM, Cortex-M0 and Cortex-M1 instruction sets.

```
: T2-mode       cc/i    \ -- ; select 32 bit mode
```
Selects generation of code using the full Thumb-2 instruction set and a register allocation model optimised for Thumb-2. This is the default.

```
: Cortex-M0     cc/i    \ --
```
Select Cortex-M0 code generation.

```
: Cortex-M1     cc/i    \ --
```

Select Cortex-M1 code generation.

`: Cortex-M3 cc/i \ --`
Select Cortex-M3 code generation.

`: Cortex-M4 cc/i \ --`
Select Cortex-M4 code generation with the integer DSP extension.

`: Cortex-M4F cc/i \ --`
Select Cortex-M4F code generation with the integer DSP and single precision VFP extensions.

`: Cortex-M7 cc/i \ --`
Select Cortex-M7 code generation with the integer DSP and single and double precision VFP extensions.

`: ARM-mode cc/i \ -- ; select 32 bit mode`
Selects generation of code using the ARM 32 bit instruction set and a register allocation model optimised for Thumb-2. This switch is for code generation with interworking between ARM32 and Thumb code.

`: Legacy-mode cc/i \ -- ; select 32 bit mode`
Selects generation of code using the ARM 32 bit instruction set and the register allocation model used by previous ARM compilers that do not support Thumb-2. Selecting this mode provides compatibility with previous ARM-only source code that may include assembler.

`: ARM7TDMI [+interpreter] Legacy-mode [previous] ;`
A synonym for `Legacy-mode`.

`: 32bit-mode [+interpreter] Legacy-mode [previous] ;`
A synonym for `Legacy-mode`.

`: +ARM32-Cortex-A/R cc/i ( -- ) <ARMunpredictable> on ;`
ARM32 code that must run on Cortex-A and Cortex-R cores must allow for increased restrictions in the register usage, especially for the LDRx and STRx instructions.

`: -ARM32-Cortex-A/R cc/i ( -- ) <ARMunpredictable> off ;`
Remove the restrictions introduced by `+ARM32-Cortex-A/R`.
To support standalone Forths in which RAM or auxiliary Flash may be more than 16/32 Mb away from the primary Flash area, optimisation of `>BODY` and `TO-DO` can be controlled by the `xLongCallsx` directives below. These are necessary for processors such as the Philips LPC families, in which the Flash and RAM locations in memory are fixed. Note that `+LongCalls` is only required for an interactive Forth or when Forth calls to code outside the normal 16/32Mb branch range are required.

`: +LongCalls cc/i \ --`
Permit long call sequences.

`: -LongCalls cc/i \ --`
Forbid long call sequences (default).

`: LongCalls? cc/i \ -- flag`
Long call sequences permitted?

`: inlining cc/i \ u -- ; base version with protection`
Specifies that words of less than *u* bytes can be binary inlinined. Use zero to disable binary inlining. Note that inlining interacts with the `+LeafCalls` directive. The use of `INLINING` is no longer recommended.
Cortex and ARM CPUs save subroutine return addresses in the LINK register. If a word calls

another word, the LINK register must be preserved, usually on the return stack. Faster and shorter code can be generated by only saving the LINK register as required. We refer to this as the `LeafCall` optimisation. All branches ensure that the LINK register is saved in case a later CALL/BL is made. If you know that a call will not be made, use `ISLEAF` before `: <name>` to inhibit the save of the LINK register, e.g.

```
IsLeaf : uDelay   \ n -- ; short delay
  begin  dup 1- 0= until
  drop
;
```

The default is `+LeafCalls`.

`: +LeafCalls cc/i \ -- ; enable late LINK saving`
When used, the LINK register is only saved if it has to be. Default.

`: -LeafCalls cc/i \ -- ; disable late LINK saving`
Forces colon definitions to save the LINK register on entry.

`: LeafCalls? cc/i \ -- flag ; test for late LINK saving`
Returns non-zero if late LINK saves are enabled.

`: IsLeaf cc/i \ -- ; next word is a LEAF`
Marks the next colon definition as being a leaf node, i.e. it makes no subroutine calls.

`: LoopAlignment cc/i \ n --`
Set loop starts, e.g. `BEGIN..XXX` and `DO..LOOP` to be aligned on an n-byte boundary, where n must be a power of two. This is useful to force the heads of loops onto a cache line or memory buffer boundary. The default is 4. For a Philips LPC CPU DO..LOOP performance can be improved 10-20% in some cases by using n=16.

```
  #16 LoopAlignment     \ set to 16 byte boundary
  0   LoopAlignment     \ revert to lowest setting
```

`: LoopAlignment? cc/i \ --`
Return the current loop alignment setting.

`: LoopAlign cc/i \ --`
Align with NOPs to head of loop. Used to align loop heads in assembler code.

`: +FastLVs cc/i \ -- ; enable fast local variables`
Enables generation of inline local variable entry code. This is the default condition, and is strongly recommended.

`: -FastLVs cc/i \ -- ; disable fast local variables`
Disables generation of inline local variable entry code.

`: +FixedLVs cc/i \ -- ; enable faster local variable entry code`
Enables generation of faster and shorter local variable entry code. The only penalty is that run-time adjustment of the stack frame is much more difficult and may require more stack space. This is the default condition, and is strongly recommended.

`: -FixedLvs cc/i \ -- ; disable faster local variable entry code`
Disables generation of faster/shorter inline local variable entry code.

`: SmallFrames cc/i \ -- ; max 512 bytes of LVs`
Cortex only: A local variable frame is restricted to a bit less than 512 bytes. This is enough for embedded systems use and results in smaller/faster entry code. Requires both `+FixedLVs` and `+FastLVs`.

`: LargeFrames cc/i \ -- ; max 4096 bytes of LVs`
Cortex only: A local variable frame is restricted to a bit less than 4096 bytes. This is enough for all hosted use. Requires both `+FixedLVs` and `+FastLVs`.

4.2 ARM and Cortex intrinsics

Intrinsics are what compiler people call special code generators that provide access to CPU-specific operations. The indicator `C_` in the names below is just for internal use by the compiler. The words you use are without `C_`.

The intrinsics in this section are available for all CPUs.

`c_arshift arshift \ x u -- x'`
Arithmetic right shift.

`c_ror ror \ x u -- x'`
Rotate right.

`fetchop c_w@s \ addr -- sw`
16 bit fetch and then sign extended.

`fetchop c_c@s \ addr -- sb`
8 bit fetch and then sign extended.

`addstoreop c_w+! \ n addr --`
As `+!` but for a 16 bit item.

`addstoreop c_c+! \ n addr --`
As `+!` but for an 8 bit item.

`addstoreop c_or! \ mask addr --`
`OR` the *mask* into the data at *addr*.

`addstoreop c_and! \ mask addr --`
`AND` the *mask* into the data at *addr*.

`addstoreop c_xor! \ mask addr --`
`XOR` the *mask* into the data at *addr*.

`addstoreop c_bic! \ mask addr --`
Clear the *mask* bits in the data at *addr*.

`addstoreop c_bor! \ bmask caddr --`
`OR` the *bmask* into the 8 bit data at *caddr*.

`addstoreop c_band! \ bmask caddr --`
`AND` the *bmask* into the 8 bit data at *caddr*.

`addstoreop c_bxor! \ bmask caddr --`
`XOR` the *bmask* into the 8 bit data at *caddr*.

`addstoreop c_bbic! \ bmask caddr --`
Clear the *bmask* bits in the 8 bit data at *caddr*.

`rUP  c_reg@ c_up@       \ -- addr`
Return the address of the `USER` area.

`rLP  c_reg@ c_lp@       \ -- addr`
Return the local variable pointer.

`rUP  c_reg! c_up!       \ addr --`
Set the `USER` area pointer.

`rLP c_reg! c_lp!        \ addr --`
Set the local variable pointer.

`: c_di          \ -- ; disable interrupts`
For Cortex parts, lays `CPS .ID .I`. For ARM parts, lays a call to the word `DI`.

`: c_ei          \ -- ; enable interrupts`
For Cortex parts, lays `CPS .IE .I`. For ARM parts, lays a call to the word `EI`.

`: c_dfi         \ -- ; disable fast interrupts`
For Cortex parts, lays `CPS .ID .F`. For ARM parts, lays a call to the word `DFI`.

`: c_efi         \ -- ; enable fast interrupts`
For Cortex parts, lays `CPS .IE .F`. For ARM parts, lays a call to the word `EFI`.

`: c_fsp!        \ -- ; set R8`
Set the floating point stack pointer R8, e.g.

```
<address> FSP!
```

`: c_fsp@        \ -- ; get R8`
Return the floating point stack pointer R8 in a new TOS.

4.3 Cortex-specific intrinsics

Intrinsics are what compiler people call special code generators that provide access to CPU-specific operations. The indicator `C_` in the names below is just for internal use by the compiler. The words you use are without `C_`.

`: c_sys@        \ -- ; <lit> SYS@`
Intrinsic for for MRS reg, SYSm. Use this to read a system register such as CONTROl or BASEPRI.

```
BASEPRI sys@
```

`: c_sys!        \ -- ; <val> <lit> SYS!`
Intrinsic for for MSR SYSm reg. Use this to write a system register.

```
2 CONTROL sys!
```

`: c_set-sp      \ -- ; <lit> SET-SP`
Used to set the Forth data stack pointer during initialisation. Unlike the normal Forth `SP!`, the previous top of stack is **not** saved. After use of `SET-SP`, you can assume nothing about the data stack except that it is empty.

`: c_ByteRevL    \ --`
Swap the bytes in a 32 bit long word.

`: c_ByteRevW    \ --`
Swap the bytes in the low 16 bits of TOS and swap the bytes in the upper 16 bits. This is safe to use to get a zero extended result immediately after `W@`.

`: c_ByteRevWS \ --`
Swap the bytes in the low 16 bits of TOS and sign extend them to 32 bits.

`: c_ByteRevWZ \ --`
Swap the bytes in the low 16 bits of TOS and zero extend it.

`: c_wfe \ --`
Flushes the stack and generates a `WFE .N` instruction.

`: c_wfi \ --`
Flushes the stack and generates a `WFI .N` instruction.

`: c_dsb#0F \ --`
Flushes the stack and generates a `DSB # $0F` instruction.

`: c_isb#0F \ --`
Flushes the stack and generates an `ISB # $0F` instruction.

4.4 Optimiser in-built macros

The ARM Cortex VFX optimiser includes a number of special cases which may not be present in the target code, but can be easily written if required. In the main, these are present to support peripheral register handling.

`BOR! BAND! BXOR! BBIC! ( mask addr -- )`
Logical operation on bytes (8 bits) in memory for CPUs with 8 bit peripheral registers.

`OR! AND! XOR! BIC! ( mask addr -- )`
Logical operation on words (32 bits) in memory for CPUs with 32 bit peripheral registers (the vast majority).

`ARSHIFT ROL ROR ( x count -- x' )`
Shift operations in the style of `LSHIFT`. `ARSHIFT` performs an arithmetic right shift. `ROL` and `ROR` are 32 bit circular left and right shifts.

4.5 Efficient coding practice

Writing good code for ARM and Cortex CPUs can double performance by reducing instruction count and memory accesses. This is particularly true for I/O operations involving bit masking. Although register operations are nearly always single-cycle operations, on most ARM/Cortex CPU implementations memory loads (LDRx instructions) for ARM take three cycles, Cortex loads take two cycles, taken branches need three cycles, and memory stores (STRx instructions) take two cycles. All of these have to be extended by any additional memory overhead such as the effect of running with wait states and/or 16 bit memory. A small amount of instruction cache can make a huge difference to systems with slow memory.

ARM CPUs are not good at handling literals as these are assembled 8 bits at a time. Thus an I/O address may result in four 32 bit instructions. To reduce this, the code generator places literals that cannot be coded in one instruction in a literal pool after the end of the word.

Cortex-M3/4 CPUs have better literal handling plus additional instructions for loading 16-bit immediates. A 32 bit literal can be loaded into a register in two instructions (8 bytes). For Cortex-M3/4, the code density benefit of a literal pool is minimal, but the performance cost is high.

The following code fragments illustrate the benefits of efficient coding. In MPE practice, absolute peripheral addresses are indicated by a leading underscore character '_', whereas offsets from a base address do not have an underscore. These examples are for an NXP LPC2106 CPU with an Ethernet controller connected to GPIO lines.

The first example is the simplistic operation.

```
: eth>read      \ -- ; turns data bus for reading
  _IODIR @
  $1FFE00FF and            \ set data bus to i/p
  $E00000F0 or             \ set others to o/p
  _IODIR !
;
dis eth>read
ETH>READ
ldr r0, [ pc, # $1C ]   ( @$B3CC = $E0028008 )
ldr r8, [ r0, # $00 ]
ldr r0, [ pc, # $18 ]   ( @$B3D0 = $1FFE00FF )
and r8, r8, r0
ldr r0, [ pc, # $14 ]   ( @$B3D4 = $E00000F0 )
orr r8, r8, r0
ldr r1, [ pc, # $10 ]   ( @$B3D8 = $E0028008 )
str r8, [ r1, # $00 ]
mov pc, r14
36 bytes, 9 instructions.
```

To this we have to add four 32 bit literals in the literal pool for a total of 12 words or 48 bytes (excluding the final return). At one cycle for the eight instructions we have to add two cycles each for the three loads (+6) and one cycle for the write (+1). The total is 48 bytes and 8+6+1=15 cycles.

The second example is written knowing that we have an ARM CPU and with understanding of the GPIO structure.

```
: erdmode       \ -- ; turns data bus for reading
  _GPIO
  dup IODIR + @
  edbmask invert and
  swap IODIR + !
;
dis erdmode
ERDMODE
ldr r8, [ pc, # $0C ]   ( @$B770 = $E0028000 )
ldr r7, [ r8, # $08 ]
bic r7, r7, # $FF00
str r7, [ r8, # $08 ]
mov pc, r14
20 bytes, 5 instructions.
```

Again ignoring the final return we have four effective instructions, one literal pool fetch and one store, resulting in 20 bytes (vs 48) and 4+2+1=7 cycles (vs 15). Removing the redundant OR has saved us four cycles and twelve bytes, and loading the base address (literal) once has saved us another four cycles and eight bytes.

Most peripheral operations require access to more than one register, e.g. command/status/data. Loading the base address once and then accessing individual registers by fixed offsets from the base allows the VFX code generator to optimise more away at the expense of slightly longer source code. Use of the built-in optimiser macros can reduce the expansion of the source code. Because the PC relative loads have a limited range, the code generator builds a new literal pool for each word. Most source files concentrate on a limited set of peripheral register base addresses. You can avoid the size penalty of the literal pool by providing a local pointer to the base address in the `CDATA` section and referencing it throughout the source file. This will be accessed by a PC relative load.

```
L: ^GPIOe
  _GPIO ,
: erdmode2      \ -- ; turns data bus for reading
  ^GPIOe @
  dup IODIR + @
  edbmask invert and
  swap IODIR + !
;
dis erdmode2
ERDMODE2
ldr r8, [ pc, # $-1C ]  ( @$B9F4 = $E0028000 )
ldr r7, [ r8, # $08 ]
bic r7, r7, # $FF00
str r7, [ r8, # $08 ]
mov pc, r14
20 bytes, 5 instructions.
```

Thus the rule of thumb is to use the Forth stack to `DUP` or `OVER` literals rather than to reference them several times as literals, and to avoid use of literals that do not fit in the ARM 4/8 bit literal format. By doing this you will also make better use of future improvements to the VFX code generator.

5 ARM Cortex Cross assembler

The MPE cross compiler has a built-in cross-assembler. This gives you the ability to define new Forth words in assembler as well as in Forth. You can also assemble code to anywhere in memory. This section is **not** a treatise on ARM Cortex assembly language programming. The essential document for Cortex-M is the ARMv7-M Architecture Reference Manual, which is available from `www.arm.com` as a PDF file document reference *ARM DDI 0403*. You may have to register to download it. The ARM 32 bit instruction set is documented in *ARM DDI 0100*. A copy is provided in the *Docs\ARM* folder in PDF format. The ARMv6 instruction set used in the Raspberry Pi is documented as an appendix to the ARMv7-A Architecture Reference Manual.

The full Thumb-2 instruction set for Cortex-M3/M4 is supported. The instruction notation is very similar to that provided with the MPE ARM compiler. The Cortex assembler notation is very close to that referred to by ARM as UAL (Unified Assembly Language), which was designed to improve portability between ARM and Cortex at the assembler source code level.

This assembler can be switched between Cortex and ARM instruction sets in order to support the Cortex-A profile. Note that the Thumb-2 instruction set can be regarded as an encoding of the ARM instruction set with better code density and features to support system programming. That ARM have achieved this objective is indicated by the fact that with only minor extensions to the compiler (see "Intrinsics" in the code generator chapter), there is no assembly code in the Forth start-up files.

By default, the assembler and VFX code generator are set to use the legacy ARM instruction set with the TOS register set to R10. This is the configuration used by VFX Forth for ARM Linux.

5.1 Why write in assembler?

Forth is compact and quick, so why write in assembler? An assembler definition is normally faster than a group of corresponding Forth words. For Cortex CPUs, hand coding can improve performance by keeping more data in registers in loops and by taking more advantage of conditional execution.

That having been said, MPE does not write Cortex or ARM code (even interrupt handlers) in assembler except in very rare cases.

5.2 Creating Forth words in assembler

Forth words can easily be defined in assembler. They increase the execution speed of your code and can sometimes make your code smaller.

5.2.1 Defining assembler words

Forth words written in assembler follow a similar form to a word written in Forth. Instead of a colon you have `CODE`. Instead of semi-colon you have `END-CODE`. For example:

```
CODE <name>
  ...
  ...
  NEXT,
END-CODE
```

creates a word called `<name>`. Any assembler code between the `CODE` and `END-CODE` will be assembled into the word. When executed, the macro `NEXT,` will stop the execution of the assembler and return to the calling word.

5.2.2 Writing assembler words

The syntax used for the opcodes has been kept similar to the standard ARM syntax. See the list at the end of the chapter for a comparison of ARM versus Forth syntax.

As a company, ARM changes or extends the instruction sets in use at any time as new cores are designed. By normal Forth standards, this assembler is a **huge** piece of code. Forgive us for deviations from the expected instruction syntax, in places our assembler notation reflects ease of implementation.

5.2.3 Register names

Registers are defined as follows.

`Rn`	The familiar integer registers R0..R15
`CRn`	Coprocessor registers CR0..CR15
`xPSR`	Processor status registers, e.g. CPSR and SPSR
`Sn`	Single-precision (32 bit) floating point or vector register S0..S31
`Dn`	Double-precision (64 bit) floating point or vector register D0..D15/31
`Qn`	Quadruple (128 bit) vector register, Q0..Q15

Note that the floating point and vector registers overlap. The mapping between the registers is as follows:

- S<2n> maps to the least significant half of D<n>
- S<2n+1> maps to the most significant half of D<n>
- D<2n> maps to the least significant half of Q<n>
- D<2n+1> maps to the most significant half of Q<n>

Where the instruction cannot distinguish between floating point and integer operation, use the

5.2.4 Preserving the Forth registers

The Forth interpreter and compiler use some of the target processor's registers. These must be preserved if they are used in the assembler. They can be saved on the stack, in memory or in other registers and restored at the end of the word. The ARM registers that are used are shown in the table below.

Cortex register usage

Cortex register	Forth register	Notes
R15 or PC	IP	The program counter. Altering this register will cause the processor to jump to a new address.
R14 or LINK	_	The link register. When a subroutine is entered via the BL instruction, the return address is cached in R14. If a further BL is to be executed within the subroutine, remember to save the contents of R14, usually on the return stack. Note that in Thumb mode, bit 0 will be set to 1 to indicate that the CPU is in Thumb mode.
R13 or SP	RSP	Forth return stack pointer, do not change this without good reason. This stack holds return addresses. Note that when entering a subroutine or word via the BL instruction the return address is cached in R14, the link register. R13 is used in many ARM systems as a stack pointer.
R12	PSP (1)	Forth data stack pointer, do not change this without good reason. Use it for passing parameters between words. When writing assembler code, use `PSP` rather than `R12`. Future versions of the compiler may use a different register for the data stack pointer.
R11	UP	Pointer to the base of the current User Area.
R10	TOS (1)	Currently the default register for TOS for ARM32, but this may change when interworking ARM and Cortex code.
R9	LP	Local variable frame pointer.
R8	–	Currently unused, but we have plans for it.
R7	TOS (2)	Instead of holding the top item of the Forth data stack in main memory, it is held in a register. This allows many simple operations to execute faster, and it also reduces the amount of memory traffic. For hosted systems such as VFX Forth for Linux, TOS will be in R10 by default, but for code density in the Thumb-2 instruction set, R7 is a better choice.
R6	PSP (2)	For best code density with the Thumb-2 instruction set, R6 is a better choice than R10 as the PSP.
R0..R6	_	Scratch

Table 5.1: Cortex Register usage

ARM register usage

ARM register	Forth register	Notes
R15 or PC	IP	The ARM program counter. Altering this register will cause the processor to jump to a new address.
R14 or LINK	_	The ARM link register. When a subroutine is entered via the BL instruction, the return address is cached in R14. If a further BL is to be executed within the subroutine, remember to save the contents of R14, usually on the return stack.
R13 or SP	RSP	Forth return stack pointer, do not change this without good reason. This stack holds return addresses. Note that when entering a subroutine or word via the BL instruction the return address is cached in R14, the link register. R13 is used in many ARM systems as a stack pointer.
R12	PSP	Forth data stack pointer, do not change this without good reason. Use it for passing parameters between words.
R11	UP	Pointer to the base of the current User Area.
R10	TOS	Instead of holding the top item of the Forth data stack in main memory, it is held in a register. This allows many simple operations to execute faster, and it also reduces the amount of memory traffic.
R9	LP	Local variable frame pointer.
R0..R8	_	Scratch

Table 5.2: ARM Register usage

5.2.5 Executing an assembler word

A Forth word written in assembler is executed in the same way as a word written in Forth. It is executed in the same way as a normal word, by stating its name.

5.3 Assembling into memory

Assembler code can be assembled into memory and not in a Forth word. To do this you need to:

- turn on the assembler
- write your assembler code
- turn off the assembler

To turn on the assembler, use the word `AsmCode`. To switch back to Forth use the word `End-Code`. Between these two words, any assembler will be assembled. The assembled code will be placed in the dictionary without a header. The code can be executed by the use of labels. This is often used to define low-level interrupts. See the chapter on Interrupts for more details on writing low-level interrupts.

5.4 Creating defining words in assembler

The cross compiler allows you to define the run-time (`DOES>`) part of a defining word in assembler. To do this use `;CODE` in the form:

```
: <name>
  CREATE
    ...
  ;CODE
    ...
END-CODE
```

An example is shown below:

```
: VARIABLE    \ <spaces>name -- ; -- addr
  CREATE                    \ Create header
    0 ,                     \ Initial value
  ;CODE                     \ Run-time action
\ Cortex version
    str tos, [ psp, # -4 ] ! \ save TOS
    ldr tos, [ link, # -1 ]  \ get const val/addr from afer BL
    pop     { pc }
END-CODE
\ ARM version
    stmfd psp ! { tos }      \ Save TOS
    ldr tos, [ link ], # 4   \ get pointer to data
    ldr tos, [ tos ]         \ get variable address
    NEXT,
END-CODE
```

5.5 Structured programming

Three facilities are available to give you the advantages of structured programming, in assembler:

- control structures
- labels
- local labels

5.5.1 Control structures

There are assembler equivalents to the Forth control structures. The available structures are:

```
AHEAD, ... THEN, or ENDIF,
cc IF, ... THEN, or ENDIF,
cc IF, ... ELSE, ... THEN, or ENDIF,
BEGIN, ... cc UNTIL,
BEGIN, ... cc WHILE, ... REPEAT
BEGIN, ... AGAIN,
```

where cc is one of the condition codes in the table below.

ARM	Forth	Condition	ARM	Forth	Condition
.CS	CS,	carry set	.NE	NE,	not equal or non-zero
.CC	CC,	carry clear	.GE	GE,	greater than or equal
.PL	PL,	plus - positive or zero	.LT	LT,	less than
.MI	MI,	minus - negative	.GT	GT,	greater than
.VS	VS,	overflow set	.LS	LS,	unsigned less than or equal (same)
.VC	VC,	overflow clear	.HS	HS,	unsigned greater than or equal (same). Same as CS
.LE	LE,	less than or equal	.LO	LO,	unsigned less than. Same as CC
.EQ	EQ,	equal or zero	.HI	HI,	unsigned greater than
.AL	* Always (default)				

Table 5.3: Condition code mnemonics

Cortex IT instruction

The Thumb-2 instruction set does not support the conditional execution facilities of the ARM instruction set. Instead, it provides the `IT` instruction.

In order to avoid performance penalties caused by taken branches and associated cache flushes, the Thumb-2 instruction set provides the `ITxxx <cond>` instruction. This permits up to four instructions to be executed depending on the condition flags at the start. The first instruction is executed if <cond> is true. The next three instructions are executed if <cond> is true and x=T or if <cond> is false and x=E. The following example illustrates the use of the IT instruction.

```
CODE WITHIN?     \ n1 n2 n3 -- flag
\ Return TRUE if N1 is within the range N2..N3.
\ This word uses signed arithmetic.
  ldmfd   psp ! { r0, r1 }
  mov .s  r2, # 0
  mov .s  r3, # 0
  cmp     r1, r0
  it .ge  \ next instruction if condition met
    mov     r2, # 1
  cmp     r1, tos
  it .le  \ next instruction if condition met
    mov     r3, # 1
  tst     r2, r3
  ite .ne
    mvn     tos, # 0  \ if condition met
    mov     tos, # 0  \ if condition not met
  next,
END-CODE
```

ARM conditional instructions

The ARM is different from many processors in that many instructions can be executed conditionally depending on the processor status flags, by appending one of the mnemonics in the table above to the instruction. An instruction without a condition suffix is assumed to use `.AL`. Note that most instructions (except the test and compare instructions) do not set the status flags by default. This has to be done with the `.S` suffix:

```
ADD .S R0, R1, R2          \ Add, set condition codes
ADD .NE .S R0, R1, R2      \ if NE and set condition codes

CS, IF,                    \ do between IF, and ENDIF, if CS set
  ...
ENDIF,
```

It is often quicker to avoid short jumps in code such as those typically generated by `IF,` statements, by the use of conditionally executed instructions. Skipping several instructions is generally faster than using a branch instruction as this involves flushing the processor pipeline. See the file *CODEARM.FTH* for examples of conditional execution.

5.5.2 Labels

Labels can be used to mark a place in assembler code. That place can then be referenced in other areas of code.

Creating a label

Labels can be defined by using the command `L: <name>`. It is used in the form:

`l: <name>`

where `<name>` is what you want to call the label.

Referencing a label

A label is referenced by stating its name. For example,

```
B .EQ <name>
```

5.5.3 Local labels

If you need to use labels within a code definition, you may use the local labels provided. These are used just as normal labels in the assembler, but some restrictions apply:

- there is a maximum of ten labels
- the names are in the form L$n where n is in the range 1 to 10
- a reference is valid until the next occurrence of `CODE` or `;CODE`.

Creating a local label

```
L$1:
```

Referencing a local label

To reference a local label, type its name. For example,

```
B L$1
```

assembles code for a branch to L$1:.

5.6 Creating macros

A macro is a word that lays down code 'in-line' within an assembler definition. Macros are used when there is a repetitive use of a series of opcodes.

5.6.1 Defining a macro

The easiest way to create a macro is by using `MACRO:`. The macro below can be used as a divide step operation.

```
macro: Udiv63/31_step   \ --
  adc .s  r1, r1, r1
  adc .s  r0, tos, r0, lsl # 1
  sub .cc r0, r0, tos
;m
```

The assembler's prefix notation leads to some peculiarities when writing macros.

1. Opcode words, e.g. `ADC` above, assemble the code for the **previous** instruction, so you may not know the stack conditions. Put literal data on the return stack in the macro, and retrieve it as needed.
2. The assembler wordlist is first in the search order, so be careful that a normal Forth word name is not a name in the assembler.
3. Register names are executable words. To pass registers by number use `R# ( n -- )` for general purpose registers, `CR# ( n -- )` for coprocessor registers, and `SR# ( n -- )` for status registers (0=CPSR, 1=SPSR).

A macro can also be defined using colon and semi-colon before CROSS-COMPILE is executed. It must be defined in the cross compiler's ASM-ACCESS vocabulary. The place to create a macro is in the control file and it must be defined before CROSS-COMPILE. As an example, the macro NEXT, is shown below. NEXT, is defined as a macro, so each time it is used, its code is laid down. This makes it quicker than calling a subroutine.

```
\ switch to cross compilers assembler vocab
FORTH ALSO C-C ALSO ASSEMBLER
ALSO ASM-ACCESS DEFINITIONS
\ define NEXT
: NEXT,   \ -- ; lay in-line next code
  bx  lr
;
\ switch back to normal forth vocabulary
ONLY FORTH DEFINITIONS
```

5.6.2 Using a macro

A macro is used by stating its name. For example, in a CODE definition, NEXT, is a macro.

5.7 Debugging

It is possible to disassemble compiled words using:

```
XDASM <name>

DIS <name>
```

This can be done during compilation by including an XDASM statement in the control file, or interactively after compilation by including the word INTERACTIVE before FINIS.

5.8 CPU selection

The instruction set of the processor is extended on various processor cores. The selection available in this assembler is for ARM32 with/without Thumb-1 and the Cortex-M0, M1, M3 and M4.

5.9 Number bases

The number base in the Forth assembler can be indicated by BINARY, DECIMAL, and HEX. In addition, numbers prefixed by the '$' '#' and '%' characters are treated as special cases. These characters affect the number base for that number only. Note that the characters '$' and '%' follow Motorola usage. Note also that the '#' symbol attached to a number is not the same as the word # word that indicates immediate addressing.

Symbol	Base	Example
$	hex	$55AA
#	decimal	#1234
%	binary	%1011001

Table 5.4: Number bases

5.10 Thumb-2 instruction set

Thumb-2 literals are encoded in a number of ways. The term *imm* refers to an immediate value that is directly encoded across a number of bit fields. The term *const* refers to an immediate value that comes from a restricted but wide range. See *ARM DDI 0403* for the gory details. Items in square brackets are optional, e.g. `[.s]` indicates that `.s` to set the conditions flags is optional.

When one of the `ITxxx` instruction applies, **do not** apply `.s`. The assembler will do what it has to.

The term <shift> means one of

```
.LSL # n    logical left shift by n
.LSR # n    logical right shift by n
.ASR # n    arithmetic right shift by n
.ROR # n    rotate right by n
.RRX        1 bit rotate right, carry in to new bit 31,
            old bit 0 to carry out.
```

You can leave out <shift> and it will be encoded as LSL # 0. Note that when .S applies, the rules for the final value of the carry flag are a bit arcane.

You can force an instruction to use the 16 bit form by using the `.n` indicator. You can force the 32 bit form using the `.w` indicator. Without either of these, the assembler will choose the shortest form. Using R0..R7 (the low registers) and .S usually generates the shortest code. Except in a few cases, using R8..R15 (the high registers) will generate a 32 bit instruction.

Where a register appears twice in an instruction (usually Rd), that particular encoding (usually a 16 bit form) is only generated when the same register appears twice.

Use of the PC (R15) or SP (R13) registers may cause assembler errors as these register fields are frequently used to handle special instructions. Consult *ARM DDI 0403*.

5.10.1 Base instruction set

This is the Cortex-M3 instruction set. Note that the Cortex-M0/M1 instruction set is a subset of this. In ARM terminology the Cortex-M0/M1 is defined in ARMv6-M.

In some cases using `.s` leads to shoter code because a 16 bit Thumb-1 encoding is available. When coding for Cortex-M0 remember that the `.s` is often required.

```
ADC [.s]    Rd, Rn, # <const>
ADC [.s]    Rd, Rn, Rm
ADC [.s]    Rd, Rn, Rm <shift>

ADD .s      Rd, Rn, # <imm3>
ADD .s      Rd, Rd, # <imm8>
ADD [.s]    Rd, Rn, # <const>
ADD [.s]    Rd, Rn, # <imm12>
ADD .s      Rd, Rn, Rm
ADD         Rd, Rd, Rm
ADD [.s]    Rd, Rn, Rm <shift>
ADD         Rd, SP, # <imm8>
ADD         SP, SP, # <imm7>
ADD [.s]    Rd, SP, # <const>
ADD [.s]    Rd, SP, # <imm12>
ADD         Rd, SP, Rd
ADD         SP, SP, Rm
ADD         Rd, SP, Rm <shift>

ADR         Rd, <label>

AND         Rd, Rn, # <const>
AND         Rd, Rd, Rm
AND [.s]    Rd, Rn, Rm <shift>

ASR .s      Rd, Rn, # <imm5>
ASR [.s]    Rd, Rn, # <imm5>
ASR .s      Rd, Rd, Rm
ASR [.s]    Rd, Rn, Rm
```

```
B <cond>    <label>
B           <label>

BFC         Rd, # <lsb> # <width>
BFI         Rd, Rn, # <lsb> # <width>

BIC [.s]    Rd, Rn, # <const>
BIC .s      Rd, Rd, Rm
BIC [.s]    Rd, Rn, Rm <shift>

BKPT        # <imm8>

BL          <label>

BLX         Rm

BX          Rm
```

```
CBNZ        Rn, <label>
CBZ         Rn, <label>

CDP         <copro> <opc1> CRd, CRn, CRm, <opc2>
CDP2        <copro> <opc1> CRd, CRn, CRm, <opc2>

CLREX

CLZ         Rd, Rm

CMN         Rn, # <const>
CMN         Rn, Rm
CMN         Rn, Rm <shift>

CMP         Rn, # <imm8>
CMP         Rn, # <const>
CMP         Rn, Rm
CMP         Rn, Rm <shift>

CPS .ie     [.i] [.f]
CPS .id     [.i] [.f]
```

```
DBG         # <opt4>
DMB         # <opt4>
DSB         # <opt4>

EOR [.s]    Rd, Rn, # <const>
EOR .s      Rd, Rd, Rm
EOR [.s]    Rd, Rn, Rm <shift>

ISB         # <opt4>

IT <cond>
IT <cond>
ITT <cond>
ITE <cond>
ITTT <cond>
ITET <cond>
ITTE <cond>
ITEE <cond>
ITTTT <cond>
ITETT <cond>
ITTET <cond>
ITEET <cond>
ITTTE <cond>
ITETE <cond>
ITTEE <cond>
ITEEE <cond>
```

```
LDC         <copro> CRd, [ Rn, # +/-<imm8> ]
LDC         <copro> CRd, [ Rn, # +/-<imm8> ] !
LDC         <copro> CRd, [ Rn ], # +/-<imm8>
LDC         <copro> CRd, [ Rn ], {} <option>
LDC         <copro> CRd, <label>
LDCL        <copro> CRd, [ Rn, # +/-<imm8> ]
LDCL        <copro> CRd, [ Rn, # +/-<imm8> ] !
LDCL        <copro> CRd, [ Rn ], # +/-<imm8>
LDCL        <copro> CRd, [ Rn ], {} <option>
LDCL        <copro> CRd, <label>
LDC2        <copro> CRd, [ Rn, # +/-<imm8> ]
LDC2        <copro> CRd, [ Rn, # +/-<imm8> ] !
LDC2        <copro> CRd, [ Rn ], # +/-<imm8>
LDC2        <copro> CRd, [ Rn ], {} <option>
LDC2        <copro> CRd, <label>
LDC2L       <copro> CRd, [ Rn, # +/-<imm8> ]
LDC2L       <copro> CRd, [ Rn, # +/-<imm8> ] !
LDC2L       <copro> CRd, [ Rn ], # +/-<imm8>
LDC2L       <copro> CRd, [ Rn ], {} <option>
LDC2L       <copro> CRd, <label>

LDM         Rn, [!] { ra, rb ... rn }
LDMIA       Rn, [!] { ra, rb ... rn }
LDMFD       Rn, [!] { ra, rb ... rn }
LDMDB       Rn, [!] { ra, rb ... rn }
LDMEA       Rn, [!] { ra, rb ... rn }
```

```
LDR         Rt, [ Rn, # <imm5*4> ]
LDR         Rt, [ SP, # <imm8*4> ]
LDR         Rt, [ Rn, # <imm12> ]
LDR         Rt, [ Rn, # -<imm8> ]
LDR         Rt, [ Rn ], # +/-<imm8>
LDR         Rt, [ Rn, # +/-<imm8> ] !
LDR         Rt, <label>
LDR         Rt, [ Rn ++ Rm ]
LDR         Rt, [ Rn ++ Rm, LSL # <imm2> ]
LDR         Rt, @= <imm32>    \ loads from literal pool
                              \ Use FLUSHLITPOOL to lay pool.

LDRB        Rt, [ Rn, # <imm5> ]
LDRB        Rt, [ Rn, # <imm12> ]
LDRB        Rt, [ Rn, # -<imm8> ]
LDRB        Rt, [ Rn ], # +/-<imm8>
LDRB        Rt, [ Rn, # +/-<imm8> ] !
LDRB        Rt, <label>
LDRB        Rt, [ Rn ++ Rm ]
LDRB        Rt, [ Rn ++ Rm, LSL # <imm2> ]

LDRBT       Rt, [ Rn, # +<imm8> ]

LDRD        Rt, Rt2, [ Rn, # -<imm8> ]
LDRD        Rt, Rt2, [ Rn ], # +/-<imm8>
LDRD        Rt, Rt2, [ Rn, # +/-<imm8> ] !
LDRD        Rt, Rt2, <label>

LDREX       Rt, [ Rn, # +<imm8> ]
LDREXB      Rt, [ Rn ]
LDREXH      Rt, [ Rn ]

LDRH        Rt, [ Rn, # <imm5*2> ]
LDRH        Rt, [ Rn, # <imm12> ]
LDRH        Rt, [ Rn, # -<imm8> ]
LDRH        Rt, [ Rn ], # +/-<imm8>
LDRH        Rt, [ Rn, # +/-<imm8> ] !
LDRH        Rt, <label>
LDRH        Rt, [ Rn ++ Rm ]
LDRH        Rt, [ Rn ++ Rm, LSL # <imm2> ]

LDRHT       Rt, [ Rn, # +<imm8> ]
```

```
LDRSB       Rt, [ Rn, # <imm12> ]
LDRSB       Rt, [ Rn, # -<imm8> ]
LDRSB       Rt, [ Rn ], # +/-<imm8>
LDRSB       Rt, [ Rn, # +/-<imm8> ] !
LDRSB       Rt, <label>
LDRSB       Rt, [ Rn ++ Rm ]
LDRSB       Rt, [ Rn ++ Rm, LSL # <imm2> ]

LDRSBT      Rt, [ Rn, # +<imm8> ]

LDRSH       Rt, [ Rn, # <imm12> ]
LDRSH       Rt, [ Rn, # -<imm8> ]
LDRSH       Rt, [ Rn ], # +/-<imm8>
LDRSH       Rt, [ Rn, # +/-<imm8> ] !
LDRSH       Rt, <label>
LDRSH       Rt, [ Rn ++ Rm ]
LDRSH       Rt, [ Rn ++ Rm, LSL # <imm2> ]

LDRSHT      Rt, [ Rn, # +<imm8> ]

LDRT        Rt, [ Rn, # +<imm8> ]

LSL         Rd, Rm, # <imm5>
LSL [.s]    Rd, Rm, # <imm5>
LSL .s      Rd, Rd, Rm
LSL [.s]    Rd, Rn, Rm

LSR         Rd, Rm, # <imm5>
LSR [.s]    Rd, Rm, # <imm5>
LSR .s      Rd, Rd, Rm
LSR [.s]    Rd, Rn, Rm
```

```
MCR         <copro> <opc1> Rt, CRn, CRm, <opc2>
MCR2        <copro> <opc1> Rt, CRn, CRm, <opc2>
MCRR        <copro> <opc1> Rt, Rt2, CRm, <opc2>
MCRR2       <copro> <opc1> Rt, Rt2, CRm, <opc2>

MLA         Rd, Rn, Rm, Ra
MLS         Rd, Rn, Rm, Ra

MOV .s      Rd, # <imm8>
MOV [.s]    Rd, # <const>
MOV         Rd, # <imm16>
MOV         Rd, Rm
MOV [.s]    Rd, Rm
MOV [.s]    Rd, Rm <shiftop> # n
MOV [.s]    Rd, Rm <shiftop> Rs
MOV [.s]    Rd, Rm .RRX
MOVT        Rd, # <imm16>

MRC         <copro> <opc1> Rt, CRn, CRm, <opc2>
MRC2        <copro> <opc1> Rt, CRn, CRm, <opc2>
MRRC        <copro> <opc1> Rt, Rt2, CRm, <opc2>
MRRC2       <copro> <opc1> Rt, Rt2, CRm, <opc2>

MRS         Rd, <SYSm8>
MSR         <SYSm8> Rn
<SYSm8> is a special register number in the range 0..255

MUL .s      Rd, Rn, Rd
MUL [.s]    Rd, Rn, Rm

MVN [.s]    Rd, # <const>
MVN .s      Rd, Rm
MVN [.s]    Rd, Rm <shift>
```

```
NEG .s      Rd, Rm

NOP [.n/.w]

ORN [.s]    Rd, Rm, # <const>
ORN [.s]    Rd, Rn, Rm <shift>

ORR [.s]    Rd, Rm, # <const>
ORR .s      Rd, Rd, Rm
ORR [.s]    Rd, Rn, Rm <shift>

PLD         [ Rn, # +<imm12> ]
PLD         [ Rn, # -<imm8> ]
PLD         <label>
PLD         [ Rn ++ Rm, LSL # <imm2> ]
PLDW        [ Rn, # +<imm12> ]
PLDW        [ Rn, # -<imm8> ]
PLI         [ Rn, # +<imm12> ]
PLI         [ Rn, # -<imm8> ]
PLI         <label>
PLI         [ Rn ++ Rm, LSL # <imm2> ]

POP         { ra, rb ... rn }
PUSH        { ra, rb ... rn }

RBIT        Rd, Rm
REV         Rd, Rm
REV16       Rd, Rm
REVSH       Rd, Rm

ROR [.s]    Rd, Rm, # <imm5>
ROR .s      Rd, Rd, Rm
ROR [.s]    Rd, Rn, Rm

RRX         Rd, Rm
```

```
RSB .s      Rd, Rn, # 0
RSB [.s]    Rd, Rn, # <const>
RSB [.s]    Rd, Rn, Rm <shift>

SBC [.s]    Rd, Rn, # <const>
SBC .s      Rd, Rd, Rm
SBC [.s]    Rd, Rn, Rm <shift>

SBFX        Rd, Rn, # <lsb> # <width>

SDIV        Rd, Rn, Rm

SEV

SMLAL       Rdlo, Rdhi, Rn, Rm
SMULL       Rdlo, Rdhi, Rn, Rm

SSAT        Rd, # <imm5> Rn <shift>

STC         <copro> CRd, [ Rn, # +/-<imm8> ]
STC         <copro> CRd, [ Rn, # +/-<imm8> ] !
STC         <copro> CRd, [ Rn ], # +/-<imm8>
STC         <copro> CRd, [ Rn ], {} <option>
STCL        <copro> CRd, [ Rn, # +/-<imm8> ]
STCL        <copro> CRd, [ Rn, # +/-<imm8> ] !
STCL        <copro> CRd, [ Rn ], # +/-<imm8>
STCL        <copro> CRd, [ Rn ], {} <option>
STC2        <copro> CRd, [ Rn, # +/-<imm8> ]
STC2        <copro> CRd, [ Rn, # +/-<imm8> ] !
STC2        <copro> CRd, [ Rn ], # +/-<imm8>
STC2        <copro> CRd, [ Rn ], {} <option>
STC2L       <copro> CRd, [ Rn, # +/-<imm8> ]
STC2L       <copro> CRd, [ Rn, # +/-<imm8> ] !
STC2L       <copro> CRd, [ Rn ], # +/-<imm8>
STC2L       <copro> CRd, [ Rn ], {} <option>

STM         Rn, [!] { ra, rb ... rn }
STMIA       Rn, [!] { ra, rb ... rn }
STMEA       Rn, [!] { ra, rb ... rn }
STMDB       Rn, [!] { ra, rb ... rn }
STMFD       Rn, [!] { ra, rb ... rn }
```

```
STR         Rt, [ Rn, # <imm5*4> ]
STR         Rt, [ SP, # <imm8*4> ]
STR         Rt, [ Rn, # <imm12> ]
STR         Rt, [ Rn, # -<imm8> ]
STR         Rt, [ Rn ], # +/-<imm8>
STR         Rt, [ Rn, # +/-<imm8> ] !
STR         Rt, [ Rn ++ Rm ]
STR         Rt, [ Rn ++ Rm, LSL # <imm2> ]

STRB        Rt, [ Rn, # <imm5> ]
STRB        Rt, [ Rn, # <imm12> ]
STRB        Rt, [ Rn, # -<imm8> ]
STRB        Rt, [ Rn ], # +/-<imm8>
STRB        Rt, [ Rn, # +/-<imm8> ] !
STRB        Rt, [ Rn ++ Rm ]
STRB        Rt, [ Rn ++ Rm, LSL # <imm2> ]

STRBT       Rt, [ Rn, # <imm8> ]

STRD        Rt, Rt2, [ Rn, # -<imm8> ]
STRD        Rt, Rt2, [ Rn ], # +/-<imm8>
STRD        Rt, Rt2, [ Rn, # +/-<imm8> ] !

STREX       Rd, Rt, [ Rn, # +<imm8> ]
STREXB      Rd, Rt, [ Rn ]
STREXH      Rd, Rt, [ Rn ]

STRH        Rt, [ Rn, # <imm5*2> ]
STRH        Rt, [ Rn, # <imm12> ]
STRH        Rt, [ Rn, # -<imm8> ]
STRH        Rt, [ Rn ], # +/-<imm8>
STRH        Rt, [ Rn, # +/-<imm8> ] !
STRH        Rt, [ Rn ++ Rm ]
STRH        Rt, [ Rn ++ Rm, LSL # <imm2> ]

STRHT       Rt, [ Rn, # +<imm8> ]

STRT        Rt, [ Rn, # <imm8> ]
```

```
SUB .s      Rd, Rn, # <imm3>
SUB .s      Rd, Rd, # <imm8>
SUB [.s]    Rd, Rn, # <const>
SUB [.s]    Rd, Rn, # <imm12>
SUB .s      Rd, Rn, Rm
SUB [.s]    Rd, Rn, Rm <shift>
SUB         SP, SP, # <imm7*4>
SUB [.s]    Rd, SP, # <const>
SUB [.s]    Rd, SP, # <imm12>
SUB         Rd, SP, Rm <shift>

SVC         # <imm8>

SXTB        Rd, Rm
SXTB        Rd, Rm, <rotation>  ( 0/8/16/24 bits )
SXTH        Rd, Rm
SXTH        Rd, Rm, <rotation>  ( 0/8/16/24 bits )

TBB         [ Rn, Rm ]
TBH         [ Rn, Rm ]
```

```
TEQ         Rn, # <const>
TEQ         Rn, Rm <shift>

TST         Rn, # <const>
TST         Rn, Rm
TST         Rn, Rm <shift>

UBFX        Rd, Rn, # <lsb> # <width>

UDIV        Rd, Rn, Rm

UMLAL       Rdlo, Rdhi, Rn, Rm
UMULL       Rdlo, Rdhi, Rn, Rm

USAT        Rd, # <imm5> Rn <shift>

UXTB        Rd, Rm
UXTB        Rd, Rm, <rotation>  ( 0/8/16/24 bits )
UXTH        Rd, Rm
UXTH        Rd, Rm, <rotation>  ( 0/8/16/24 bits )

WFE
WFI
YIELD
```

5.10.2 Integer DSP

These instructions are integer DSP instructions added to Cortex-M4.

```
PKHBT       Rd, Rn, Rm
PKHBT       Rd, Rn, Rm .lsl # <imm5>
PKHTB       Rd, Rn, Rm
PKHTB       Rd, Rn, Rm .asr # <imm5>
```

```
QADD        Rd, Rn, Rm
QADD16      Rd, Rn, Rm
QADD8       Rd, Rn, Rm
QASX        Rd, Rn, Rm
QDADD       Rd, Rn, Rm
QDSUB       Rd, Rn, Rm
QSAX        Rd, Rn, Rm
QSUB        Rd, Rn, Rm
QSUB16      Rd, Rn, Rm
QSUB8       Rd, Rn, Rm
```

```
SADD16      Rd, Rn, Rm
SADD8       Rd, Rn, Rm
SASX        Rd, Rn, Rm
SEL         Rd, Rn, Rm
SHADD16     Rd, Rn, Rm
SHADD8      Rd, Rn, Rm
SHASX       Rd, Rn, Rm
SHSAX       Rd, Rn, Rm
SHSUB16     Rd, Rn, Rm
SHSUB8      Rd, Rn, Rm
```

```
SMLABB      Rd, Rn, Rm, Ra
SMLABT      Rd, Rn, Rm, Ra
SMLATB      Rd, Rn, Rm, Ra
SMLATT      Rd, Rn, Rm, Ra
SMLAD       Rd, Rn, Rm, Ra
SMLADX      Rd, Rn, Rm, Ra
SMLALBB     Rd, Rn, Rm, Ra
SMLALBT     Rd, Rn, Rm, Ra
SMLALTB     Rd, Rn, Rm, Ra
SMLALTT     Rd, Rn, Rm, Ra
SMLALD      RdLo, RdHi, Rn, Rm
SMLALDX     RdLo, RdHi, Rn, Rm
SMLAWB      Rd, Rn, Rm, Ra
SMLAWT      Rd, Rn, Rm, Ra
SMLSD       Rd, Rn, Rm, Ra
SMLSDX      Rd, Rn, Rm, Ra
SMLSLD      RdLo, RdHi, Rn, Rm
SMLSLDX     RdLo, RdHi, Rn, Rm
```

```
SMMLA       Rd, Rn, Rm, Ra
SMMLAR      Rd, Rn, Rm, Ra
SMMLS       Rd, Rn, Rm, Ra
SMMLSR      Rd, Rn, Rm, Ra
SMMUL       Rd, Rn, Rm
SMMULR      Rd, Rn, Rm
SMUAD       Rd, Rn, Rm
SMUADX      Rd, Rn, Rm
SMULBB      Rd, Rn, Rm
SMULBT      Rd, Rn, Rm
SMULTB      Rd, Rn, Rm
SMULTT      Rd, Rn, Rm
SMULWB      Rd, Rn, Rm
SMULWT      Rd, Rn, Rm
SMUSD       Rd, Rn, Rm
SMUSDX      Rd, Rn, Rm
```

```
SSAT16      Rd, # <imm4> Rn
SSAX        Rd, Rn, Rm
SSUB16      Rd, Rn, Rm
SSUB8       Rd, Rn, Rm
SXTAB       Rd, Rn, Rm
SXTAB       Rd, Rn, Rm, <rotation>  ( 0/8/16/24 bits )
SXTAB16     Rd, Rn, Rm
SXTAB16     Rd, Rn, Rm, <rotation>  ( 0/8/16/24 bits )
SXTAH       Rd, Rn, Rm
SXTAH       Rd, Rn, Rm, <rotation>  ( 0/8/16/24 bits )
SXTB16      Rd, Rm
SXTB16      Rd, Rm, <rotation>  ( 0/8/16/24 bits )
```

```
UADD16      Rd, Rn, Rm
UADD8       Rd, Rn, Rm
UASX        Rd, Rn, Rm
UHADD16     Rd, Rn, Rm
UHADD8      Rd, Rn, Rm
UHASX       Rd, Rn, Rm
UHSAX       Rd, Rn, Rm
UHSUB16     Rd, Rn, Rm
UHSUB8      Rd, Rn, Rm
UMAAL       RdLo, RdHi, Rn, Rm
```

```
UQADD16     Rd, Rn, Rm
UQADD8      Rd, Rn, Rm
UQASX       Rd, Rn, Rm
UQSAX       Rd, Rn, Rm
UQSUB16     Rd, Rn, Rm
UQSUB8      Rd, Rn, Rm
```

```
USAD8       Rd, Rn, Rm
USADA8      Rd, Rn, Rm, Ra
USAT16      Rd, # <imm4> Rn
USAX        Rd, Rn, Rm
USUB16      Rd, Rn, Rm
USUB8       Rd, Rn, Rm
```

```
UXTAB       Rd, Rn, Rm
UXTAB       Rd, Rn, Rm, <rotation>  ( 0/8/16/24 bits )
UXTAB16     Rd, Rn, Rm
UXTAB16     Rd, Rn, Rm, <rotation>  ( 0/8/16/24 bits )
UXTAH       Rd, Rn, Rm
UXTAH       Rd, Rn, Rm, <rotation>  ( 0/8/16/24 bits )
UXTB16      Rd, Rm
UXTB16      Rd, Rm, <rotation>  ( 0/8/16/24 bits )
```

5.10.3 Floating point for Cortex-Mx and ARM32

These are single-precision floating point instructions added for the Cortex-M4F instruction set upwards and for the ARM32 instruction set. Double precision instructions are available for systems that support them.

```
VABS .f32   Sd, Sm
VABS .f64   Dd, Dm
VADD .f32   Sd, Sn, Sm
VADD .f64   Dd, Dn, Dm
VCMP .f32   Sd, Sm
VCMP .f32   Sd, # 0
VCMP .f64   Dd, Dm
VCMP .f64   Dd, # 0
VCMPE .f32  Sd, Sm
VCMPE .f32  Sd, # 0
VCMPE .f64  Dd, Dm
VCMPE .f64  Dd, # 0
```

```
VCVT .si32<f32     Sd, Sm
VCVT .ui32<f32     Sd, Sm
VCVT .f32<si32     Sd, Sm
VCVT .f32<ui32     Sd, Sm
VCVT .si32<f64     Sd, Dm
VCVT .ui32<f64     Sd, Dm
VCVT .f64<si32     Dd, Sm
VCVT .f64<ui32     Dd, Sm
VCVT .xs32<f32     Sd, Sd, # fbits
VCVT .xu32<f32     Sd, Sd, # fbits
VCVT .xs16<f32     Sd, Sd, # fbits
VCVT .xu16<f32     Sd, Sd, # fbits
VCVT .xs32<f64     Dd, Dd, # fbits
VCVT .xu32<f64     Dd, Dd, # fbits
VCVT .xs16<f64     Dd, Dd, # fbits
VCVT .xu16<f64     Dd, Dd, # fbits
VCVT .f32<xs32     Sd, Sd, # fbits
VCVT .f32<xu32     Sd, Sd, # fbits
VCVT .f32<xs16     Sd, Sd, # fbits
VCVT .f32<xu16     Sd, Sd, # fbits
VCVT .f64<xs32     Dd, Dd, # fbits
VCVT .f64<xu32     Dd, Dd, # fbits
VCVT .f64<xs16     Dd, Dd, # fbits
VCVT .f64<xu16     Dd, Dd, # fbits
VCVT .f32<f16      Qd, Dm             \ ASIMD
VCVT .f16<f32      Dd, Qm             \ ASIMD
VCVT .f64<f32      Dd, Sm
VCVT .f32<f64      Sd, Dm
```

```
VCVTR .si32<f32    Sd, Sm
VCVTR .ui32<f32    Sd, Sm
VCVTR .si32<f64    Sd, Dm
VCVTR .ui32<f64    Sd, Dm
VCVTB .f32<f16     Sd, Sm
VCVTB .f16<f32     Sd, Sm
VCVTT .f32<f16     Sd, Sm
VCVTT .f16<f32     Sd, Sm
```

```
VDIV .f64          Dd, Dn, Dm
VDIV .f32          Sd, Sn, Sm
VFMA .f64          Dd, Dn, Dm
VFMA .f32          Sd, Sn, Sm
VFMS .f64          Dd, Dn, Dm
VFMS .f32          Sd, Sn, Sm
VFNMA .f64         Dd, Dn, Dm
VFNMA .f32         Sd, Sn, Sm
VFNMS .f64         Dd, Dn, Dm
VFNMS .f32         Sd, Sn, Sm
```

```
VLDMIA              Rn, { Dx-Dy }
VLDMIA              Rn ! { Dx-Dy }
VLDMDB              Rn ! { Sx-Sy }
VLDR                Dd, [ Rn, # imm ]
VLDR                Dd, label
VLDR                Sd, [ Rn, # imm ]
VLDR                Sd, label
```

```
VMLA .f64           Dd, Dn, Dm
VMLA .f32           Sd, Sn, Sm
VMLS .f64           Dd, Dn, Dm
VMLS .f32           Sd, Sn, Sm
VMOV .f64           Dd, # imm
VMOV .f32           Sd, # imm
VMOV .f64           Dd, Dm
VMOV .f32           Sd, Sm
VMOV .32            Dd [0/1] Rt
VMOV .32            Rt, Dn [0/1]
VMOV                Sn, Rt
VMOV                Rt, Sn
VMOV                Sm, Sm1, Rt, Rt2        \ Sm1=Sm+1
VMOV                Rt, Rt2, Sm, Sm1        \ Sm1=Sm+1
VMOV                Dm, Rt, Rt2
VMOV                Rt, Rt2, Dm
VMRS                Rt, FPSCR
VMSR                Rt, FPSCR
VMUL .f64           Dd, Dn, Dm
VMUL .f32           Sd, Sn, Sm
```

```
VNEG .f64           Dd, Dm
VNEG .f32           Sd, Sm
VNMLA .f64          Dd, Dn, Dm
VNMLA .f32          Sd, Sn, Sm
VNMLS .f64          Dd, Dn, Dm
VNMLS .f32          Sd, Sn, Sm
VNMUL .f64          Dd, Dn, Dm
VNMUL .f32          Sd, Sn, Sm
```

```
VPOP                { Dx-Dy }
VPOP                { Sx-Sy }
VPUSH               { Dx-Dy }
VPUSH               { Sx-Sy }
```

```
VSQRT .f64          Dd, Dm
VSQRT .f32          Sd, Sm
VSTMIA              Rn, { Dx-Dy }
VSTMIA              Rn ! { Dx-Dy }
VSTMDB              Rn ! { Sx-Sy }
VSTMIA              Rn, { Dx-Dy }
VSTR                Dd, [ Rn, # imm ]
VSTR                Dd, label
VSTR                Sd, [ Rn, # imm ]
VSTR                Sd, label
VSUB .f64           Dd, Dn, Dm
VSUB .f32           Sd, Sn, Sm
```

5.11 ARM instruction set

The ARM instruction set is mostly highly orthogonal. All data processing instructions work on the contents of registers and immediate constants only. Any data held in memory has to be loaded into a register, manipulated, then saved back to memory using one of the memory transfer instructions. This may appear to be restrictive, but due to the large number of general-purpose registers available for scratch storage, memory read/writes can be kept to a minimum. The assembler is of the prefix variety, with the instruction mnemonic preceding its parameters. Valid instructions are:

```
B | BL <<cond>> expression

BLX expression
BLX Rm

MOV | MVN <<cond>> <<S>> Rd op2
CMN | CMP | TEQ | TST <<cond>> <<P>> Rn op2
ADC | ADD | AND | BIC | EOR | ORR | RSB | RSC | SBC | SUB <<cond>>
<<S>> Rd Rn op2

MRS <<cond>> Rd psr
MSR <<cond>> psr Rm
MSR <<cond>> psrf Rm
MSR <<cond>> psrf #expression

MUL <<cond>> <<S>> Rd Rm Rn
MLA <<cond>> <<S>> Rd Rm Rs Rn

UMULL | SMULL | UMLAL | SMLAL <<cond>> <<S>> RdLo RdHi Rm Rs

LDR | LDRB | LDRH | STR | STRB | STRH <<cond>> Rd address <<!>>

LDMFD | LDMED | LDMFA | LDMEA | LDMIA | LDMIB | LDMDA | LDMDB |
STMFD | STMED | STMFA | STMEA | STMIA | STMIB | STMDA | STMDB
<<cond>> Rn <<!>> Rlist <<^>>

SWP | SWPB <<cond>> Rd Rm [ Rn ]

SWI <<cond>> expression

CDP <<cond>> CP# operation CRd CRn CRm info

LDC | LDCL | STC | STCL <<cond>> CP# CRd address

MCR | MRC <<cond>> CP# operation Rd CRn CRm info
```

Two pseudo instructions `MVL` and `ADR` are also available. `NOP` is supported as a synonym for:

```
MOV  R0, R0
```

No switches are accepted for `NOP`.

Parameter	Explanation
<<cond>>	Optional conditional execution code, i.e. .NE. See Control Structures.
<<S>>	Optional suffix .S to set the processor status flags.
<<P>>	Optional suffix .P to modify the PSR in 26-bit modes.
<<!>>	Optional `!` enables write-back of the base register in loads and stores.
<<^>>	Optional `^` sets the status flags when loading the PC from memory with the `LDMxx` instructions. Can also be used to force loading and storing of user bank registers in non-user modes.
Rd, RdLo, RdHi, Rm, Rn, Rs	Equates to a valid register number. See Register naming. Some instructions, notably the multiplication set, place restrictions on the combinations of registers allowed.
op2	Is either one of the operands produced by the ARM's barrel shifter or an immediate constant. See Shift operations and Immediate constants.
expression	For the `B` and `BL` instructions this is a label name or an expression evaluating to a branch address. The address is converted to make it pc relative, allowing for the effects of pipelining on the program counter. For the `SWI` instruction it is the number of the SWI to be called.
#expression	Evaluates to a 32-bit value. This has to be a valid Immediate constant.
address	A valid address specification. See Addressing modes.
Rlist	A list of registers enclosed by braces. See Register lists.
psr	Is the CPSR or SPSR register names.
psrf	Is the CPSR_flag or SPSR_flag register names. Only the N, Z, C and V flags are written into the status register. Use the forms _C _X _S _F to indicate which sections are to be restored.
CP#	The unique number of the required coprocessor.
CRd, CRn, CRm	Equates to a valid coprocessor register number. See "Register Naming" below.
operation	Is evaluated to a constant.
info	Is evaluated to a constant.

Table 5.5: ARM instruction notation

5.12 Register naming

There are fifteen general-purpose registers available in user mode, plus the program counter. These are named `R0` through `R15`. `R15` is the program counter. Coprocessor registers are named `CR0` through `CR15`. The Current Program Status Register and Saved Program Status Register are named `CPSR` and `SPSR` respectively. If transferring just the status flags then `CPSR_flg` and `SPSR_flg` can be used. As of v6.2 the notation

```
CPSR_flag
```

is superceded by

```
CPSR _c _x _s _f
```

where the valid field definers are:

```
_C _X _S _F _CXSF _FSXC _ALL
```

Standard ARM names are also available. `SP` refers to `R13` (commonly used as a stack pointer), `LINK` refers to `R14` (the link register), and `PC` refers to `R15` (the program counter).

Forth register names can be used in place of the standard register names. These are `TOS`, `LP`, `UP`, `RSP` and `PSP`. As mentioned earlier these can be assigned to different ARM registers.

All register names can be used with or without a trailing comma. This makes for code that is more readable to the seasoned ARM programmer. Character case is not important.

5.13 Immediate constants

Rather then specify the name of a register whose contents are to be used in an operation, it is possible with many instructions to specify a numeric value which is encoded with the opcode mnemonic at assembly time.

When the `#` is encountered, the assembler recognises that the following input is to be interpreted as a numeric value. The value itself can be prefixed with the usual number base selectors such as `#` for decimal, `$` for hexadecimal, `%` for binary and `@` for octal:

```
ADD R2, R3, # $32   \ Add $32 to contents of R3
                    \ and place result in R2
```

Note that in the UK, there may be confusion with some printers between the hash symbol `#` and the pound symbol.

There are restrictions regarding the range of immediate constants that can be used. As mentioned before each instruction and its operands are encoded as a single 32-bit value on the ARM. Obviously some of the 32 bits are going to be given over to the instruction type, suffixes, and destination register etc. leaving only 12-bits to represent the constant. 12 bits does not allow many immediate constants to be used, so this is split into two fields. One, 8-bits wide, specifies the constant while the other 4-bit wide field specifies a value to shift the constant by (this is actually a rotate right by the shift value times two places). This widens the range of immediate constants that can be used, but has the restriction that not every number in the full 32-bit range can be used. Note that the range of negative immediate constants that can be represented is very limited as these appear to the ARM to be very large numbers i.e. -1 = $FFFFFFFF, and the larger a number is the harder it is to represent using the method described above. Judicial use of such instructions as `CMN` (compare negative), `MVN` (move inverted data - not negated!) and `RSB` (reverse subtract) can get around this problem.

Cortex only: If the literal pool is enabled you can load a register with a 32-bit immediate value using the form:

```
LDR   Rx, @= imm32
```

e.g.

```
LDR   R4, @= $12345678
```

You must flush the literal pool yourself using *\fo{FLUSHLITPOOL (–) after `END-CODE` or `ENDPROC`.

5.14 Shift operations

Most data processing instructions allow operand two (the second source operand) to be specified as a shifted register. Here the contents of the register can be shifted at run-time by either a fixed amount or by the contents of another register. This can be done with one of the ARM's shift instructions, e.g.

```
ADD R0, R2, R7 LSL # 4   \ R7 logically shifted left by
                         \ 4 places
BIC R2, R4, R7 ASR R6    \ R7 arithmetically shifted
                         \ right by the contents of R6
```

Note that the contents of the register being shifted are not changed by the shift. The shifted value is only used during the instruction to calculate the new value to be stored in the destination register.

Note also that a shift by zero bits causes no change in the carry flag!

Shift operations supported by the ARM are:

Instruction	**Purpose**
LSL # n or LSL Rn	Logical shift left
ASL # n or ASL Rn	Arithmetic shift left (identical to LSL)
LSR # n or LSR Rn	Logical shift right
ASR # n or ASR Rn	Arithmetic shift right
ROR # n or ROR Rn	Rotate right
RRX	Rotate right with extend - (no shift value or register is needed as the shift is by one place only)

Table 5.6: ARM shift instructions

Note that as with immediate constants, if the shift is by a fixed amount it should be preceded by the `#` symbol to inform the assembler that it is not dealing with a register.

5.15 Addressing modes

The ARM data processing instructions all work on the contents of registers and immediate operands. To transfer data to and from single registers and memory either the `LDR`, `STR`, `LDC` or `STC` instructions and their variants have to be used. Addresses can be specified in three ways.

5.15.1 Pre-indexed addressing

Pre-indexed addressing allows an offset to be added to (or subtracted from) an address held in

a base register to form the address from which data is to be transferred. The address has the following format:

```
[ Rn <<offset>> ]
```

Where `Rn` is the base register name and the optional offset is either:

- A simple register
- An immediate constant
- A shifted register

The address expression must be terminated by a `]`. The initial `[` is not strictly necessary but leads to code that is more readable for experienced ARM programmers.

A simple or shifted register offset needs to be prefixed with `++` or `--` indicating whether the contents of the register should be added to or subtracted from the base register. Immediate constants do not use the 8/4-bit field format but rather range from -4095 to 4095. Shifted registers can only be shifted by a constant preceded by the `#` symbol and not by the contents of another register.

The address calculated by combining the base and offset registers is often useful in subsequent loads and stores, especially when a sequence of memory locations are to be accessed. Use the `!` operator after the closing `]` to enable the write back feature of the ARM. This will write the calculated address back into the base register for subsequent instructions to use.

Instruction	**Address**
LDR Rd, [Rn]	Load from Rn. Treated as LDR Rd, [Rn, # 0]
LDR Rd, [Rn, ++ Rm]	Load from Rn plus Rm
LDR Rd, [Rn, – Rm] !	Load from Rn minus Rm with write back
LDR Rd, [Rn, ++ Rm LSL # 5] !	Load from Rn plus Rm shifted logically left five places with write back
LDR Rd, [Rn, # 20]	Load from Rn plus twenty
LDR Rd, [Rn, # -40] !	Load from Rn minus forty with write back

Table 5.7: Pre-indexed addressing

5.15.2 Post-indexed addressing

Post-indexed addresses have the following form:

```
[ Rn ], <<offset>>
```

Post-indexed addressing adds the offset to the base register `Rn` after the data has been transferred from the address held in the base register. This implies that write back always occurs so it is not necessary to specify it. It can be used however to force non-privileged mode for the transfer cycle (same as the T suffix on some ARM assemblers).

The offset is specified in exactly the same way as for pre-indexed addressing. Examples of post-indexed addressing are:

Instruction	Address
LDR Rd, [Rn], ++ Rm	Load from Rn then add Rm to Rn
LDR Rd, [Rn], – Rm	Load from Rn then subtract Rm from Rn
LDR Rd, [Rn], ++ Rm LSL # 5	Load from Rn then add Rm, shifted logically left five places, to Rn
LDR Rd, [Rn], – Rm LSL # 5	Load from Rn then subtract Rm, shifted logically left five places, from Rn
LDR Rd, [Rn], # 20	Load from Rn then add 20 to Rn
LDR Rd, [Rn], # -40	Load from Rn then subtract 40 from Rn

Table 5.8: Post-indexed addressing

5.15.3 PC relative addressing

The assembler also recognises addresses specified as either an absolute number or an assembler label, e.g.

```
LDR R2, # $600          \ Load from memory location $600
LDR R2, label           \ Load from the address marked by label
```

Addresses specified using PC relative addressing are actually converted into pre-indexed addresses that load from the program counter (R15) plus or minus an immediate constant. This means that the address of the desired memory location has to lie within +/-4096 bytes of the address of the instruction referencing it. The assembler will take into account the effects of pipelining on the program counter when calculating the value of the offset.

5.15.4 Byte and half word addressing

The instructions `LDRB` and `STRB` plus `LDRH` and `STRH` can be used to transfer bytes or half words between memory and registers. Byte loads and stores only utilise the bottom 8-bits of the destination register and half words only the bottom 16-bits. The contents of the rest of the register are ignored on a store, and zeroed on a load from memory. Unlike word memory transfers, byte loads and stores do not have to be aligned, but half word transfers should be aligned to a two-byte boundary.

5.16 Register lists

Multiple registers can be loaded from and stored to memory using the `LDM` and `STM` instructions. The format is:

```
LDMxx Rd, <<!>> { Ra, Rb, Rx-Ry, ... } <<^>>
STMxx Rd, <<!>> { Ra, PC, LINK, Re-Rf } <<^>>
```

```
{ R0 R1 R2 R6 R12 }
{ R0-R2, R6, R12 }
{ R6, R12, R0-R2 }
```

Each register can only be specified once.

The optional final ^ sets the status flags when loading the PC from memory with the LDMxx instruction. It can also be used to force loading and storing of user bank registers in non-user modes. MVL and ADR

As indicated earlier, a common source of problems when programming with ARM assembler is the restriction placed on the range of immediate constants that can be used with the data processing instructions. To get around this the pseudo instruction MVL can be used to move any signed/unsigned 32-bit number into a register.

```
MVL R2, # 127653
```

The MVL pseudo instruction will attempt to use a single MOV or MVN instruction if possible, but may generate up to four ARM instructions or two Cortex instructions to get the value into the register.

For Cortex, you can also use

```
MVL32 R2, # $12345678
```

to load a 32 bit value into a register. This is useful when you are referencing the data address of a VALUE or a Forth word, e.g.

```
$12345678 value foo
Proc MyISR
  ...
  mvl32  r0, # ' foo >body  \ load data address
```

For Cortex, branch destination addresses that are loaded into the PC must have bit0 set to 1. To this, use:

```
MVL32+1 R2, # <value>
```

The value can be forward referenced.

ADR is a pseudo instruction (macro) for ARM, but is a real instruction for Thumb-2. The ARM ADR pseudo instruction performs is used to move a 32-bit address into a register.

```
ADR label
```

Due to the possibility that a label might be forward referenced and need 'fixing up' later on in the compilation, the ADR pseudo instruction will always generate a MOV and three ORR instructions.

5.17 Switching between Forth and Assembler

The compiler allows you to add in-line assembler inside colon definitions, and to add high level phrases inside code definitions.

Inline assembler code can be compiled inside a colon definition using [ASM and ASM]. Use these in the form:

```
: <name>
  ... [ASM <assembler-code> ASM] ...
;
```

High level code can be compiled inside a `CODE` definition using `[FORTH` and `FORTH]`. Use these in the form:

```
CODE <name>
  ... [FORTH <high-level> FORTH] ...
END-CODE
```

Note that the optimiser is not flushed by the switches into assembler. This can (and should) be achieved by placing `[O/F]` before `[ASM` and `FORTH]`.

5.18 Glossary

This glossary details the lwords provided within the cross-assembler to control the use of the assembler.

• Bit31 always 0

`: ARM32 \ --`
Select ARM7 mode for assembler

`: ArmArch5 \ --`
Select ARMv5 mode for assembler

`: Thumb-1 \ --`
Select full Thumb-1 mode for assembler and code generator.

`: Thumb-2 \ --`
Select full Thumb-2 mode for assembler and code generator.

`: Cortex-M0 \ --`
Select Cortex-M0 for assembler and code generator.

`: Cortex-M1 \ --`
Select Cortex-M1 for assembler and code generator.

`: Cortex-M3 \ --`
Select Cortex-M3 for assembler and code generator.

`: Cortex-M4 \ --`
Select Cortex-M4 (includes integer DSP) for the assembler and code generator.

`: Cortex-M4F \ --`
Select Cortex-M4F (includes integer DSP and single-precision VFP) for the assembler and code generator.

`: Cortex-M7 \ --`
Select Cortex-M7 (includes integer DSP, single and double precision VFP) for the assembler and code generator.

`: ARM32? \ -- flag`
Return true if in 32 bit ARM mode.

`: ArmArch5?  \ -- flag`
Return true if in 32 bit ARMv5 mode.

`: Thumb1?  \ -- flag`
Return true if in Thumb-1 mode.

`: Thumb2?  \ -- flag`
Return true if in Thumb-2 mode.

`: Thumb?  \ -- flag`
Return true if in either Thumb mode.

`: Cortex-M0?  \ -- flag`
Return true if the Cortex-M0 instruction set has been selected.

`: Cortex-M1?  \ -- flag`
Return true if the Cortex-M1 instruction set has been selected.

`: Cortex-M0/M1? \ -- flag`
Return true if the Cortex M0 or M1 instruction sets have been selected.

`: Cortex-M3?  \ -- flag`
Return true if the Cortex-M3 instruction set has been selected.

`: Cortex-M4?  \ -- flag`
Return true if the Cortex-M4 instruction set has been selected.

`: Cortex-M4F?  \ -- flag`
Return true if the Cortex-M4 instruction set has been selected.

`: Cortex-M7?  \ -- flag`
Return true if the Cortex-M7 instruction set has been selected.

`: IDSP?  \ -- flag`
Return true if the Cortex integer DSP instructions are present.

`: Not-M0/M1?  \ -- flag`
Return true if the Cortex M0 or M1 instruction sets have not been selected.

`: M0/M1?  \ -- flag`
Return true if Thumb-2 and Cortex M0 or M1 has been selected.

`: .f32  \ --`
Indicates that the data in the Sn/Dn/Qm registers is 32 bit.

`: .f64  \ --`
Indicates that the data in the Dn/Qm registers is 64 bit.

`: .s8  \ --`
Indicates that the data in the Sn/Dn/Qm registers is signed 8 bit.

`: .s16  \ --`
Indicates that the data in the Sn/Dn/Qm registers is signed 16 bit.

`: .s32  \ --`
Indicates that the data in the Sn/Dn/Qm registers is signed 32 bit.

`: .s64  \ --`
Indicates that the data in the Dn/Qm registers is signed 64 bit.

`: .u8  \ --`

Indicates that the data in the Sn/Dn/Qm registers is unsigned 8 bit.

```
: .u16          \ --
```
Indicates that the data in the Sn/Dn/Qm registers is unsigned 16 bit.

```
: .u32          \ --
```
Indicates that the data in the Sn/Dn/Qm registers is unsigned 32 bit.

```
: .u64          \ --
```
Indicates that the data in the Dn/Qm registers is unsigned 64 bit.

```
: .i8           \ --
```
Indicates that the data in the Sn/Dn/Qm registers is signed 8 bit.

```
: .i16          \ --
```
Indicates that the data in the Sn/Dn/Qm registers is signed 16 bit.

```
: .i32          \ --
```
Indicates that the data in the Sn/Dn/Qm registers is signed 32 bit.

```
: .i64          \ --
```
Indicates that the data in the Dn/Qm registers is signed 64 bit.

```
: .8            \ --
```
Indicates that the data in the Sn/Dn/Qm registers is 8 bit.

```
: .16           \ --
```
Indicates that the data in the Sn/Dn/Qm registers is 16 bit.

```
: .32           \ --
```
Indicates that the data in the Sn/Dn/Qm registers is 32 bit.

```
: .64           \ --
```
Indicates that the data in the Dn/Qm registers is 64 bit.

```
: [n]           ( u -- )  <VFPindex> !  ;
```
Set an index for VMOV.

```
: [0]           ( -- )  0 [n]  ;
```
Set index 0 for VMOV.

```
: [1]           ( -- )  1 [n]  ;
```
Set index 1 for VMOV.

```
: [2]           ( -- )  2 [n]  ;
```
Set index 2 for VMOV.

```
: [3]           ( -- )  3 [n]  ;
```
Set index 3 for VMOV.

```
: [4]           ( -- )  4 [n]  ;
```
Set index 4 for VMOV.

```
: [5]           ( -- )  5 [n]  ;
```
Set index 5 for VMOV.

```
: [6]           ( -- )  6 [n]  ;
```
Set index 6 for VMOV.

```
: [7]           ( -- )  7 [n]  ;
```
Set index 7 for VMOV.

`: dxb \ b -- ; lay byte`
Lay an 8-bit byte into the instruction stream. No alignment is performed. Use in the form:

```
dxb $55
```

`: dxw \ w -- ; lay 16 bits`
Lay a 16-bit word into the instruction stream. No alignment is performed. Use in the form:

```
dxw $55AA
```

`: dxl \ l -- ; lay 32 bit long`
Lay a 32-bit dword into the instruction stream. No alignment is performed. Use in the form:

```
dxl $11223344
```

`: db \ b -- ; lay byte`
Lay a single byte inline. **Obsolete**, will be removed in a future release, use `DXB` instead.

`: dw \ b -- ; lay 16 bits`
Lay a 16 bit item inline. No alignment is performed. **Obsolete**, will be removed in a future release, use `DXW` instead.

`: dd \ l -- ; lay 32 bit long`
Lay a 32 bit item inline. No alignment is performed. **Obsolete**, will be removed in a future release, use `DXL` instead.

`: dl \ l -- ; lay 32 bit long`
Lay a 32 bit item inline. No alignment is performed. **Obsolete**, will be removed in a future release, use `DXL` instead.

`: align4 \ --`
Forces the PC to a four byte boundary.

`: $ \ -- chere`
Returns the current value of the PC.

`: ;CODE \ --`
A defining word used in the form:

```
: <namex>  CREATE ... ;CODE ... END-CODE
```

Stops compilation, and enables the assembler. This word is used with `CREATE` to produce defining words whose run-time portion is written in code, in the same way that `CREATE ... DOES>` is used to create high level defining words .

The data structure is defined between `CREATE` and `;CODE` and the run-time action is defined between `;CODE` and `END-CODE`. The current value of the data stack pointer is saved by `;CODE` for later use by `END-CODE` for error checking. When `<namex>` executes the address of the data area will be found on the processor stack, from which it must be removed.

`: ASMCODE \ --`
Starts a section of assembler code and turns on the assembler, but without generating a dictionary header. This action is particularly useful for generating the start-up code. Examples of this can be found in *CODEARM.FTH*.

```
ASMCODE ... END-CODE
```

`: CODE \ --`
A defining word used in the form:

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

Creates a dictionary entry for <name> to be defined by a following sequence of assembly language words. Words thus defined are called code definitions. CODE stores the current data stack pointer for later error checking by END-CODE.

: END-CODE \ --
Terminates a code definition and checks the data stack pointer against the value stored when ;CODE or CODE is executed. The assembler is disabled. See: CODE ;CODE.

: IS-ACTION-OF \ addr --
Used to tell the cross-compiler that the given address is to be used as the run time action of the word whose name follows. Usually found in code definitions, but can also be used for high-level definitions. For example:

```
ASMCODE
HERE IS-ACTION-OF CONSTANT
  ...
END-CODE
ASSEMBLER
HERE IS-ACTION-OF <<high-level-definer>>
B DODOES
END-CODE
] ... EXIT [
```

: PROC \ -- ; PROC <label-name>
Starts a section of assembler code and turns on the assembler, defining a label. This action is particularly useful for generating interrupts or shared subroutines. Examples of this can be found in *Cortex/CodeCortex.fth*.

: !call \ dest ^ins --
Patch a branch opcode at ^*ins* to branch to target address *dest*. The opcode portion is not changed, so that this word works with both B and BL. Note that this word may be redefined in some target code for ARM7 and ARM9 devices.

6 Cortex interrupts

This chapter describes how to write interrupt handlers in both Forth and assembler. It details how to set up and control interrupt handlers. The first part of the chapter describes how to set up and use the interrupts, and the second part provides more technical detail.

Unlike ARM7 and ARM9 CPUs, Cortex devices all use the same interrupt controller, the Nested Vectored Interrupt Controller, or NVIC for short. The target code file *Cortex/IntCortex.fth* contains the code.

Cortex devices differentiate between interrupts and exceptions. Exceptions may be fault handlers, or system interrupts such as the system ticker or the SVC call to a service. Fault handlers provided by MPE use a slightly different entry sequence for fault handlers. See the file *Cortex/FaultCortex.fth* for the details. Conditional compilation handles the differences between Cortex-M variants.

6.1 Interrupts on the Cortex-M3+

When an interrupt occurs and is accepted, a Cortex processor branches through a location (a 'vector'), which is usually in low memory. Each exception or interrupt has its own location. The vector table starts at address 0, the vector used being dependent on the source of the interrupt. The code at the address stored in the vector table is then executed. The first 16 vectors are for reset and system exceptions, the rest are defined by the chip designer.

The Forth code refers to all exceptions by their index in the exception vector table. Thus exception 1 is the reset entry and exception 16 is the first chip-specific interrupt. This is unlike the ARM CMSIS habit, which uses negative numbers for the system exceptions and positive for chip-specific interrupts. Not all of the 16 system vectors are defined, and some manufacturers use one or more of the undefined locations for chip-specific data. Index zero is special in that it contains the value of the R13 stack pointer used at start up (SP_main in ARM documentation). The MPE code uses SP_main for interrupt and exception handlers, and the main application code uses SP_process. Normally Forth code runs in "privileged Thread mode" in ARM parlance.

For more information on Cortex interrupts, refer to the device's user guide and the Cortex-M3 Technical Reference Manual from *www.arm.com*. In this chapter we are mostly interested in the interrupts, although the principles apply equally well to the other exceptions.

6.2 Interrupts on the Cortex-M0/M1

The files *Cortex\IntCortex.fth* and *Cortex\FaultCortex.fth* contain conditional compilation to handle the differences between Cortex-M0/M1 and Cortex-M3 upwards.

6.3 Writing Forth interrupt handlers

A Forth interrupt service routine (ISR) is just like any other Forth word. It can therefore be tested and debugged like a normal Forth word. Only when the word is fully tested need it be assigned to an interrupt.

6.3.1 Setting an interrupt

```
' A/D-READY 23 Exc: adc-isr
```

The name `adc-isr` is a label you choose. It returns the address of the exceptions's entry point.

6.4 Some common problems

There are a few common problems that might cause an interrupt not to work correctly:

- a stack fault
- the source is not cleared
- the interrupts are not enabled

6.4.1 Stack faults

An interrupt service routine can use the stack while it is executing, but must clear up the stack before returning from the interrupt. The normal symptom of a stack fault is that the interrupt handler runs but then the target board crashes, either immediately or after a length of time.

6.4.2 Source is not cleared

Once an interrupt handler is triggered by an interrupt, the source of the interrupt must be told that the interrupt is being serviced. If this is not done, the source of the interrupt will carry on generating interrupts. Normally this appears as the interrupt handler executing once and then the target board 'locking'. Interrupts are not enabled

Interrupts need to be enabled with `EI` before any interrupts will be serviced. The vectors must be set up or the interrupt handler assigned to the deferred word before the interrupts are enabled. Fault handlers must be enabled with `EFI`.

6.5 Writing assembler interrupt handlers

Just write your handler as a `CODE ... END-CODE` definition and add it to the vector table in the usual way. If your handler only needs to use registers R0..R3 and R12, these are saved and restored by the core itself. In this case you will not need the MPE wrapper for a high level interrupt handler. The following template comes from a SysTick handler.

```
0 value ticks

Proc SysTickISR \ --
  mvl32   r0, # ' ticks >body
  ldr     r1, [ r0, # 0 ]
  add .s  r1, r1, # tick-ms
  str     r1, [ r0, # 0 ]
  bx      lr
end-code
SysTickISR SysTickvec# setExcVec
If you want to replace one of the MPE interrupt handlers, please
use the code in Cortex/IntCortex.fth as an example if you
have not written Cortex interrupt handlers before.
```

6.6 Controlling the interrupts

Interrupts can be in one of two states, enabled or disabled.

6.6.1 Enabling interrupts

To enable interrupts use `EI`. Once `EI` has been executed, all interrupts are enabled. MPE code uses `EI` to enable interrupts, and `EFI` to enable the fault exceptions.

6.6.2 Disabling interrupts

To disable interrupts use *DI*. Once *DI* has been used, all interrupts are disabled. MPE code uses `DI` to disable interrupts, and `DFI` to disable fault exceptions.

6.7 A simple example

The following example patches a high-level interrupt service routine (ISR) onto a timer interrupt.

6.7.1 The timer ISR

The example ISR increments a variable that can be fetched in the foreground to detect that the timer is working. The ISR also needs to acknowledge that the interrupt has happened.

```
VARIABLE TICKS
: TIMER-SERVICE   \ --
  _TCSR C@ DROP            \ must *read* the register
  _TCSR_VALUE _TCSR W!     \ clear interrupt
  1 TICKS +!               \ increment counter
;
```

6.7.2 Initialising the timer

Before the timer ISR will run it requires initialising:

```
_TCSR_/32                \ select required clock rate
_TCSR_TME OR             \ enable timer and
_TCSR_PASSWORD OR        \ use "password"
EQU _TCSR_VALUE          \ to build the *word* that
                         \ we will send to tscr.
: TIMER-INIT   \ --
  _TCSR_VALUE _TCSR W!   \ enable timer, select rate
  EI                     \ enable interrupts
;
```

6.7.3 Setting the vector

The ISR needs to be patched into the vector table.

```
' TIMER-SERVICE <vec#> Exc: TOV-ISR
```

6.7.4 Testing the timer is running

The timer needs to be initialised with `TIMER-INIT` and then the timer can be tested by checking the variable `TICKS`.

```
TIMER-INIT
TICKS ?
TICKS ?
```

This displays the current value of `TICKS`.

6.8 Interrupt handlers in detail

When an interrupt occurs and is accepted, an ARM CPU branches through a location (a 'vector') in low memory. A conventional interrupt handler written in assembler simply saves any registers and fixed locations it will use, then calls or executes the required routine, and then restores the previous state of registers and locations, and returns. The Forth interrupt handlers work in the same way, with a few additions.

Interrupt handlers written entirely in assembler are written in the usual way. When a Forth `VARIABLE` is referred to by name inside a code section, its data address in RAM is returned. Thus variables can be shared between high level routines and subroutines or assembler interrupt handlers. The code for the interrupt handler can be found in the file *Cortex/IntCortex.fth*. This source code can be modified and updated as required. The interrupt handler code can be adapted for single-chip use.

6.8.1 Interrupt protection

The word `[I` saves the current interrupt status on the return stack, and disables interrupts, which can then be restored by `I]` which restores the interrupt mask to the state saved by `[I`. If simple enabling and disabling of interrupts is required, the words `EI` and `DI` respectively perform these functions. `[I` and `I]` are particularly necessary for critical sections of code, and for implementing semaphores.

In the Cortex compiler these words are implemented by code generators, and so it is not necessary

to supply target definitions for them. If they are required interactively, example definitions can be found in *Cortex/LibCortex.fth*, and will be included automatically if required, providing that you use the relevant `LIBRARIES ... END-LIBS` sequence.

6.8.2 End of interrupt requirements

Most interrupts on Cortex cores require that the programmer explicitly clear the interrupt source before returning from the interrupt. This action must be performed by the high-level interrupt routine. In many cases, resetting the appropriate bit in a status register clears the interrupt source.

6.9 Glossary

This glossary contains details of the major words in the interrupt system. Other words exist, but are only used as fractions of the words below. The source code for all these words may be found in *Cortex/CodeCortex.fth*.

`code DI` `\ --`
Disables interrupts.

`code EI` `\ --`
Enables interrupts.

`code DFI` `\ --`
Disables fault exceptions. Not available for Cortex-M0/M1.

`code EFI` `\ --`
Enables fault exceptions. Not available for Cortex-M0/M1.

`code [I` `\ R: -- x1 x2`
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.

`code I]` `\ R: x1 x2 --`
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.

7 ARM interrupts

This chapter describes how to write interrupt handlers in both Forth and assembler. It details how to set up and control interrupt handlers. The first part of the chapter describes how to set up and use the interrupts, and the second part provides more technical detail.

Different ARM implementations use different interrupt controllers, although many are derived from, or similar to the ARM PL190 controller. The code for the ones we have used is collated into the file *ARM\IntARM3.fth*. The syntax setting for interrupt handlers varies slightly between the controllers.

7.1 Interrupts on the ARM

When an interrupt occurs and is accepted, an ARM processor branches through a location (a 'vector') in low memory. The vector table starts at address 0, the vector used being dependent on the source of the interrupt. The code at the address stored in the vector table is then executed. For more information on interrupts for the ARM, refer to the processor's user guide. In this chapter we are mostly interested in the IRQ and FIQ interrupts, although the principles apply equally well to the other interrupts.

7.2 Writing Forth interrupt handlers

A Forth interrupt service routine (ISR) is just like any other Forth word. It can therefore be tested and debugged like a normal Forth word. Only when the word is fully tested need it be assigned to an interrupt.

The interrupt handler allows you to chain several interrupt handlers onto the primary IRQ and FIQ handlers. This makes complex code much easier to manage. The action of of each handler should be:

- Exit if it isn't for me.
- Reset the interrupt source
- Process the interrupt

7.2.1 Setting an interrupt

```
' A/D-READY ADD-IRQ
```

which adds the word `A/D-READY` to the IRQ chain.

7.3 Some common problems

There are a few common problems that might cause an interrupt not to work correctly:

- a stack fault
- the source is not cleared
- the interrupts are not enabled

7.3.1 Stack faults

An interrupt service routine can use the stack while it is executing, but must clear up the stack before returning from the interrupt. The normal symptom of a stack fault is that the interrupt handler runs but then the target board crashes, either immediately or after a length of time.

7.3.2 Source is not cleared

Once an interrupt handler is triggered by an interrupt, the source of the interrupt must be told that the interrupt is being serviced. If this is not done, the source of the interrupt will carry on generating interrupts. Normally this appears as the interrupt handler executing once and then the target board 'locking'. Interrupts are not enabled

Interrupts need to be enabled with `EI` before any interrupts will be serviced. The vectors must be set up or the interrupt handler assigned to the deferred word before the interrupts are enabled.

7.4 Writing assembler interrupt handlers

If you use the standard IRQ and FIQ chains, just write your handler as a `CODE ... END-CODE` definition and add it to the chain in the usual way. If you want to replace one of the MPE interrupt handlers, please use the code in *ARM\INTARM3.FTH* as an example if you have not written ARM interrupt handlers before.

7.5 Controlling the interrupts

Interrupts can be in one of two states, enabled or disabled.

7.5.1 Enabling interrupts

7.5.2 Disabling interrupts

To disable interrupts use DI. Once DI has been executed, all interrupts are disabled. Some MPE code uses `DI` to disable the IRQ interrupt, and `DFI` to disable the FIQ line.

7.6 A simple example

The following example patches a high-level interrupt service routine (ISR) onto a timer interrupt.

7.6.1 The timer ISR

The example ISR increments a variable that can be fetched in the foreground to detect that the timer is working. The ISR also needs to acknowledge that the interrupt has happened.

```
VARIABLE TICKS
: TIMER-SERVICE \ --
  _TCSR C@ DROP          \ must *read* the register
  _TCSR_VALUE _TCSR W!   \ clear interrupt
  1 TICKS +!                     \ increment counter
;
```

7.6.2 Initialising the timer

Before the timer ISR will run it requires initialising:

```
_TCSR_/32                \ select required clock rate

_TCSR_TME OR             \ enable timer and
_TCSR_PASSWORD OR        \ use "password"
EQU _TCSR_VALUE          \ to build the *word* that
                         \ we will send to tscr.
: TIMER-INIT   \ --
  _TCSR_VALUE _TCSR W!   \ enable timer, select rate
  EI                     \ enable interrupts
;
```

7.6.3 Patching your ISR onto the timer

The ISR needs to be patched onto the timer interrupt. This interrupt has been setup so you need only to assign the timer code to the DEFERed word TOV-ISR.

```
ASSIGN TIMER-SERVICE TO-DO TOV-ISR
```

7.6.4 Testing the timer is running

The timer needs to be initialised with TIMER-INIT and then the timer can be tested by checking the variable TICKS.

```
TIMER-INIT
TICKS ?
TICKS ?
```

This displays the current value of TICKS.

7.7 Interrupt handlers in detail

When an interrupt occurs and is accepted, an ARM CPU branches through a location (a 'vector') in low memory. A conventional interrupt handler written in assembler simply saves any registers and fixed locations it will use, then calls or executes the required routine, and then restores the previous state of registers and locations, and returns. The Forth interrupt handlers work in the same way, with a few additions.

Interrupt handlers written entirely in assembler are written in the usual way. When a Forth VARIABLE is referred to by name inside a code section, its data address in RAM is returned. Thus variables can be shared between high level routines and subroutines or assembler interrupt handlers. The code for the interrupt handler can be found in the file ARM\INTARM3.FTH. This source code can be modified and updated as required. The interrupt handler code can be adapted for single-chip use. This file supports a number of different interrupt controllers. The functions supported are documented within the file itself and in the ARM specific target manual. The files *INTARM.FTH* and *INTARM2.FTH* are provided for compatibility with earlier releases, but

are no longer supported by MPE. We strongly encourage you to convert code using the earlier files to use *INTARM3.FTH*.

7.7.1 Interrupt structure

A separate user area is reserved for the interrupt handler, and is shared by all the high-level interrupt handlers. Consequently high-level Forth interrupts cannot be nested if user variables are changed, although an interrupt that does not use the Forth interrupt handler can be nested. The layout of the interrupt RAM page is identical to that for a task page (see later).

When an interrupt that has a high-level handler occurs, all registers are pushed onto the stack. These are then loaded with the required values for interrupt processing.

The interrupt code causes the processor to jump to an area of memory that contains calls to two words. The first is a call to a word to run the interrupt; this can be reassigned. The second is a call to the return from interrupt word.

7.7.2 Setting an interrupt

To set the action of an interrupt, simply assign that word to do the appropriate deferred word. The return from interrupt is provided by the second word of the table entry. This means that the action word need contain no return action, and consequently it may be tested from the keyboard. In the earlier example, the variable `TICKS` can be inspected to see that the timer is running. Changes to this structure can be made by rebuilding the Forth kernel with the cross compiler.

In particular, some interrupts may need to be serviced by an assembler routine, and the overhead of the Forth dispatcher is unwanted. In this instance edit the interrupt entry code to call a different routine.

Since ARM interrupts (exceptions) are vectored, the final requirement is to set the vector itself.

If you wish to add vectors for processing other exceptions, you should use the code provided in *INTARM3.FTH* as a prototype, and follow a similar structure.

The words `ASSIGN` and `TO-DO` can be used to assign a Forth word to be executed by the interrupt handler in the following manner.

```
ASSIGN action TO-DO xxxx-ISR
```

where `xxxx` is the name of the interrupt, as given in the table below, and `action` is the Forth word that the interrupt handler must execute. Thus to set up a complete interrupt system the vector must be assigned, and the word connected to the interrupt structure.

```
ASSIGN action TO-DO xxxx-ISR
<xxxx-INTO>  <vector-name> !
```

where `<vector-name>` is the vector name. Note that interrupt handlers written in assembler do not need to use these structures at all. They must still, however, set the CPU vectors in low memory.

Source	Vector name	Entry point	Deferred word
SWI	SWIV	SWI_exception	SWI_handler
Undef	UndefV	Undef_exception	Undef_handler
Prefetch abort	PabortV	PAbort_exception	PAbort_handler
Data abort	DabortV	DAbort_exception	DAbort_handler
Unused	UnusedV	Unused_exception	Unused_handler
IRQ	IRQV	IRQ_exception	Chained by ADD-IRQ
FIQ	FIQV	FIQ_exception	Chained by ADD-FIQ

Table 7.1: ARM interrupt vectors

7.7.3 Interrupt protection

The word `[I` saves the current interrupt status on the return stack, and disables interrupts, which can then be restored by `I]` which restores the interrupt mask to the state saved by `[I`. If simple enabling and disabling of interrupts is required, the words `EI` and `DI` respectively perform these functions. `[I` and `I]` are particularly necessary for critical sections of code, and for implementing semaphores. In general the use of `[I ... I]` requires less code than the use of `SAVE-INT ... RESTORE-INT` below, which are now deprecated.

The word `SAVE-INT` saves the current interrupt status, and disables interrupts, which can then be restored by `RESTORE-INT` which restores the interrupt mask to the state saved by `SAVE-INT`.

On the ARM compiler these words are implemented by code generators, and so it is not necessary to supply target definitions for them. If they are required interactively, example definitions can be found in *LIBARM.FTH*, and will be included automatically if required, providing that you use the relevant `LIBRARIES ... END-LIBS` sequence.

7.7.4 End of interrupt requirements

Certain interrupts on the ARM cores require that the programmer explicitly clear the interrupt source before returning from the interrupt. This action must be performed by the high-level interrupt routine. In many cases, resetting the appropriate bit in a status register clears the interrupt source. Glossary

This glossary contains details of the major words in the interrupt system. Other words exist, but are only used as fractions of the words below. The source code for all these words may be found in *ARM\IntARM3.fth*.

```
code DI         \ --
```
Disables interrupts.

```
code EI         \ --
```
Enables interrupts.

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

```
code I]         \ R: x --
```

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.

`code RESTORE-INT \ x --`
Restore the interrupt enable state previously saved by `SAVE-INT`. OBSOLETE, use `I]` instead.

`code SAVE-INT \ -- x`
Saves the current state of the interrupt enable on the stack, and disables interrupts. See `RESTORE-INT`. OBSOLETE, use `[I` instead.

8 Calling functions in other languages

8.1 Introduction

The ARM/Cortex Forth cross-compiler supports calling functions in C or any language that can provide functions that use the AAPCS calling convention. This is an ARM convention documented in *IHI0042F_aapcs.pdf*.

This chapter documents the Forth side of the Forth to C interface. A separate chapter discusses the C side of the interface.

In order to provide the greatest isolation between sections of code written in other languages, the functions foreign to Forth are accessed by SVC calls and/or jump tables. The simple solution just uses SVC calls for all foreign functions.

All foreign function calls are made using the AAPCS calling convention.

Calls with a variable number of parameters (**varargs**) are not supported.

The interface is defined for Cortex-Mx CPUs only. If you need an ARM32 interface, contact MPE.

This work is directly inspired by Robert Sexton's Sockpuppet interface:

`https://github.com/rbsexton/sockpuppet`

His contribution and permission are gratefully acknowledged.

8.1.1 SVC calls

Access to functions in a library is provided by the `SVC( n )` syntax which is similar to a C style function prototype, e.g.

```
SVC( n ) int open(
  const char * pathname, int flags, mode_t mode
);
```

where n is the SVC call number. Everything on the source line after the closing ')' is discarded. The example is for the function **open()** from a file API, and produces a Forth word `open`.

```
open  \ pathname flags mode -- int
```

The calling convention is always AAPCS. Because of the variations between C compilers and silicon vendor conventions, there are options for saving R9 and R12. In many cases of embedded systems compilers these registers do not need to be saved.

The parser used to separate the tokens is not ideal. If you have problems with a definition, make sure that `*` tokens are white-space separated. Formal parameter names, e.g. *argv* above are ignored. Array indicators, *[]* above, are also ignored when part of the names.

A small number of SVC calls are defined by the system. Regard entries 0..15 as being reserved by the Sockpuppet API.

```
SVC( 0 ) int GetSAPIversion( void );
\ Returns the Sockpuppet interface version.
SVC( 1 ) void * GetLinkList( void );
\ Returns the head of the linked list used to share data.
SVC( 7 ) uint32_t GetTimeMS( void );
\ Returns the value of the millisecond timer. On overflow,
this wraps around.
SVC( 14 ) void * GetDirFnTable( void );
\ Returns the address of the direct jump table, which may
\ be the same as the SVC jump table below.
SVC( 15 ) void * GetSVCFnTable( void );
\ Returns the address of the SVC jump table.
```

8.1.2 Jump tables

Calling through SVC calls is simple, but has overhead. In addition, lower priority interrupts are disabled by an SVC call and so interrupt latency suffers badly. You cannot do ticker-based timing inside an SVC-based call without messing about with the ARM/Cortex priority mechanisms. The solution is is to call the functions through a jump table as often used by an SVC call mechanism. This requires a little bit more code than the pure SVC approach, but has **much** better interrupt latency and gives more flexibility.

The example of **open()** is used again. Access to functions in a library is provided by the `JTI( n )` syntax which is similar to a C style function prototype, e.g.

```
JTI( n ) int open(
  const char * pathname, int flags, mode_t mode
);
```

where n is the index (0, 1, 2, ...) into the jump table. The parsing rules are as for SVC calls. The definition results in a Forth word `open`.

```
  open  \ pathname flags mode -- int
```

Before use, you have to tell the cross compiler where the address of the jump table is held, and then use an SVC call find the address of the jump table.

```
SVC( 15 ) void * GetDirFnTable( void );
\ Define SVC call that returns the address of the
\ jump table.

variable JT     \ -- addr
\ Holds the address of the jump table.
JT holdsJumpTable
\ Tell the cross compiler where the jump table address
\ is held.

: initJTI       \ -- ; initialise jump table calls
  GetDirFnTable JT !  ;

JTI( n ) int open(
  const char * pathname, int flags, mode_t mode
);
```

8.1.3 Double indirect call tables

Before use, you have to declare the base address of the primary ROM table used for calling ROM functions. For Luminary/TI CPUs, this will probably be:

```
$0100:0010 setPriTable
```

Now you can define a set of ROM calls, for example, again for a TI CPU.

```
DIC( 4, 0 ) void ROM_GPIOPinWrite(
   uint32 ui32Port, uint8 ui8Pins, uint8 ui8Val
);
```

where

- ROM_APITABLE is an array of pointers located at 0x0100.0010.
- ROM_GPIOTABLE is an array of pointers located at ROM_APITABLE[4].
- ROM_GPIOPinWrite is a function pointer located at ROM_GPIOTABLE[0].

Parameters:

- ui32Port is the base address of the GPIO port.
- ui8Pins is the bit-packed representation of the pin(s).
- ui8Val is the value to write to the pin(s).

Description:

Writes the corresponding bit values to the output pin(s) specified by ui8Pins. Writing to a pin configured as an input pin has no effect. The pin(s) are specified using a bit-packed byte, where each bit that is set identifies the pin to be accessed, and where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

Returns:

None.

To call this function, use the Forth form:

```
port pins val ROM_GPIOPinWrite
```

8.1.4 Pointer to double indirect call tables

Before use, you have to declare the address that holds the addressof the primary ROM table used for calling ROM functions.

```
variable PriPointer    \ set at powerup somehow
PriPointer setPriPointer       \ declare to cross compiler
```

Now you can define a set of ROM calls, for example, the call to the serial version of `TYPE` in a BBC micro:bit is:

```
PDIC( 2, 3 ) void serType( uint8_t *buff, int length );
```

To call this function, use the Forth form:

```
<caddr> <len> serType
```

8.1.5 Direct calls

Where the address of the routine is known at the Forth compile time, you can use a direct call.

```
DIR( addr ) int foo( int a, char *b, char c );
```

The Forth word marshalls the parameters and calls the subroutine at target address *addr*.

8.2 C comments in declarations

Rudimentary support for C comments in declarations is provided, but it is good enough for the vast majority of declarations.

- Comments can be `// ...` or `/* ... */`,
- Comments must be at the end of the line,
- Comments are treated as extending to the end of the line,
- Comments must not contain the ')' character.

The example below is taken from a *SQLite* interface.

```
SVC( x ) int sqlite3_open16(
  const void * filename,  /* Database filename [UTF-16] */
  sqlite3 ** ppDb         /* OUT: SQLite db handle */
);
```

8.3 Format

```
'SVC(' <n> ')' <return> [ <callconv> ] <name> '(' <arglist> ')' ';'
'JTI(' <n> ')' <return> [ <callconv> ] <name> '(' <arglist> ')' ';'
'DIC(' <n1>, <n2> ')' <return> [ <callconv> ] <name> '(' <arglist> ')' ';'
'PDIC(' <n1>, <n2> ')' <return> [ <callconv> ] <name> '(' <arglist> ')' ';'
'DIR(' <n> ')' <return> [ <callconv> ] <name> '(' <arglist> ')' ';'

<n>         := { literal }
<return>    := { <type> [ '*' ] |  void }
<arg>       := { <type> [ '*' ] [ <name> ] }
<args>      := { [ <arg>, ]* <arg> }
<arglist>   := { <args> | void }        Note: "void, void" etc. is illegal.
<callconv>  := { PASCAL | WINAPI | STDCALL | "PASCAL" | "C" }
<name>      := <any Forth acceptable namestring>
<type>      := ... (see below, "void" is a valid type)
```

The Forth <name> is case-insensitive.

As a standard Forth's string length for dictionary names is only guaranteed up to 31 characters for portable source code, long API names can cause problems.

8.4 Calling Conventions

In the discussion **caller** refers to the Forth system (below the application layer and **callee** refers to a a foreign function. The `SVC()` only supports the AAPCS standard as used by C.

When using calls to floating point libraries, there are three possible calling conventions, `noFPAPI`, `softFPABI) and *\fo{hardFPABI`.

- *\fo{noFPABI): No floating point support is provided. The value `FPABI` is set to 0.
- *\fo{softFPABI): Floating point numbers are passed to and from the system in integer registers or on the stack. The value `FPABI` is set to 1.
- `hardFPABI`: Floating point numbers are passed to and from the system in the VFP registers. The value `FPABI` is set to 2.

For now, the only combination supported by the cross compiler uses **FPsystem=2** and **FPABI=2**, i.e 32 bit VFP floats, a separate Forth float stack, and a hardfloat ABI.

8.5 Promotion and Demotion

The system generates code to either promote or demote non-CELL sized arguments and return results which can be either signed or unsigned. Although Forth is an un-typed language it must deal with libraries which do have typed calling conventions. In general the use of non-CELL arguments should be avoided but return results should be declared in Forth with the same size as the C or PASCAL convention documented.

8.6 Argument Reversal

The default calling convention for AAPCS is used. The right-most argument/parameter in the C-style prototype is on the top the Forth data stack. When calling an external function the parameters are reordered as required by the AAPCS; this is to enable the argument list to read left to right in Forth source as well as in the C-style operating system documentation.

8.7 Controlling external references

`: setFPABI ( u -- ) to FPABI ;`
When using calls to floating point libraries, there are three possible values, 0 for `noFPAPI`, 1 for\fo{softFPABI) and 2 for `hardFPABI`. At present, only the `hardFPABI` is supported.

`: setFPsystem ( u -- ) to FPsystem ;`
Defines which floating point pack is installed and active, 0 for none, 1 for software floating point, 2 for VFP2 and 3 for VFP3. At present, only the VFP2 option is supported, which is binary compatible with VFP3.

`1 value saveR9? \ --`
Set true if R9 is to be saved by calls.

`1 value saveR12? \ --`
Set true if R12 is to be saved by calls.

`: SVC( \ "text" -- ; )`
Declare an external API reference of the form:

```
SVC( n ) int foo( int a, char *b, char c );
```

The Forth word marshalls the parameters and calls SVC/SWI n. The Forth word has the same name as the function in the library, but the Forth word name is **not** case-sensitive. The length of the function's name may not be longer than a Forth word name.

`: holdsJumpTable \ addr(t) --`
The base address of the jump table is stored at the given target address. `HoldsJumpTable` must be used before any `JTI(` definition is made.

```
variable JT     \ -- addr
JT holdsJumpTable
```

`: JumpTable \ -- addr`
Returns the target address that holds the jump table base address

`: JTI( \ "text" -- ; )`
Declare an external API reference of the form:

```
JTI( n ) int foo( int a, char *b, char c );
```

The Forth word marshalls the parameters and calls entry n (0, 1, 2, ...) in the jump table The Forth word can have the same name as the function in the library, but the Forth word name is **not** case-sensitive. The length of the function's name may not be longer than a Forth word name.

`: DIR( \ "text" -- ; )`
Declare an external API reference of the form:

```
DIR( addr ) int foo( int a, char *b, char c );
```

The Forth word marshalls the paramters and calls the subroutine at target address *addr*. The Forth word can have the same name as the function in the library, but the Forth word name is **not** case-sensitive. The length of the function's name may not be longer than a Forth word name.

`: setPriTable   \ addr(t) --`
Set the base address of the Primary ROM table used for calling ROM functions. For Luminary/TI CPUs, this will probably be:

```
$0100:0010 setPriTable
```

`: DIC(          \ "text" -- ; )`
Declare an external API reference of the form:

```
DIC( pri, sec ) int foo( int a, char *b, char c );
```

The comma betwee *pri* and *sec* is optional. This form is used when calling ROM routines in devices from TI, NXP and others. In these the primary table entries hold the address of another table. The secondary index is then used to fetch the address of the actual routine to be called. The Forth word marshalls the parameters and calls the routine. The Forth word can have the same name as the function in the library, but the Forth word name is **not** case-sensitive. The length of the function's name may not be longer than a Forth word name.

`: setPriPointer \ addr(t) --`
Set the base address of the Primary ROM table used for calling ROM functions. For Luminary/TI CPUs, this will probably be:

```
$0100:0010 setPriTable
```

`: PDIC(         \ "text" -- ; )`
Declare an external API reference of the form:

```
PDIC( pri, sec ) int foo( int a, char *b, char c );
```

The comma betwee *pri* and *sec* is optional. This form is used when calling ROM routines in devices from TI, NXP and others. In these the primary table entries hold the address of another table. The secondary index is then used to fetch the address of the actual routine to be called. The Forth word marshalls the parameters and calls the routine. The Forth word can have the same name as the function in the library, but the Forth word name is **not** case-sensitive. The length of the function's name may not be longer than a Forth word name.

`: +ForceTbits   \ --`
Force function addresses to have the T bit (bit 0) set by the code that performs the call. This is the default.

`: -ForceTbits   \ --`
Do not compile the code that forces function addresses to have the T bit (bit 0) set by the code that performs the call. You will have to dump a table to find this out for your system.

`: +SaveR9       \ --`
Compile additional code to preserve R9 (platform register) across calls. Some silicon vendors do not know what various compilers do, so the safe thing is to use `+SaveR9`.

`: -SaveR9       \ --`
Do not compile additional code to preserve R9 across calls.

`: +SaveR12      \ --`

Compile additional code to preserve R12 (Intra-Procedure call scratch register) across calls. Some silicon vendors do not know what various compilers do, so the safe thing is to use `+SaveR12`.

`: -SaveR12 \ --`
Do not compile additional code to preserve R12 across calls.

`: +DebugExterns \ --`
Turn on display of debug information.

`: -DebugExterns \ --`
Turn off display of debug information.

8.8 Pre-Defined parameter types

The types known by the system are all found in the vocabulary `TYPES(t)`. You can add new ones at will. Each `TYPE` definition modifies one or more of the following `VALUE`s.)

`argSIZE` Size in bytes of data type.

`argDEFSIGN`
Default sign of data type if no override is supplied.

`argREQSIGN`
Sign OverRide. This and the previous use 0 = unsigned and 1 = signed.

`argISPOINTER`
1 if type is a pointer, 0 otherwise

Each `TYPES(t)` definition can either set these flags directly or can be made up of existing types.

8.8.1 Calling conventions

`: "C" \ --`
Set Calling convention to "C" standard. Arguments are reversed, and the caller cleans up the stack. This is the default.

`: "PASCAL" \ --`
Set the calling convention to the "PASCAL" standard as used by Pascal compilers. Arguments are **not** reversed, and the called routine cleans up the stack.

`: L>R \ --`
By default, arguments are assumed to be on the Forth stack with the top item matching the rightmost argument in the declaration so that the Forth parameter order matches that in the C-style declaration. `L>R` confirms this and is the default.

`: R>L \ --`
By default, arguments are assumed to be on the Forth stack with the top item matching the rightmost argument in the declaration so that the Forth parameter order matches that in the C-style declaration. `R>L` reverses this.

8.8.2 Basic Types

`: unsigned \ --`
Request current parameter as being unsigned.

`: signed \ --`
Request current parameter as being signed.

`: int \ --`

Declare parameter as integer. This is a signed 32 bit quantity unless preceeded by `unsigned`.

`: char \ --`
Declare parameter as character. This is a signed 8 bit quantity unless preceeded by `unsigned`.

`: void \ --`
Declare parameter as void. A `VOID` parameter has no size. It is used to declare an empty parameter list, a null return type or is combined with `*` to indicate a generic pointer.

`: * \ --`
Mark current parameter as a pointer.

`: ** \ --`
Mark current parameter as a pointer.

`: *** \ --`
Mark current parameter as a pointer.

`: const ; \ --`
Marks next item as **constant** in C terminology. Ignored by Forth.

`: int32 \ --`
A 32bit signed quantity.

`: int16 \ --`
A 16 bit signed quantity.

`: int8 \ --`
An 8 bit signed quantity.

`: uint32 \ --`
32bit unsigned quantity.

`: uint32_t \ --`
32bit unsigned quantity.

`: uint16 \ --`
16bit unsigned quantity.

`: uint16_t \ --`
16bit unsigned quantity.

`: uint8 \ --`
8bit unsigned quantity.

`: uint8_t \ --`
8bit unsigned quantity.

`: LongLong \ --`
A 64 bit signed or unsigned integer. At run-time, the argument is taken from the Forth data stack as a normal Forth double with the top item on the top of the data stack.

`: LONG int ;`
A 32 bit signed quantity.

`: SHORT \ --`
For most compilers a **short** is a 16 bit signed item, unless preceded by `unsigned`.

`: BYTE \ --`
An 8 bit unsigned quantity.

`: float \ --`
32 bit float.

`: double \ --`
64 bit float.

`: bool1 \ --`
One byte boolean.

`: bool4 \ --`
Four byte boolean.

`: ... \ --`
The parameter list is of unknown size. This is an indicator for a C varargs call. Run-time support for this varies between CPUs and operating system implementations of VFX Forth. Test, test, test.

9 JLink Flash and debug interface

The compiler can be integrated with a Segger JLink to provide Flash programming and debugging facilities. Instructions for downloading the required software will have been provided with your JLink. The Segger website is at `http://www.segger.com`. Make sure that the installed Segger shared library is at least v4.78e or v4.78.5.

These facilities are available for Windows, Mac OS X and Linux.

9.1 Installing Segger software

9.1.1 Windows

Download the latest Segger tools; you need at least v4.78e or v4.78.5. Place the file *JLinkARM.dll* in a directory in the search path. The directory containing the compiler is always in the search path.

9.1.2 Mac OS X

Download the latest Segger tools; you need at least v4.78e or v4.78.5. We need the shared library *libjlinkarm.4.xx.y.dylib* and the link *libjlinkarm.4.dylib*. Check that the following file exists:

```
/Applications/Segger/JLink/libjlinkarm.dylib
```

If it is present, you are all done. If it does not exist, either the installation has failed or Segger have changed the rules. Note that this file is a symbolic link.

For the 64 bit compiler, put the shared libraries *libjlinkarm.*.dylib* in */usr/local/lib*. If the cross-compiler cannot find the Segger library, make a symbolic link *libjlinkarm.dylib* to it in */usr/local/lib*.

You will have to repeat this incantation whenever you update the Segger tools.

9.1.3 Linux

Download the latest Segger tools; you need at least v4.78e or v4.78.5. After installation or update, you should have a directory */opt/SEGGER/JLink* containing the files *libjlinkarm.so.4* and *libjlinkarm.so.4.xx.y*. Copy these to */usr/lib* or a system directory which is searched for 32 bit libraries. For example:

```
sudo  cp /opt/SEGGER/JLink/libjlinkarm.so.* /usr/lib/
```

You will have to repeat this incantation whenever you update the Segger tools.

9.2 Controlling the J-Link

JLink commands can be executed at any time, either in the control file or interactively (at the keyboard). However, the memory image and output files are not finalised until the word `FINIS` has been executed. Actions that must be performed after `FINIS` can be specified with `AFTERWARDS`, which saves the rest of the line as text to be interpreted at the end of `FINIS`.

AFTERWARDS can be used several times, and each line is interpreted in order, first encountered being interpreted first. The example below sets up the JLink and then causes the code sections to be downloaded to Flash.

```
jlink STM32F407ZG 0 Project.jlink \ device, speed, project file
afterwards  download-sections   \ download image
```

The list of device names can be found in the file <*jlink*>*ETC**JFlash**MCU.csv* supplied by Segger. The speed (0=automatic) and project file (Project.jlink) are defaults and rarely need to be changed.

9.2.1 Umbilical Forth

At the end of the control file, you can place something like:

```
...

jlink LPC1114/302 0 Project.jlink  \ device, speed, project file
useSWD                  \ use SWD rather than the default JTAG
PSUART0 set-sjl-link    \ select Umbilical link through J-Link

umbilical-forth
```

The word jlink sets the device (important), the speed (0=autodetect to maximum) and project file. By default, the J-Link operates in JTAG mode. For systems that require SWD operation useSWD selects SWD operation. We can fake a serial line for the Umbilical Forth link using set-sjl-link which takes the address of the four-byte data used for talking to the J-Link.

The file *Cortex/Drivers/serPSUART.fth* contains an example link driver for the target side. It's rather slow because of the USB connection to the J-Link, but it works.

If you have a system with a real serial port, you can still use it but you will have to replace the line

```
PSUART0 set-sjl-link    \ select Umbilical link through J-Link
```

with serial configuration lines such as

```
c" COM4:" console-speed serial
c" dtr=off rts=off" set-control
```

9.3 Initialisation and download

```
: JLINK         \ -- ; jlink <device> <speed> <projfile>
```

Set the JLink to use the given device, connection speed and the given project file. The list of devices can be found in the file <*jlink*>*ETC**JFlash**MCU.csv* supplied by Segger. The connection speed is specified in kHz, where 0 specifies automatic speed detection, and 65535 specifies adaptive clocking. For example:

```
jlink STM32F407ZG 1000 Project.jlink
```

selects an STM32F407ZG device, 1000 kHz and the project file is *Project.jlink* in the compiler's working directory.

`: speedJLink \ kHhz --`
Set the JLink speed in kHz. A value of zero selects automatic detection of JLink speed. A value of 65535 selects adaptive clocking. This word is useful when a CPU runs at a low speed after reset, e.g. at 32 kHz until the clock unit has been set up.

`: useJTAG \ --`
The JLink connects to the target by JTAG. This is the default.

`: useSWD \ --`
The JLink connects to the target by SWD.

`: download-sections \ --`
After `JLINK` above has been used, `download-sections` is set to use the JLink device to program the target. `Download-sections` closes the JLink connection after programming.

`: InitJLink \ --`
Initialise the J-Link connection.

`: TermJLink \ --`
Terminate the J-Link connection.

`: ProgramImage \ taddr "<filename>" -- ; 0 ProgramImage File.bin`
Program the file at the given target address.

`: setImageFile \ "<filename>" -- ; ImageFile <filename>`
Set the predefined image file used by commands such as `DownloadJLink` below.

`: setFlashStart \ taddr -- ; $0800:0000 setFlashStart`
Set the predefined Flash start address used by commands such as `DownloadJLink` below.

`: DownloadJLink \ --`
Download the image file predefined by `ImageFile` to the target.

9.4 Debug and stepping

The word `InitJLink` must have been used before any of these words have been run.

`: reg@ \ reg# -- x`
Return the data in the saved register (0..15). The CPU must have been halted.

`: reg! \ x reg# --`
Apply the data to the saved register (0..15).

`: .regs \ --`
Display the working registers.

`: stop \ --`
Stop the CPU.

`: StopCPU \ -- flag`
Return true if the CPU was previously running. If the CPU is running, halt it.

`: from \ addr --`
Set the target PC to the given address

```
: go            \ -- ; target carries on
```
Carry on execution of the target code.

```
: halt          \ --
```
Halt the CPU and dump registers.

```
: step          \ --
```
Execute a single step. There is no display.

```
: ss            \ --
```
Execute a single step and then display registers

```
: stepOver      \ --
```
If the current instruction is an unconditional BL instruction, step without display until it returns. Otherwise perform a single step. There is no display until the end.

```
: so            \ --
```
Step past the current instruction and then display registers.

```
: resetOnly     \ --
```
Reset the CPU with no checks.

```
: resetGo       \ --
```
Reset the target and set it to free run if the CPU had been halted.

```
: halted?       \ -- flag
```
Return true if the CPU is halted.

```
: running?      \ -- flag
```
Return true if the CPU is running.

9.5 Breakpoints

```
: setBP         \ addr -- bp# ; 0=failure
```
Set a breakpoint at the given address and return the breakpoint number.

```
: clrBP         \ bp# --
```
Clear the given breakpoint.

```
: clrAllBPs     \ --
```
Clear all breakpoints.

```
: waitHalt      \ --
```
Wait until the CPU halts, e.g. for a breakpoint, or until the <ESC> key is pressed.

9.6 Memory driver

Unless otherwise set, the cross compiler defaults to reading and writing target memory images in host memory. When a JLink or other hardware provides access to real target memory, the memory drivers can be selected to read and write the real target memory.

```
: all-via-jlink         \ --
```
Switch all memory sections to use the JLink to access target memory.

```
: all-via-default       \ --
```
Switch all memory sections to use the default memory image driver.

```
: [image        \ --
```

Start compiling or other actions that affect Flash. The compilation is performed to a memory image, and the result is written to Flash when `image]` is executed.

```
: image]        \ --
```

Finish compiling to memory, update the Flash and restore the memory drivers.

9.7 J-Link pseudo serial port

We use four bytes of target memory as a pretend UART (PSUART). This allows us to run debug communications and Umbilical Forth through the J-Link using no resources except the four bytes of RAM.

```
: set-sjl-link  \ addr --
```

Set the base address in the target at which the pseudo-UART four byte RAM block is located. The Umbilical Forth link drivers are also set to use the pseudo UART.

9.8 Interactive debugging

When you build a standalone target, you can still debug it using the J-Link. If the compiler directive `INTERACTIVE` has been used the compiler stays alive after executing `FINIS`. you can then use any of the commands as required. In order to permit access to any memory block in the target, we first modify our usual `SECTION` definitions in the control file.

```
$0000:0000 $FFFF:FFFF udata section CATCHALL
$0000:0000 $0000:7FFF cdata section LPC1114ufjl        \ code section
$1000:0200 $1000:0FFF idata section PROGd      \ 4k IDATA
$1000:1000 $1000:1FFF udata section PROGu      \ 4k UDATA
```

The `CATCHALL` section provides a valid memory area for every target memory address. It must be the first `SECTION` defined.

After `FINIS`, the compiler stays alive. You can now enter commands interactively at the keyboard.

```
all-via-jlink
Section PROGU switched
Section PROGD switched
Section K60TWR switched
Section CATCHALL switched ok
0 100 dump
0000:0000 00 FB 00 20 AD 05 00 00 ED 56 00 00 3D 57 00 00  .{. -...mV..=W..
0000:0010 8D 57 00 00 DD 57 00 00 2D 58 00 00 00 00 00 00  .W..]W..-X......
0000:0020 00 00 00 00 00 00 00 00 00 00 00 00 0F 02 00 00  ................
0000:0030 7D 58 00 00 00 00 00 00 13 02 00 00 EB 72 00 00  }X..........kr..
0000:0040 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0000:0050 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0000:0060 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
 ok
```

The first command, `ALL-VIA-JLINK`, switches the memory drivers to use the J-Link. The second command dumps a block of memory. You can also use the debugging commands.

```
halt
R0  = 1FFF:0120 FFFF:FFFF 0000:37A9 0000:0000
R4  = 0000:0020 0000:001E 0000:0000 1FFF:00F4
R8  = 0000:0000 0000:0000 0000:0000 2000:FF00
R12 = 2000:FEF0 2000:FDE0 0000:5B3F 0000:6E18
XPSR = 2100:0000   MSP = 2000:FB00   PSP = 2000:FDE0
( 0000:6E14 7047 pG )        bx LR ok
ss
R0  = 1FFF:0120 FFFF:FFFF 0000:37A9 0000:0000
R4  = 0000:0020 0000:001E 0000:0000 1FFF:00F4
R8  = 0000:0000 0000:0000 0000:0000 2000:FF00
R12 = 2000:FEF0 2000:FDE0 0000:5B3F 0000:5B42
XPSR = 2100:0000   MSP = 2000:FB00   PSP = 2000:FDE0
( 0000:5B3E F5E7 ug )        b # $5B2C ok
ss
R0  = 1FFF:0120 FFFF:FFFF 0000:37A9 0000:0000
R4  = 0000:0020 0000:001E 0000:0000 1FFF:00F4
R8  = 0000:0000 0000:0000 0000:0000 2000:FF00
R12 = 2000:FEF0 2000:FDE0 0000:5B3F 0000:5B30
XPSR = 2100:0000   MSP = 2000:FB00   PSP = 2000:FDE0
( 0000:5B2C FCF794FC |w.| )  bl # $2458      [I ok
go  ok
```

10 SVD file converter

The code here converts ARM CMSIS System View Description (SVD) files to a set of Forth equates for a cross compiler. These equates are in the same format as the existing "Special Function Register" files usually named *sfr<cpu>.fth* in the *Cortex* target directory. Processing SVD files automatically to produce Forth source significantly reduces the effort involved in porting Forth to a CPU. This code requires the use of the file *XML.fth* supplied with VFX Forth. In the cross compiler, the files are in the *ArmCortex/SVDparser* or *compiler/SVDparser* directory, depending on the edition of the compiler.

```
include XML.fth
include SVDconv.fth
```

The converted SVD (XML) to Forth is output using `TYPE` and `EMIT` and friends. This can be copied to an editor or the output redirected to a file. See `SVDconv` at the end of this chapter if you want the output sent to a file. Otherwise just `include` the SVD file.

```
include ATSAM4SD32C.svd
svdconv ATSAM4SD32C.svd  sfrATSAM4SD32C.raw.fth
```

Because of lack of structure namespaces in Forth, most register names will need a prefix to identify the peripheral the register applies to. In the SVD specification, this is achieved using the *prependToName* tag, e.g.

```
<prependToName>HSMCI_</prependToName>
```

for an MMC/SD card peripheral. MPE standard practice is to use a short prefix such as "mmc.". To use these, add a tag line such as

```
<mpePrepends>mmc.</mpePrepends>
```

to each peripheral section before the first register definition. Search for <peripheral> when editing the SVD file. You can select whether or not text is added to register names using the directives `-prepends`, `+prependC` and `+prependMPE` below. The default is `+prependMPE`.

The resulting output may/will require a little hand editing. Many SVD files repeat the register definitions for identical or similar peripherals such as UARTs. By using the same prefices, you can remove the repetitions. Providing that the values are the same, redefinitions of `EQU`ates are harmless, albeit noisy and inelegant. Personally, I prefer to remove the duplicates as it reduces the size of the file.

10.1 Time and date

These functions rely on the ANS Forth word `TIME&DATE ( -- s m h dd mm yyyy )` and the non-standard `DOW ( -- dow, 0=Sun)` to get the day of the week.

`create days$    \ -- addr`
String containing 3 character text for the days of the week.

`create months   \ -- addr`
String containing 3 character text for the months.

```
: .dow          \ dow --
```
Display day of week.

```
: .2r           \ n --
```
Display n as a two digit number with leading zeros.

```
: .4r           \ n --
```
Display n as a four digit number with leading zeros.

```
: .Time&Date    \ s m h dd mm yy --
```
Display the system time The format is:

```
hh:mm:ss dd Mmm yyyy
```

```
: .AnsiDate     \ zone --
```
Display the day of week, date and time. If zone is 0 GMT (system time) is displayed, otherwise local time is displayed. The format is:

```
dow, hh:mm:ss dd Mmm yyyy [GMT]
```

10.2 Data buffers and variables

SVD files are fairly simple as XML files go, so data is extracted from tags and saved in global or dynamic buffers.

```
1024 buffer: opfile$    \ -- addr
```
If set, the name of the output file. 16 bit Word counted string.

```
32 kb buffer: description$      \ -- addr
```
Contents of the <description> tags. 16 bit Word counted string.

```
1024 buffer: name$      \ -- addr
```
Contents of the <name> tags. 16 bit Word counted string.

```
1024 buffer: revision$  \ -- addr
```
Contents of the <revision> tags. 16 bit Word counted string.

```
1024 buffer: endian$    \ -- addr
```
Contents of the <endian> tags. 16 bit Word counted string.

```
1024 buffer: mpupresent$        \ -- addr
```
Contents of the <mpupresent> tags. 16 bit Word counted string.

```
1024 buffer: fpupresent$        \ -- addr
```
Contents of the <fpupresent> tags. 16 bit Word counted string.

```
1024 buffer: nvicPrioBits$      \ -- addr
```
Contents of the <nvicPrioBits> tags. 16 bit Word counted string.

```
1024 buffer: vendorSystickConfig$       \ -- addr
```
Contents of the <vendorSystickConfig> tags. 16 bit Word counted string.

```
1024 buffer: prepend$   \ -- addr
```
Holds the text prepended to each register name. This text is cleared by each *peripheral* tag. 16 bit Word counted string.

```
1024 buffer: clusName$  \ -- addr
```
Holds <name> from last <cluster>. 16 bit Word counted string.

```
0 value valueData       \ -- int
```

Integer from the contents of the <value> tag.

```
0 value dimData \ -- int
```
Integer from the contents of the <dim> tag.

```
0 value inPeriph?       \ -- flag
```
True between <peripheral> and </peripheral>.

```
0 value PeriphHead?     \ -- flag
```
True when the peripheral header has been written.

```
0 value BaseAddrValue   \ -- addr
```
Peripheral base address from the <baseAddress> tag.

```
0 value AddrOffset      \ -- offset
```
Offset from the <AddressOffset> tag.

```
0 value ClusterOffset   \ -- addr
```
Offset from the <AddressOffset> tag inside a cluster.

```
0 value InReg?          \ -- flag
```
True between <register> and </register>.

```
0 value RegWritten?     \ -- flag
```
True when the register data has been written.

```
0 value InCluster?      \ -- flag
```
True between <cluster> and </cluster> tags.

```
0 value ClusterWritten? \ -- flag
```
True when the cluster header data has been written.

```
0 value prependC?       \ -- flag
```
True to prepend C text to register names. This text is provided by lines in the SVD file such as

```
  <prependToName>HSMCI_</prependToName>
```

```
0 value prependMPE?     \ -- flag
```
True to prepend MPE text to register names. This text is provided by lines (that you must add) in the SVD file such as

```
  <mpePrepends>mmc.</mpePrepends>
```

```
: -contents     \ --
```
Discard the contents of the last tag.

```
: saveTagData   \ spad dest --
```
Save the tag data (as given by the spad), to a given buffer.

```
: saveContents  \ dest --
```
Save the last tag's contents to a buffer.

```
: contents>val  \ -- int
```
Convert the last tag's contents to an integer.

```
: \n?           \ caddr len -- caddr len flag
```
Return true if the string starts with '\n'.

```
: ?ignLineFeed  \ caddr len -- caddr' len
```
Step over \n, ignore following LF or CRLF.

```
: .periphDesc   \ buffer len --
```

Output a string translating \n pairs.

```
: ?.PeriphHead  \ -- ; SFP002
```
Output the peripheral header if not already done.

```
: ?.%s          \ --
```
Output the value of the <dim> tag.

```
: .regDesc      \ buffer len --
```
Output a string translating \n pairs.

```
: ?.Register    \ -- ; SFP002
```
Output the register line if not already done.

```
 $xx equ regname   \ description
```

```
: ?.Cluster     \ -- ; SFP002
```
Output the cluster line if not already done.

```
 \ Cluster: <name> - <description>
```

```
: .MPEhead      \ --
```
Display an MPE header.

10.3 Tag handling

Tags that are handled when encountered are words of the same name in the `inputTags` vocabulary. There are separate entries for the opening and closing tags. This can be useful and is also much quieter when debugging. The tag handlers have no overall stack effect. Many closing tags just save the contents, others display data.

```
also inputTags definitions
: description   ( -- )  ;
: /description  \ --
  description$ saveContents
;
: cpu           \ --
  ." \ " opfile$ w@
  if  opfile$ w$. ." - "  then
  description$ w$.
;
: /cpu          \ --
  cr cr ." (("
  .MPEhead
  cr cr
  cr description$ w$.
  cr ." CPU: " name$ w$. ." , " revision$ w$.
  cr ."   Endian: " endian$ w$.
  cr ."   MPU: " mpupresent$ w$.
  cr ."   FPU: " fpupresent$ w$.
  cr ."   NVIC priority bits: " nvicPrioBits$ w$.
  cr ."   Vendor Systick Config: " vendorSystickConfig$ w$.
  cr ." ))"
  contents dup sdrop snew
;
: value         ( -- )  ;
```

```
: /value        ( -- )  contents>val to ValueData  ;
: dim           ( -- )  ;
: /dim          ( -- )  contents>val to dimData  ;
: name          ( -- )  ;
: /name         ( -- )  name$ saveContents  ;
: revision      ( -- )  ;
: /revision     ( -- )  revision$ saveContents  ;
: endian        ( -- )  ;
: /endian       ( -- )  endian$ saveContents  ;
: mpuPresent    ( -- )  ;
: /mpuPresent   ( -- )  mpuPresent$ saveContents  ;
: fpuPresent    ( -- )  ;
: /fpuPresent   ( -- )  fpuPresent$ saveContents  ;
: nvicPrioBits  ( -- )  ;
: /nvicPrioBits ( -- )  nvicPrioBits$ saveContents  ;
: vendorSystickConfig   ( -- )  ;
: /vendorSystickConfig  ( -- )  vendorSystickConfig$ saveContents  ;
: interrupt     ( -- )  ?.PeriphHead  ;
: /interrupt    \ --
  base @ >r decimal
  cr  ." #" ValueData #16 + . ." equ " name$ w$. s" _IRQn" type
  r> base !  -contents
;
: peripheral    ( -- )
  1 to InPeriph?  0 to PeriphHead?  0 prepend$ c!
;
: /peripheral   ( -- )  ?.PeriphHead  0 to InPeriph?  0 to PeriphHead?  -contents  ;    \ SI
: prependToName ( -- )  ;
: /prependToName ( -- )
  prependC? if  prepend$ saveContents  else  -contents  then
;
: mpePrepends   ( -- )  ;
: /mpePrepends  ( -- )
  prependMPE? if  prepend$ saveContents  else  -contents  then
;
: baseaddress   ( -- )  ;
: /baseaddress  ( -- )  contents>val to BaseAddrValue  ;
: addressOffset ( -- )  ;
: /addressOffset ( -- )  contents>val to AddrOffset  ;
: registers     ( -- )  ?.PeriphHead  ;
: /registers    ( -- )  ( key drop )  -contents  ;
: register      ( -- )  ?.Cluster  1 to inReg?  0 to RegWritten?  0 -> dimData  ;
: /register     ( -- )  ?.Register  0 to inReg?  0 to RegWritten?  -contents  ;
: cluster       ( -- )
  InCluster? abort" Nested clusters not supported"
  1 -> InCluster?  0 -> ClusterWritten?
  0 -> AddrOffset  0 -> ClusterOffset
;
: /cluster      ( -- )
  cr ."   \ </cluster>"
  0 -> InCluster?  0 -> ClusterWritten?  0 -> ClusterOffset
  -contents
```

```
;

: fields        ( -- )  s" /fields>" skipPast  ;
((
: fields        ( -- )  ?.Cluster ?.Register  ;
: /fields       ( -- )  -contents  ;
))
previous definitions
```

10.4 File version

The word `SVDconv` below redirects output to a named file.

`: -prepends     \ --`
Do not prepend text to register names.

`: +prependC     \ --`
The C names are prepended to register names. These are found in the SVD file in the form:

```
<prependToName>HSMCI_</prependToName>
```

`: +prependMPE   \ --`
The MPE names are prepended to register names. These should be in the SVD file in the form:

```
<mpePrepends>mmc.</mpePrepends>
```

Note that you will have to add these lines to the SVD file yourself.

`: SVDconv       \ -- ; SVDconv <ifile> <ofile>`
Includes an SVD file and writes the output to a named file. Use in the form:

```
SVDconv input.svd  output.fth
```

11 Further information

11.1 MPE courses

MicroProcessor Engineering runs the following standard courses, which can be held at MPE or at your own site:

- **Architectual Introduction to Forth** (AIF): A three-day course for those with little or no experience of Forth, but with some programming experience. The AIF course provides an introduction to the architecture of a Forth system. It shows, by teaching and by practical example how software can be coded, tested and debugged quickly and efficiently, using Forth's interactive abilities.
- **Embedded Software for Hardware Engineers** (ESHE): A three-day course for hardware and firmware engineers needing to construct real-time embedded applications using Forth cross-compilers. Includes multitasking and writing interrupt handlers.

Custom courses are available

- **Quick Start Course** (QSC): A very hands-on tailored course on your site using your own hardware, and includes installation of a target Forth on your hardware, approaches to writing device drivers, designing a framework for your application and whatever else you need. The course is usually three days long.
- Other custom courses we provide are for Open Boot and Open Firmware. These are derived from the AIF course above.

11.2 MPE consultancy

MPE is available for consultancy covering all aspects of Forth and real-time software and hardware development. Apart from our Forth experience, MPE staff have considerable knowledge of embedded hardware design, Windows, Linux and DOS.

Our software orbits the earth, will land on comets, runs construction companies, laundries, vending machines, payment terminals, access control systems, theatre and concert rigging, anaesthetic ventilators, art installations, trains, newspaper presses and bomb disposal machines.

We have done projects ranging from a few days to major international projects covering several years, continents and many countries. We can operate to fixed price and fixed term contracts. Projects by MPE cover topics such as:

- Custom compiler developments, including language extensions such as SNMP, and new CPU implementations,
- Custom hardware design and compiler installations,
- Portable binary system for smart card payment systems,
- Machinery controllers,
- Connecting instrumentation to web sites,
- Virtual memory systems,
- Code porting to new hardware or operating systems.

We also have a range of outside consultants covering but not limited to:

- Communications protocols

- Windows device drivers
- All aspects of Linux
- Safety critical systems
- Project management (including international)

11.3 Recommended reading

A current list of books on Forth may be found at:
`http://www.mpeforth.com/books.htm`

For an introduction to Forth, and all available in PDF or HTML:

- "Programming Forth" by Stephen Pelc. About modern Forth systems.
- "Starting Forth" by Leo Brodie. A classic, but very dated.
- "Thinking Forth" by Leo Brodie. A classic.

For more experienced Forth programmers:

- "Object Oriented Forth" by Dick Pountain
- "Scientific Forth" by Julian Noble

Other miscellaneous Forth books:

- "Forth Applications in Engineering and Industry" by John Matthews
- "Stack Machines: The New Wave" by Philip J Koopman Jr

All of these books can be supplied by MPE.

Index

. 97

!

!call 59

"

"c" 80
"pascal" 80

$

$ 58

*

* 81
** 81
*** 81

+

+arm32-cortex-a/r 14
+debugexterns 80
+fastlvs 15
+fixedlvs 15
+forcetbits 79
+leafcalls 15
+longcalls 14
+prependc 94
+prependmpe 94
+saver12 79
+saver9 79
+short-branches 13
+use-litpool 13

—

-arm32-cortex-a/r 14
-contents 91
-debugexterns 80
-fastlvs 15
-fixedlvs 15
-forcetbits 79
-leafcalls 15
-longcalls 14
-prepends 94
-saver12 80
-saver9 79
-short-branches 13
-use-litpool 13

.

... . 82
.16 57
.2r 90
.32 57
.4r 90
.64 57
.8 57
.ansidate 90
.dow 90
.f32 56
.f64 56
.i16 57
.i32 57
.i64 57
.i8 57
.mpehead 92
.periphdesc 91
.regdesc 92
.regs 85
.s16 56
.s32 56
.s64 56
.s8 56
.time&date 90
.u16 57
.u32 57
.u64 57
.u8 56

;

;code 58

?

?.%s 92
?.cluster 92
?.periphhead 92
?.register 92
?ignlinefeed 91

[

[+short-branches 13
[-short-branches 13
[0] 57
[1] 57
[2] 57
[3] 57
[4] 57
[5] 57
[6] 57
[7] 57
[i 65, 71
[image 86
[n] 57

[short-branches ... 13
[short-branches? ... 13

\

\n? ... 91

3

32bit-mode ... 14

A

addroffset ... 91
align4 ... 58
all-via-default ... 86
all-via-jlink ... 86
and! ... 18
arm-mode ... 14
arm32 ... 55
arm32? ... 55
arm7tdmi ... 14
armarch5 ... 55
armarch5? ... 56
arshift ... 16
asmcode ... 58

B

band! ... 18
baseaddrvalue ... 91
bool1 ... 82
bool4 ... 82
buffer: ... 90
byte ... 81

C

c_and! ... 16
c_band! ... 16
c_bbic! ... 16
c_bic! ... 16
c_bor! ... 16
c_bxor! ... 16
c_byterevl ... 17
c_byterevw ... 17
c_byterevws ... 18
c_byterevwz ... 18
c_c+! ... 16
c_c@s ... 16
c_dfi ... 17
c_di ... 17
c_dsb#0f ... 18
c_efi ... 17
c_ei ... 17
c_fsp! ... 17
c_fsp@ ... 17
c_isb#0f ... 18
c_or! ... 16
c_reg! ... 17
c_reg@ ... 17
c_set-sp ... 17
c_sys! ... 17
c_sys@ ... 17
c_w+! ... 16
c_w@s ... 16
c_wfe ... 18
c_wfi ... 18
c_xor! ... 16
char ... 81
clrallbps ... 86
clrbp ... 86
clusname$ ... 90
clusteroffset ... 91
clusterwritten? ... 91
code ... 58
const ... 81
contents>val ... 91
cortex-m0 ... 13, 55
cortex-m0/m1? ... 56
cortex-m0? ... 56
cortex-m1 ... 13, 55
cortex-m1? ... 56
cortex-m3 ... 14, 55
cortex-m3? ... 56
cortex-m4 ... 14, 55
cortex-m4? ... 56
cortex-m4f ... 14, 55
cortex-m4f? ... 56
cortex-m7 ... 14, 55
cortex-m7? ... 56

D

days$ ... 89
db ... 58
dd ... 58
dfi ... 65
di ... 65, 71
dic(... 79
dimdata ... 91
dir(... 78
dl ... 58
double ... 82
download-sections ... 85
downloadjlink ... 85
dw ... 58
dxb ... 58
dxl ... 58
dxw ... 58

E

efi ... 65
ei ... 65, 71
end-code ... 59
endian$ ... 90

F

float ... 82
fpupresent$ ... 90
from ... 85

G

go ... 86

H

halt 86
halted? 86
holdsjumptable 78

I

i] 65, 71
idsp? 56
image] 87
incluster? 91
initjlink 85
inlining 14
inperiph? 91
inreg? 91
int 80
int16 81
int32 81
int8 81
is-action-of 59
isleaf 15

J

jlink 84
jti(..... 78
jumptable 78

L

l>r 80
largeframes 16
leafcalls? 15
legacy-mode 14
long 81
longcalls? 14
longlong 81
loopalign 15
loopalignment 15
loopalignment? 15

M

m0/m1? 56
months 89
mpupresent$ 90

N

name$ 90
not-m0/m1? 56
nvicpriobits$ 90

O

opfile$ 90

P

pdic(..... 79
periphhead? 91
prepend$ 90
prependc? 91
prependmpe? 91
proc 59
programimage 85

R

r>l 80
reg! 85
reg@ 85
regwritten? 91
resetgo 86
resetonly 86
restore-int 72
revision$ 90
rol 18
ror 16
running? 86

S

save-int 72
savecontents 91
saver12? 78
saver9? 78
savetagdata 91
set-sjl-link 87
setbp 86
setflashstart 85
setfpabi 78
setfpsystem 78
setimagefile 85
setpripointer 79
setpritable 79
short 81
short-branches] 13
signed 80
smallframes 16
so 86
speedjlink 85
ss 86
step 86
stepover 86
stop 85
stopcpu 85
svc(..... 78
svdconv 94

T

t2-mode 13
termjlink 85
thumb-1 55
thumb-2 55
thumb? 56
thumb1? 56
thumb2? 56

U

uint16 81
uint16_t 81
uint32 81
uint32_t 81
uint8 81
uint8_t 81
unsigned 80
usejtag 85
useswd 85

V

valuedata 90
vendorsystickconfig$ 90
void 81

W

waithalt 86

List of Tables

Table 5.1: Cortex Register usage 23
Table 5.2: ARM Register usage 24
Table 5.3: Condition code mnemonics 26
Table 5.4: Number bases 30
Table 5.5: ARM instruction notation 49
Table 5.6: ARM shift instructions 51
Table 5.7: Pre-indexed addressing 52
Table 5.8: Post-indexed addressing 53
Table 7.1: ARM interrupt vectors 71